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

Hi everyone! Since there were technical difficulties with the projector today, I'm releasing the slides from today's lecture with some detailed notes accompanying them. Feel free to ask on Piazza if you have any questions.

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

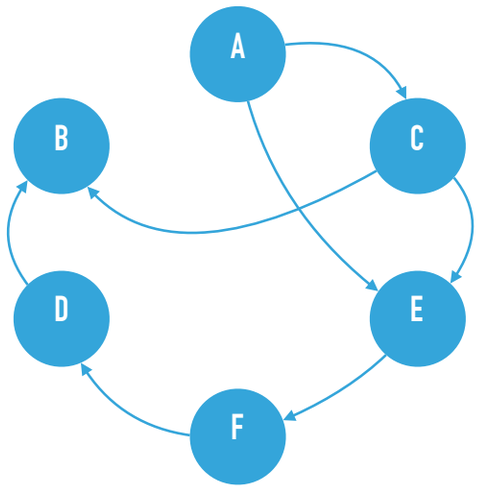
Today we're going to be talking about graphs! I'm gonna start by reviewing what we talked about on Wednesday, depth-first search and breadth-first search. Then we're going to look at some of the similarities between the two algorithms. After that, we're going to leverage some of those similarities to build a new algorithm, Dijkstra's algorithm. Then, we're going to look at a variation of Dijkstra's algorithm called A* that attempts to look into the future to perform better than Dijkstra. And if we have time, we're going to talk a little bit about how Google Maps works.

REVIEWING DFS AND BFS

So first we'll start off with some review.

DEPTH FIRST SEARCH

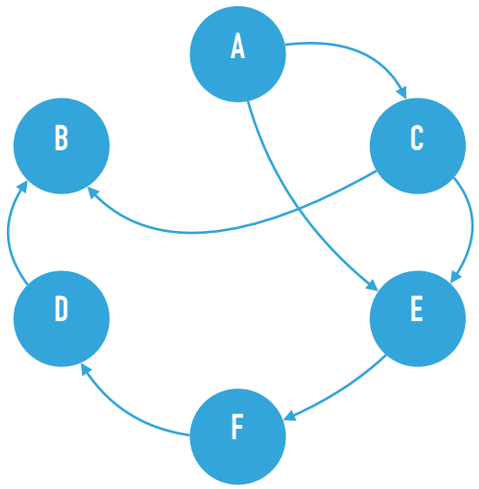
- Find a path from A to B using *iterative* depth first search
  - (Assume that nodes are pushed onto the stack in *alphabetic order*)



I'm just gonna kind of drop you all in the hot seat from the start. I have up here this graph, and I'm interested in finding a path from A to B. Can you all take a minute and try to find a path from A to B using *iterative* depth-first search? Assume that neighbors are considered in alphabetic order. And if you need a refresher on what that means, I have the pseudocode–

DEPTH FIRST SEARCH (ITERATIVE PSEUDOCODE)

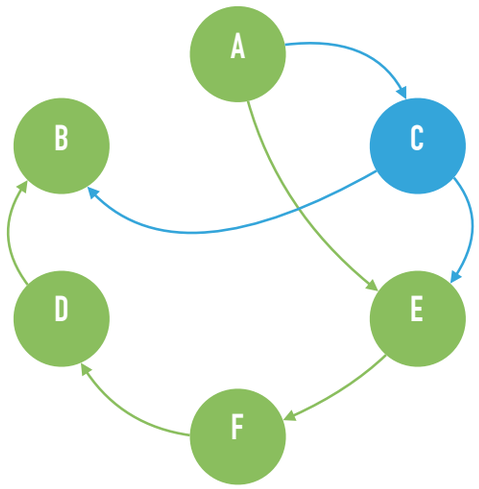
- create a path with just start node and push onto stack s
- while s is not empty
  - p = s.pop()
  - v = last node of p
  - if v is end, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - push new path onto s



–here, on this slide. Just take a minute and figure it out.

DEPTH FIRST SEARCH

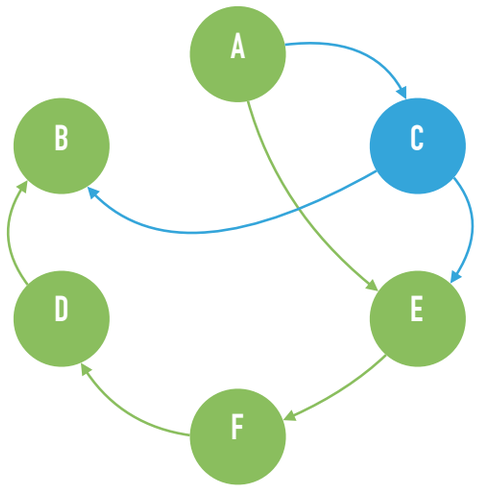
- Find a path from A to B using *iterative* depth first search
  - (Assume that nodes are pushed onto the stack in *alphabetic order*)
- A → E → F → D → B



So the first path I found running iterative DFS is this one - A to E to F to D to B. And I have a question about this path–

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?

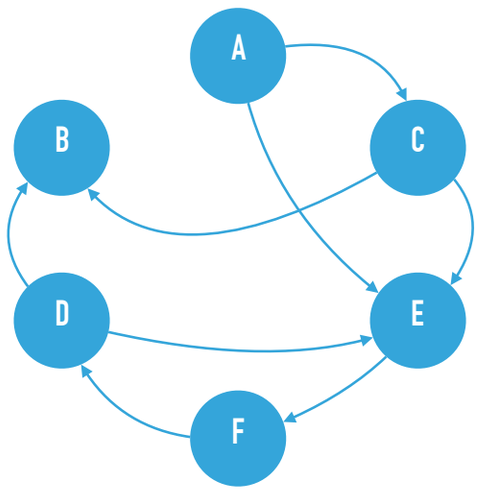


–is this the shortest path in the graph from A to B?

No, absolutely not. What is a shorter path that you all can see just by looking at the graph?

Right, there's a three node path from A to C to B. So let's look at why DFS chose this path by walking through the first few steps of the algorithm.

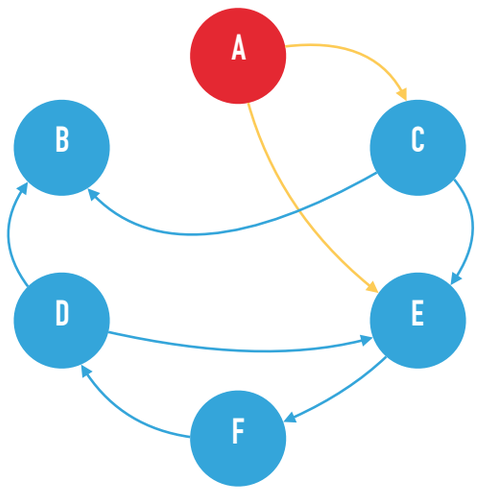
DEPTH FIRST SEARCH



Paths to Consider (Stack)

So recall that in DFS you start by pushing a path onto a stack that just contains the starting node. And then while that stack has elements in it–

DEPTH FIRST SEARCH

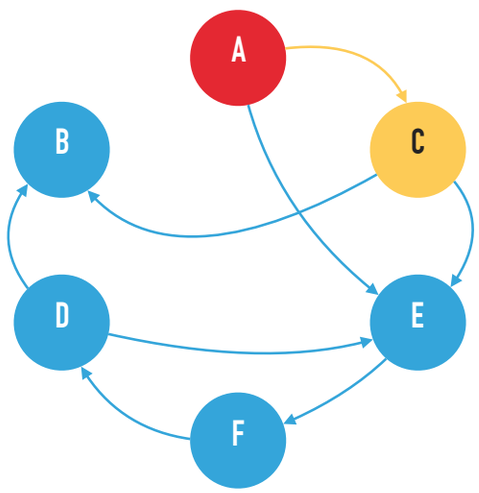


Paths to Consider (Stack)

Current Path

–you pop off a path and consider its neighbors. So here, A has two neighbors, C, and E, which we push onto the stack in alphabetical order.

DEPTH FIRST SEARCH

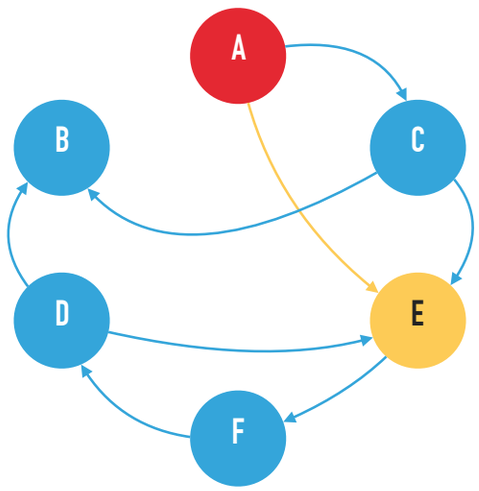


Paths to Consider (Stack)

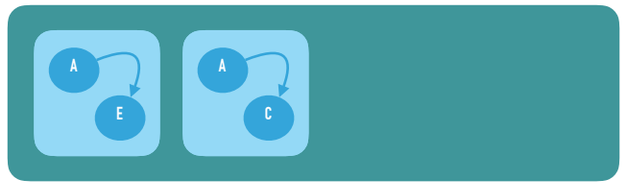
Current Path

First we push on the path from A to C...

DEPTH FIRST SEARCH



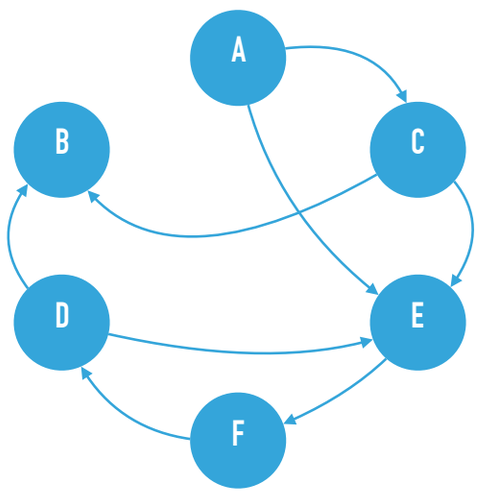
Paths to Consider (Stack)



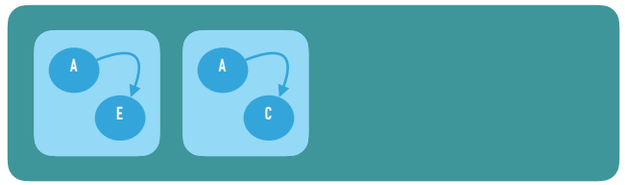
Current Path

And then we push on the path from A to E

DEPTH FIRST SEARCH



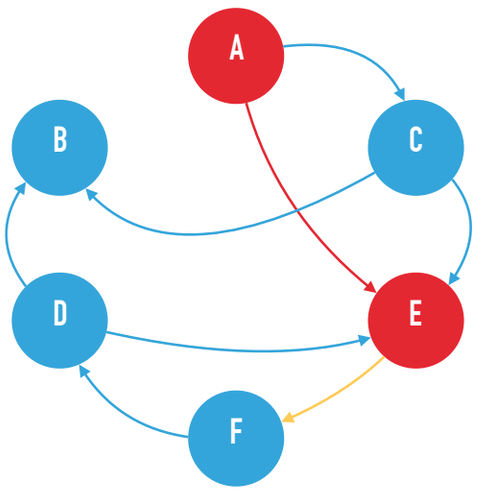
Paths to Consider (Stack)



Current Path

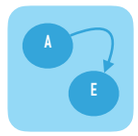
And then we dump the path with just A! We don't need it anymore.

DEPTH FIRST SEARCH



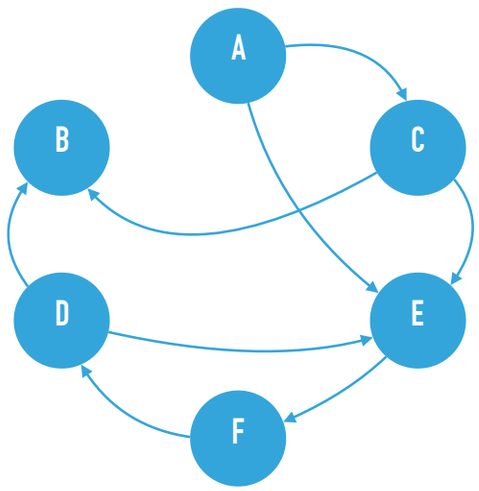
Paths to Consider (Stack)

Current Path

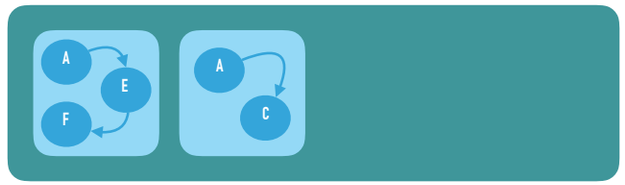


Then, the stack isn't empty. So you pop off the top element again and consider its neighbors – in this case, B and F – and then...

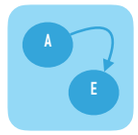
DEPTH FIRST SEARCH



Paths to Consider (Stack)

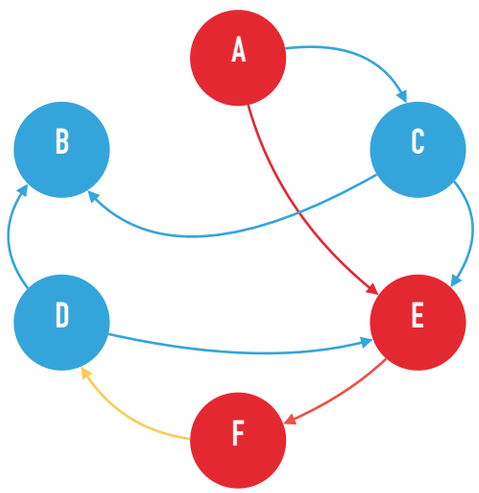


Current Path



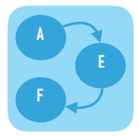
...you push them onto the stack, again in alphabetical order – so that's A to E to B, and then A to E to F.

DEPTH FIRST SEARCH



Paths to Consider (Stack)

Current Path



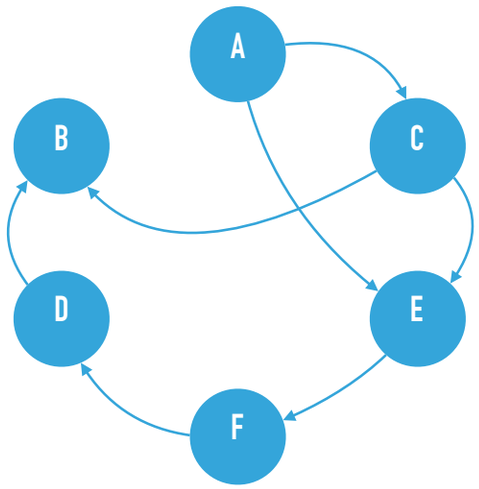
Note that there's a path that is just one hop away from finding the solution. But because A to E to F is at the top of the stack, that's the path that we consider next. And that other path, the one that is technically "closer" to the solution, it's just going to get pushed further and further down in the stack, and it'll never get considered.

You may argue that this is just because of the way that I set up the problem – I said you have to consider the neighbors in alphabetical order – but the truth of the matter is is that you can't really know for an arbitrary graph whether DFS will return the shortest path. We could change it so that we considered them in reverse alphabetical order, and that would help with this graph, but it would perform equally badly on a different graph.

So let's look at the other algorithm we talked about on Wednesday, breadth-first search.

BREADTH FIRST SEARCH

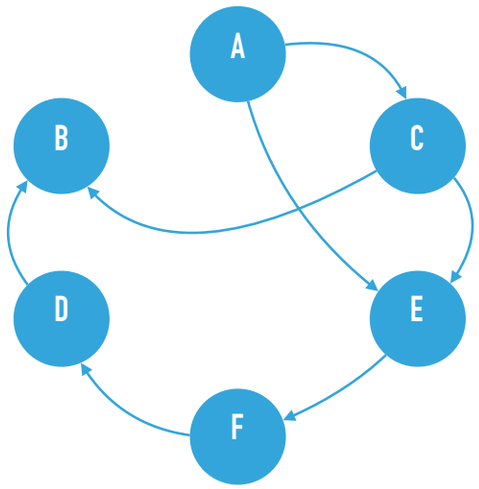
- Find a path from A to B using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)



Again, here we have the same graph. And again, I want to find a path from A to B, but this time let's use breadth first search. Again I'm going to give you a minute to work through it, and I have the pseudocode for BFS right…

BREADTH FIRST SEARCH (PSEUDOCODE)

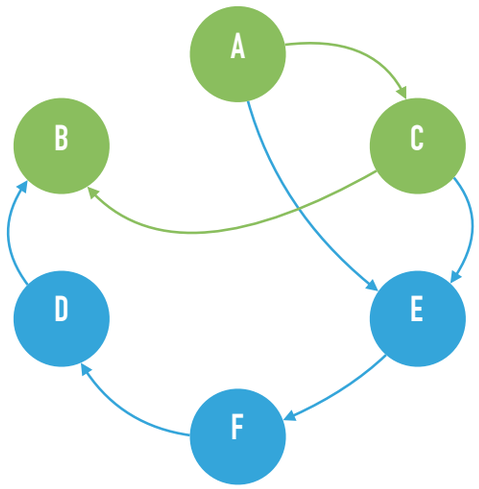
- create a path with just start node and enqueue into queue q
- while q is not empty
  - p = q.dequeue()
  - v = last node of p
  - if v is end, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - enqueue new path into q



here.

BREADTH FIRST SEARCH

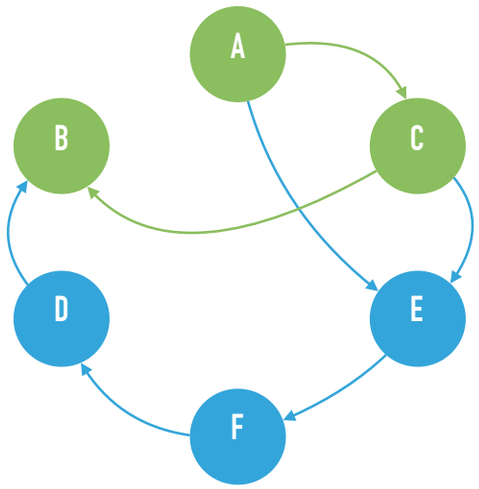
- Find a path from A to F using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)
- A → C → B



So the path I found from A to B using BFS is A to C to B. And I have a question again for you:

BREADTH FIRST SEARCH

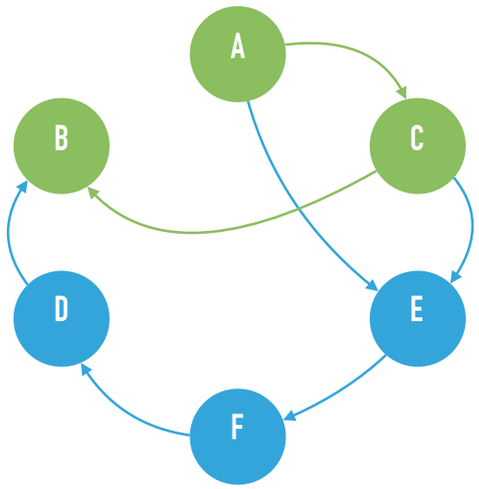
- Find a path from A to F using breadth first search
  - (Assume that nodes are pushed onto the queue in *alphabetic order*)
- A ➔ C ➔ B
- Is *this* the shortest path?



is *this* the shortest path from A to B?

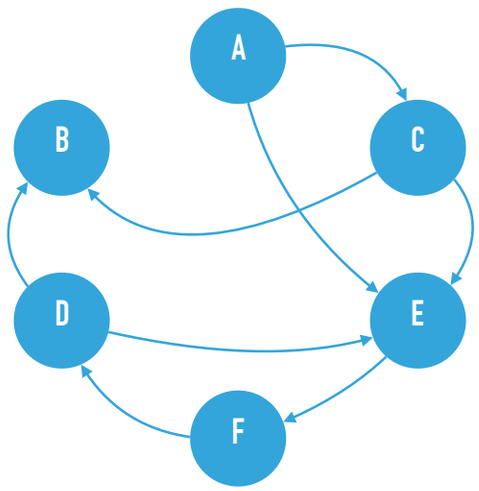
BREADTH FIRST SEARCH

- 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?
  - Yes



The answer is yes

BREADTH FIRST SEARCH

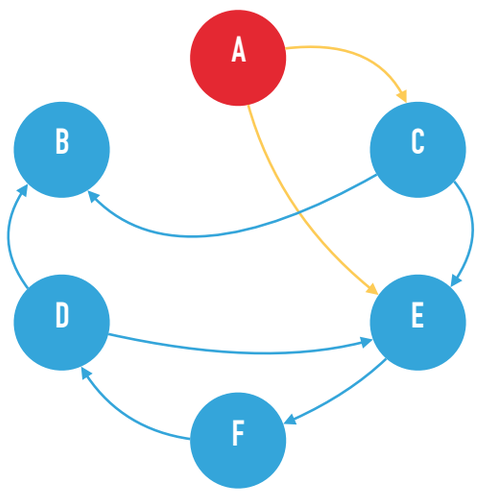


Paths to Consider (Queue)

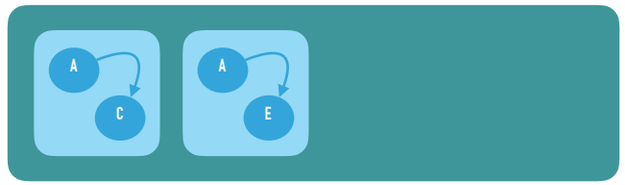
Current Path

And the reason this works is as follows. Again, we start by enqueueing a path with just the starting node.

BREADTH FIRST SEARCH



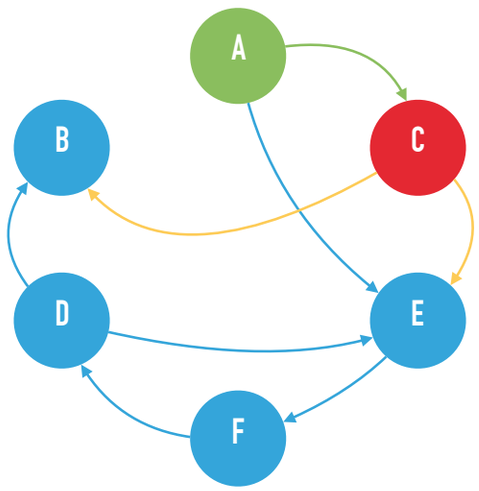
Paths to Consider (Queue)



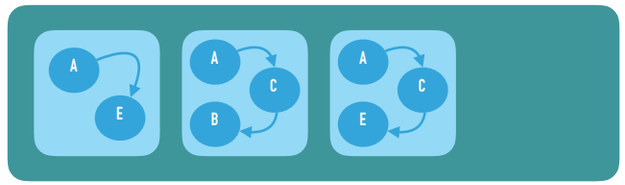
Current Path

And then we pop it off and start considering it. Again, we push them onto the queue in alphabetical order, but since it's a queue instead of a stack, they'll come out of the queue in alphabetical order.

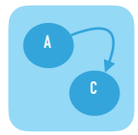
BREADTH FIRST SEARCH



Paths to Consider (Queue)

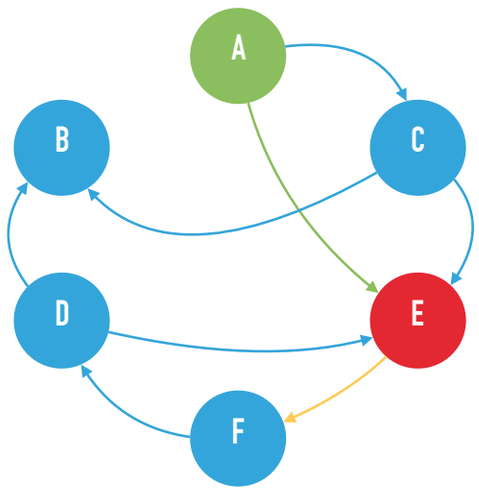


Current Path

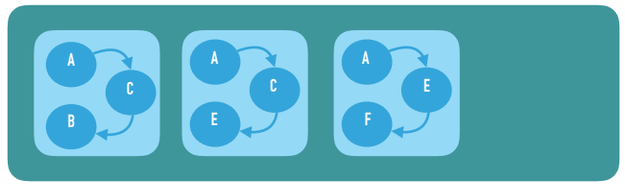


And then we enqueue the paths that have the neighbors of C.

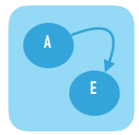
BREADTH FIRST SEARCH



Paths to Consider (Queue)



Current Path



And then the path that includes a neighbor of E.

Note that we processed all of the paths with two nodes before even looking at the paths with three nodes. And that isn't a coincidence – that's an inherent property of BFS.

YOU NEVER CONSIDER A PATH OF LENGTH K + 1

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

Generalized to all levels, with BFS, you never consider a path of length K + 1 until you've considered all paths of length K or shorter.

COMPARING DFS AND BFS

Now DFS and BFS are obviously fairly different, but they also have a lot in common. More than you might think, actually.

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

Let's look at the pseudocode for each algorithm that I had up on the slides earlier, but let's put them side by side. Does anyone see anything similar about the two?

COMPARING DFS AND BFS

DFS

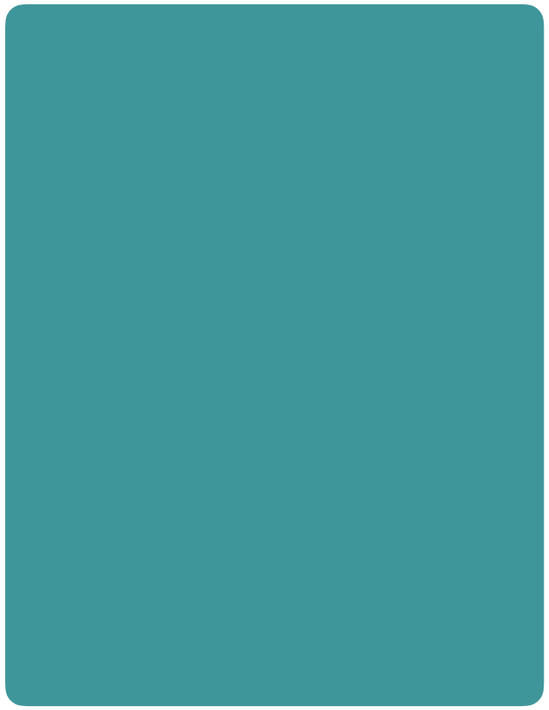
- create a path with just start node and push onto **stack s**
- while **s** is not empty:
  - p = **s.pop()**
  - v = last node of p
  - if v is end node, you're done
  - mark v as visited
  - for each unvisited neighbor:
    - create new path and append neighbor
    - **push new path onto s**

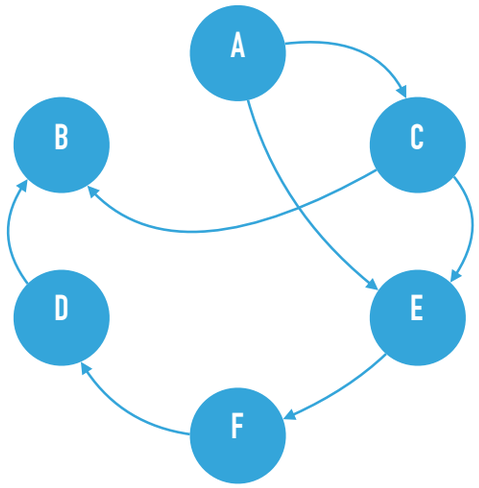
BFS

- 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**

Almost everything! The only difference is the DFS uses a stack, and BFS uses a queue. These two graph algorithms that produce substantially different results from one another are actually only different in one data structure.

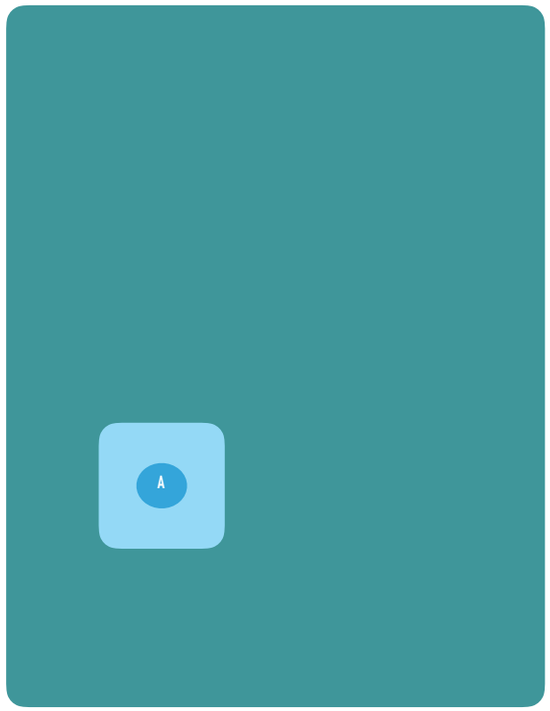
THE GRAPH SEARCH TO-DO LIST

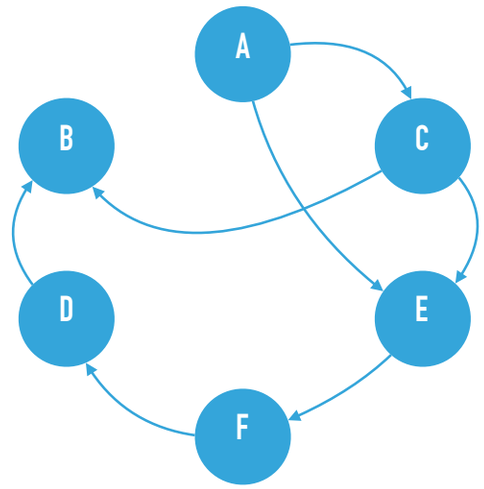




Chris Piech – not to be confused with our Chris – has a way of describing this as the "to-do list" structure. For now, we'll dispense with the idea that whatever we're putting our paths in has any particular structure. Or rather, we know that it has a structure, but for now we don't specify what it is.

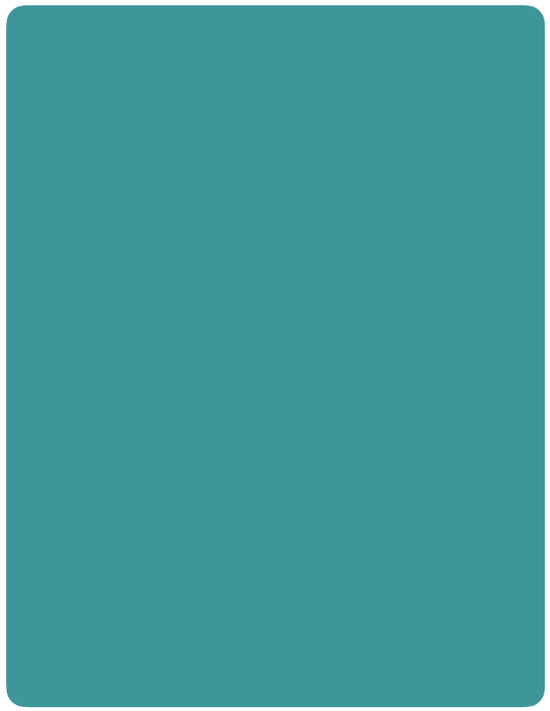
THE GRAPH SEARCH TO-DO LIST

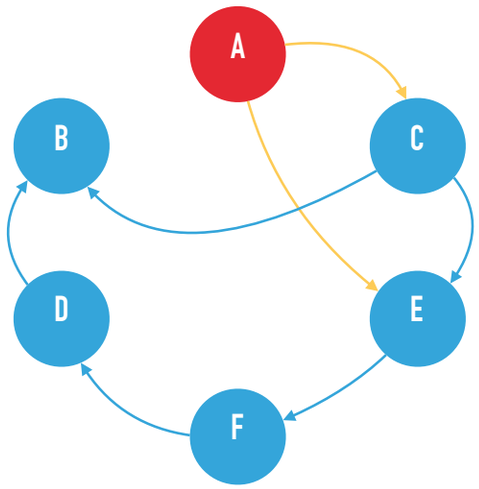




You put paths into the todo list

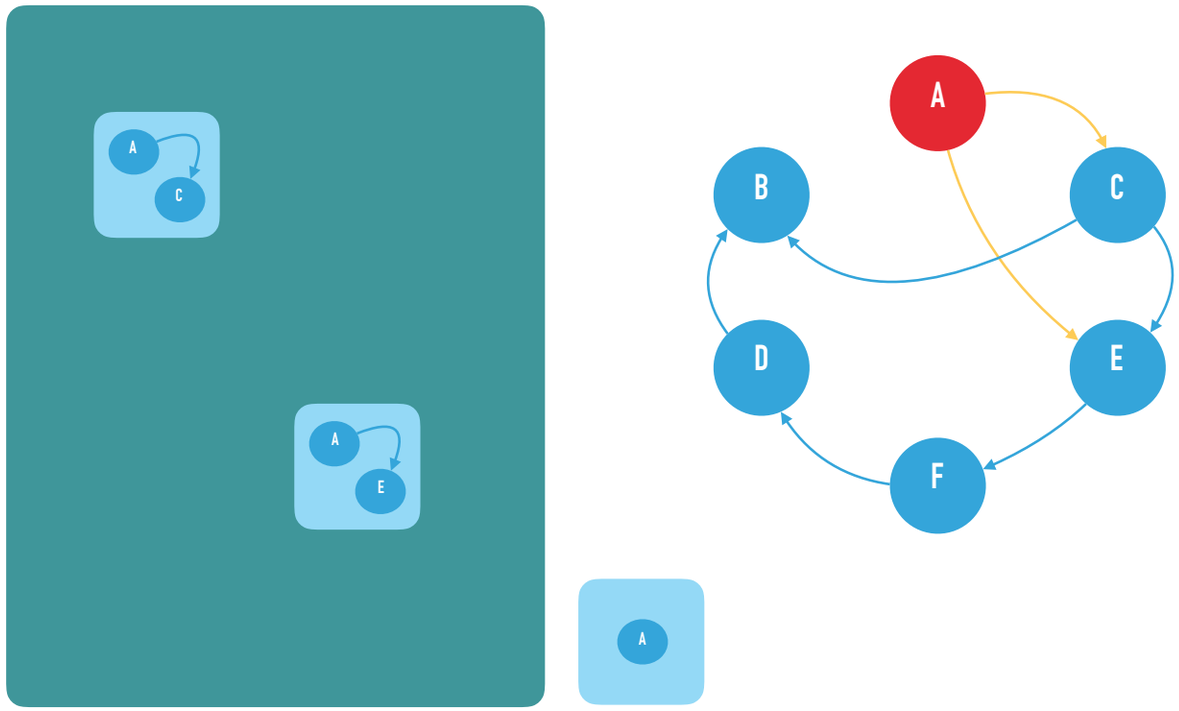
THE GRAPH SEARCH TO-DO LIST



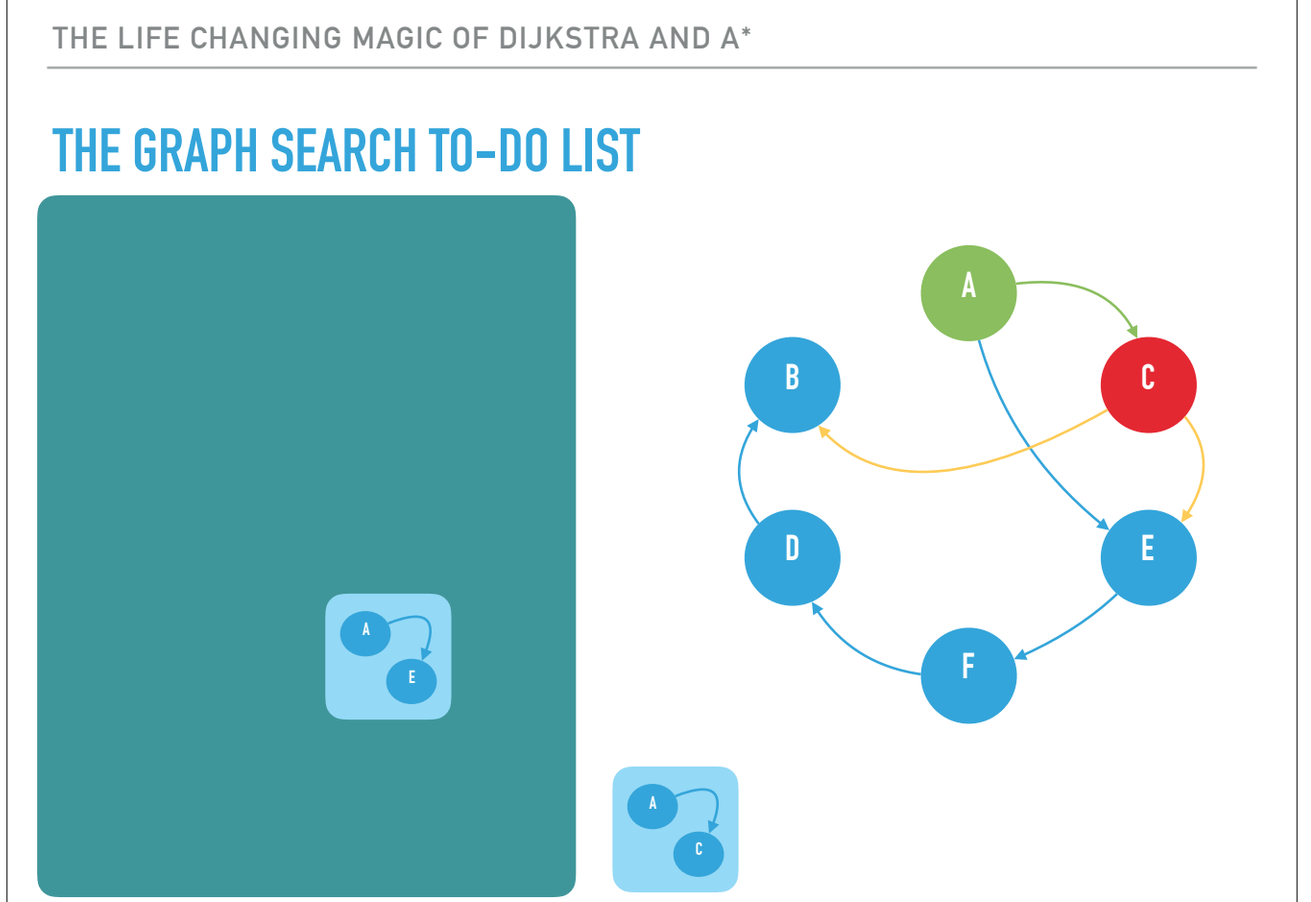


You take them out in an order determined *by the todo list*

THE GRAPH SEARCH TO-DO LIST



And then you enqueue the neighboring paths in to the todo list.



And then you repeat. The important thing here is that the order of the paths – and therefore, the algorithm you're running – is determined by the todo list.

In the case of BFS, the todo list is a queue. In the case of DFS, the todo list is a stack. But there are other data structures you can use for the todo list, without changing the code, to achieve different results.

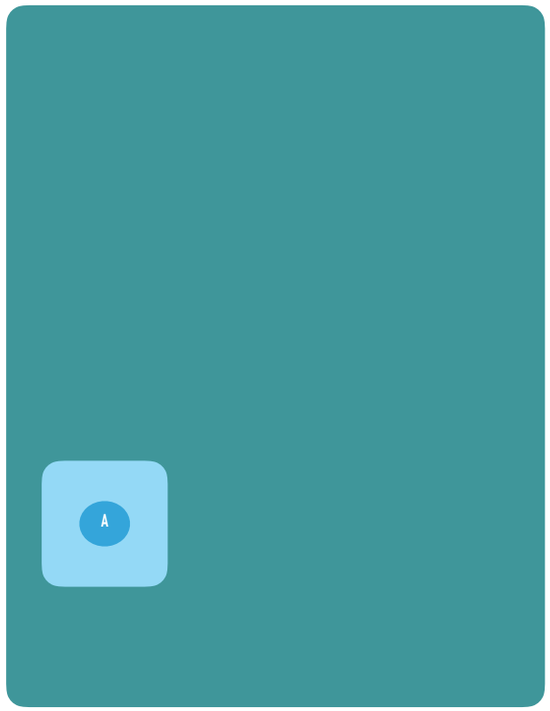
Let's put a pin in that just for just a second. We'll be back to it shortly.

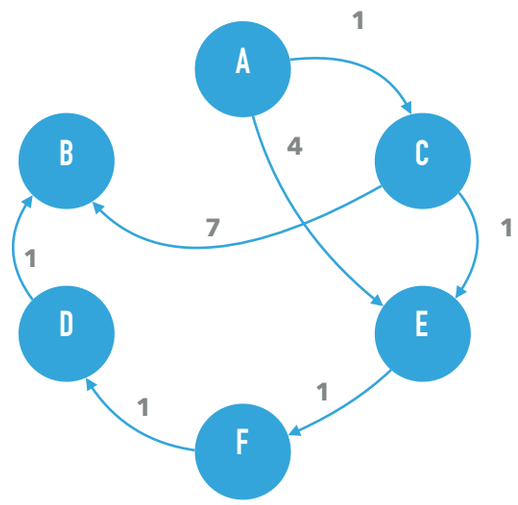
WEIGHTY DECISIONS

That's enough review for now. So let's move on to some new material. One problem with the algorithms we've looked at so far is that they don't take into account weights. And while that makes sense for a lot of graph algorithms – for example, the Wikipedia example Chris showed, or things related to your social network on Facebook – it doesn't make sense for a lot of applications.

So today we're going to look at an algorithm called Dijkstra's algorithm.

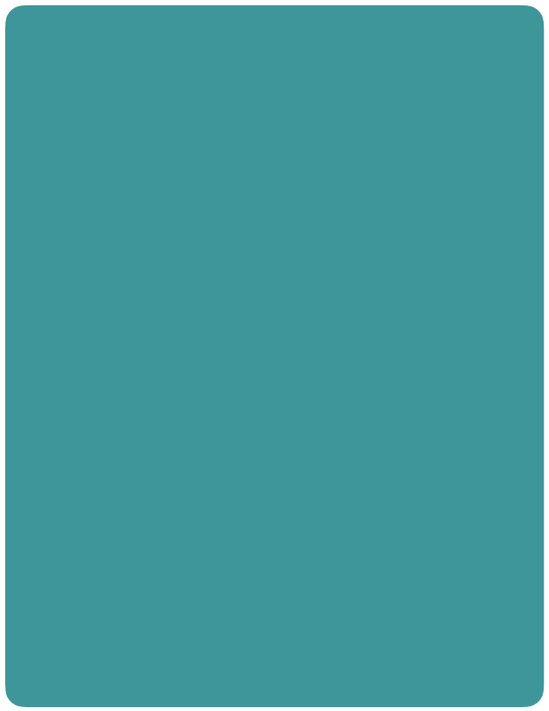
DEALING WITH WEIGHTY TOPICS

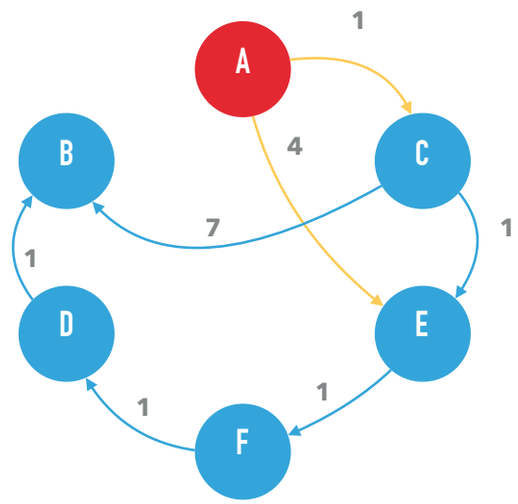




So we have here the same graph we've seen before, now with weights. And so we're going to use the todo list structure to try to create an algorithm that will find the shortest path from A to B. So using our todo list algorithm, we start by enqueueing a path with just the starting node, A.

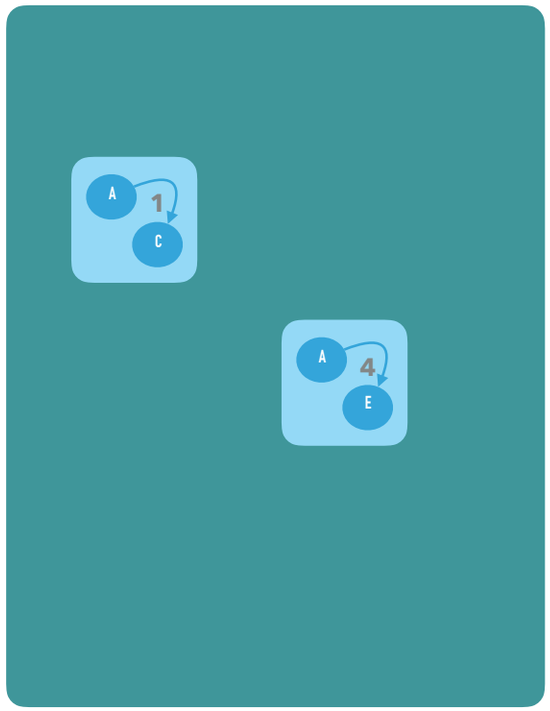
DEALING WITH WEIGHTY TOPICS

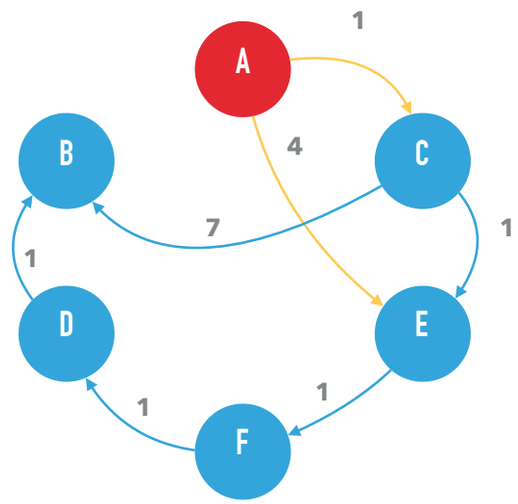




And we dequeue it

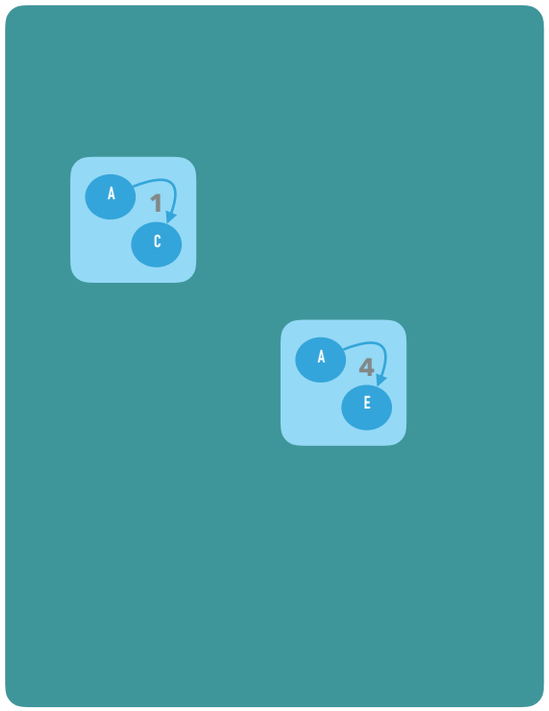
DEALING WITH WEIGHTY TOPICS

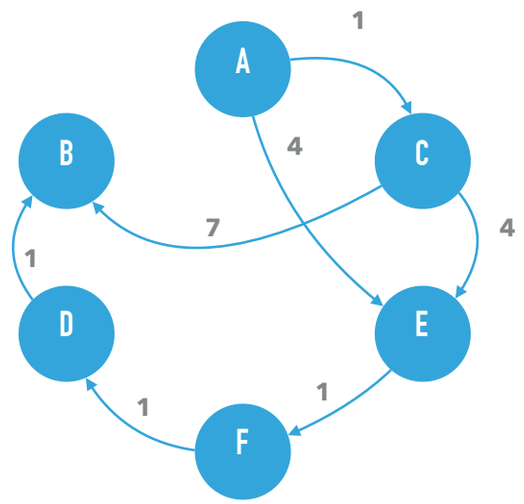




and we add the paths that end in its neighbors into the todo list.

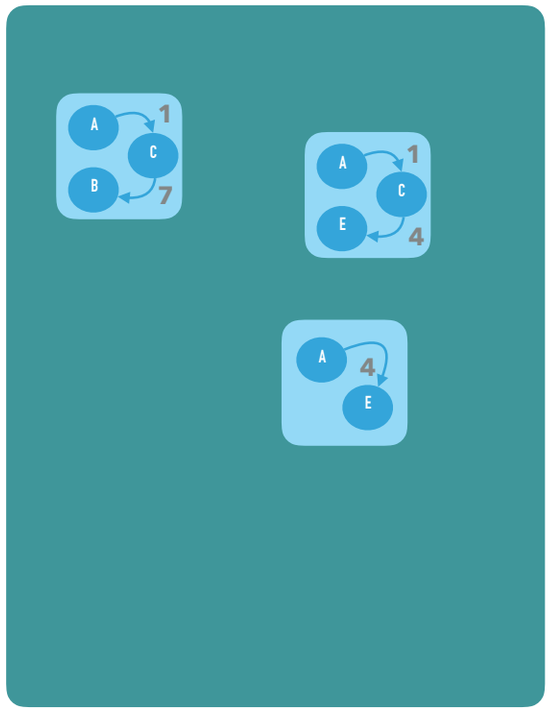
DEALING WITH WEIGHTY TOPICS

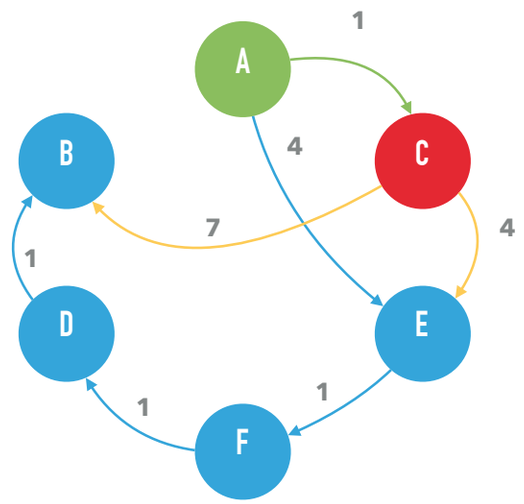




Now, we haven't figured out what structure we're going to use for our todo list, but given these two paths, which one are you going to pick?

DEALING WITH WEIGHTY TOPICS

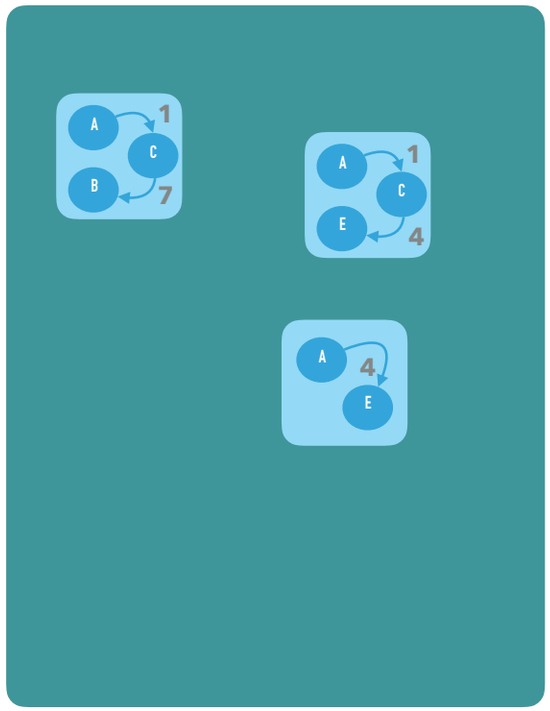


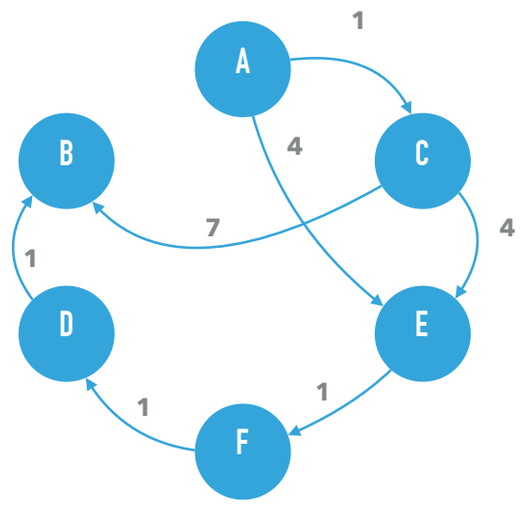


The one that makes the most sense to pick is the path from A to C. In the absence of any other information, you're going to want to pick the path that's the cheapest.

So we dequeue that one, and put in paths that have its neighbors at the end. And then we dump A to C.

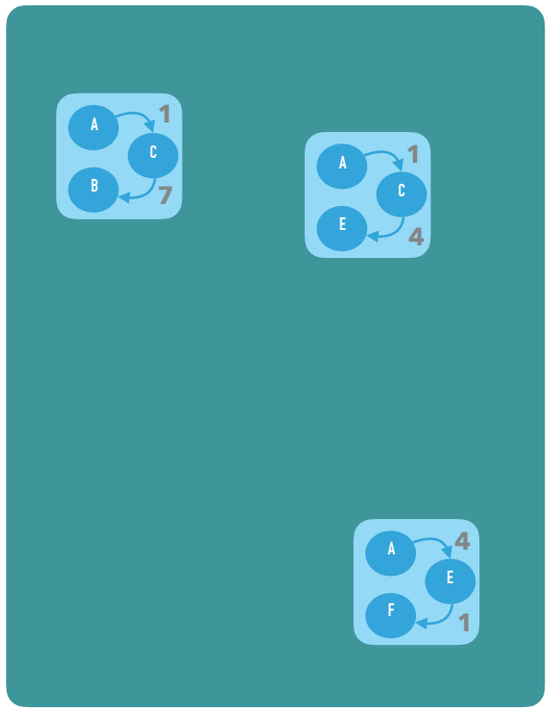
DEALING WITH WEIGHTY TOPICS

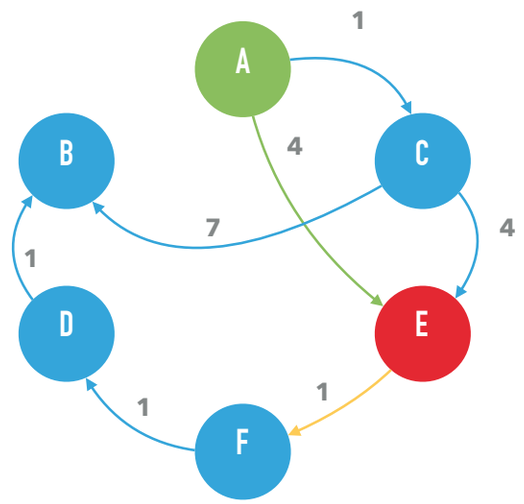




Now, given these three paths, which one are you going to pick?

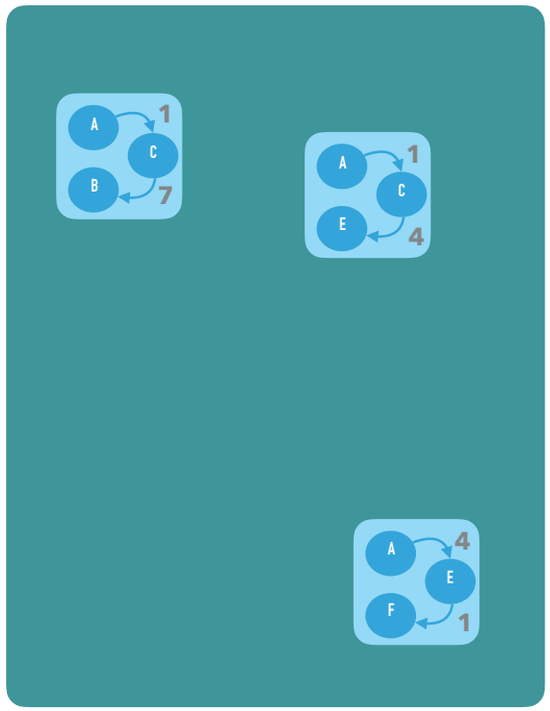
DEALING WITH WEIGHTY TOPICS

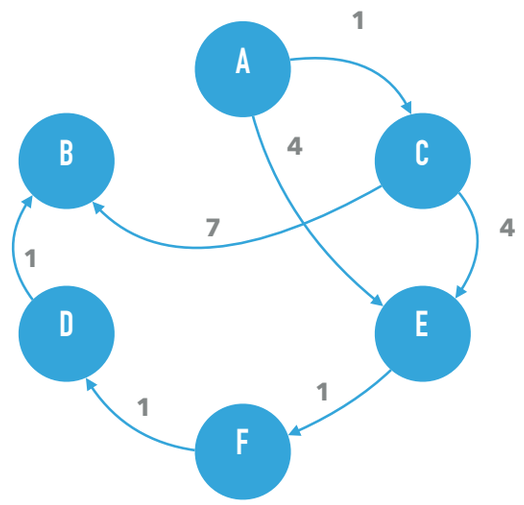




Again, you're going to pick the cheapest one (A to E) and then enqueue all the paths containing its neighbors.

DEALING WITH WEIGHTY TOPICS

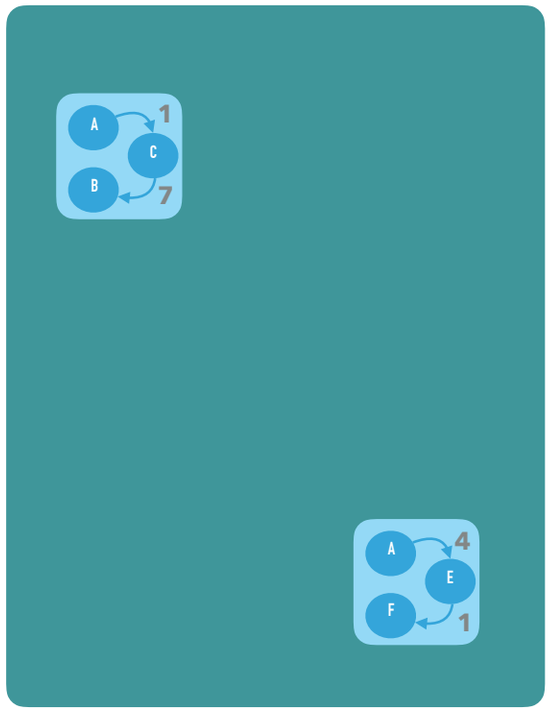


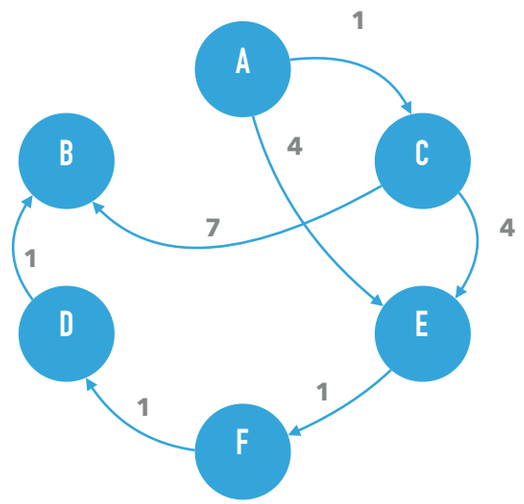


Again, which one are we going to pick in this case?

Technically there are two paths of equal weights here, so for now we'll arbitrarily pick the A to C to E path.

DEALING WITH WEIGHTY TOPICS





And then, like with all the todo list algorithms we've seen so far, we continue this until we dequeue a path that ends in our destination node.

Now, a data structure that always gives you the smallest value... what have y'all written recently that lets you do that easily?

IN DIJKSTRA'S ALGORITHM,

THE TODO LIST IS A PRIORITY QUEUE

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.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

So here's the pseudocode for Dijkstra's algorithm. And you'll note that it looks very similar to the pseudocode to breadth first search and (iterative) depth first search. There are a few things though to keep in mind.

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**

When you enqueue a path into a priority queue, you need to give it a priority. What priority should we assign a new path that we enqueue?

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**

We should enqueue it with the cost of the path

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 first path to?

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

Initializing it to a cost of 0 makes sense, since moving from a node to itself (or rather, not moving at all) doesn't cost anything.

(Technically you can initialize it to anything – it's the only thing in the queue, so it will always be dequeued first. But it's good practice to be accurate.)

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?

In the pseudocode for BFS and DFS, we said that you wait until you dequeue a path to determine whether or not you've reached your destination. That's more of a formality than an actual requirement.

With Dijkstra's, though, it's important that you don't check for the end position until you've dequeued the path. Take a moment to think about why that might be.

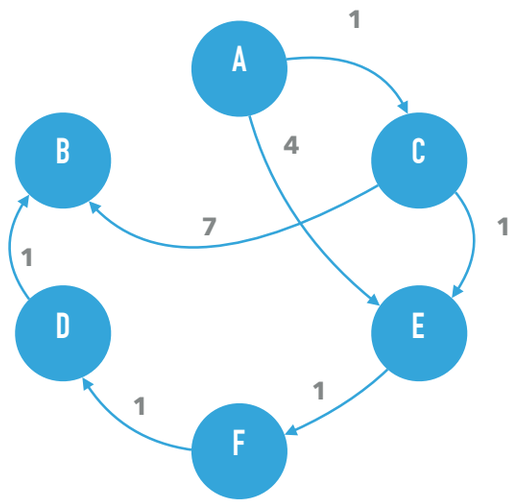
DIJKSTRA'S ODDS AND ENDS

- 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.

There might be another path to that node that's cheaper. For example, going back to the example we had earlier

DIJKSTRA'S ODDS AND ENDS



Due to the number of nodes in the path, you're going to enqueue A to C to B before you enqueue A to C to E (and so on). That means if you check to see if you've finished before you enqueue, you're going to pick a path of weight 8, instead of the cheaper path (with more nodes).

This is the most common mistake people make on Trailblazer! Make sure you are careful and avoid it!

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?

Finally, why would you want to skip a node that you've seen before?

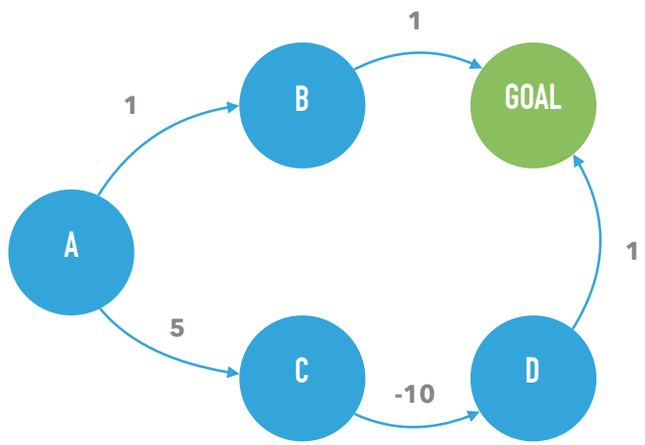
DIJKSTRA'S ODDS AND ENDS

- 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**

Because the first time you find a node, it's the cheapest path to that node. There's no point in searching from that point again.

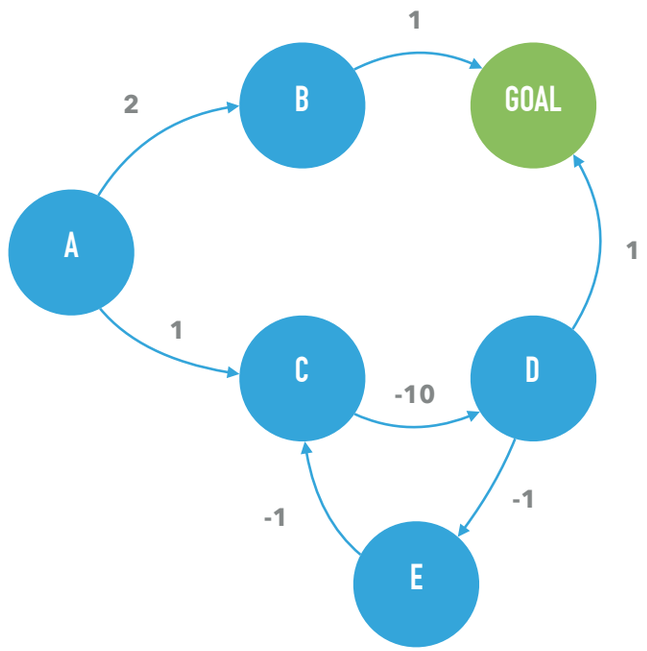
NEGATIVE EDGES



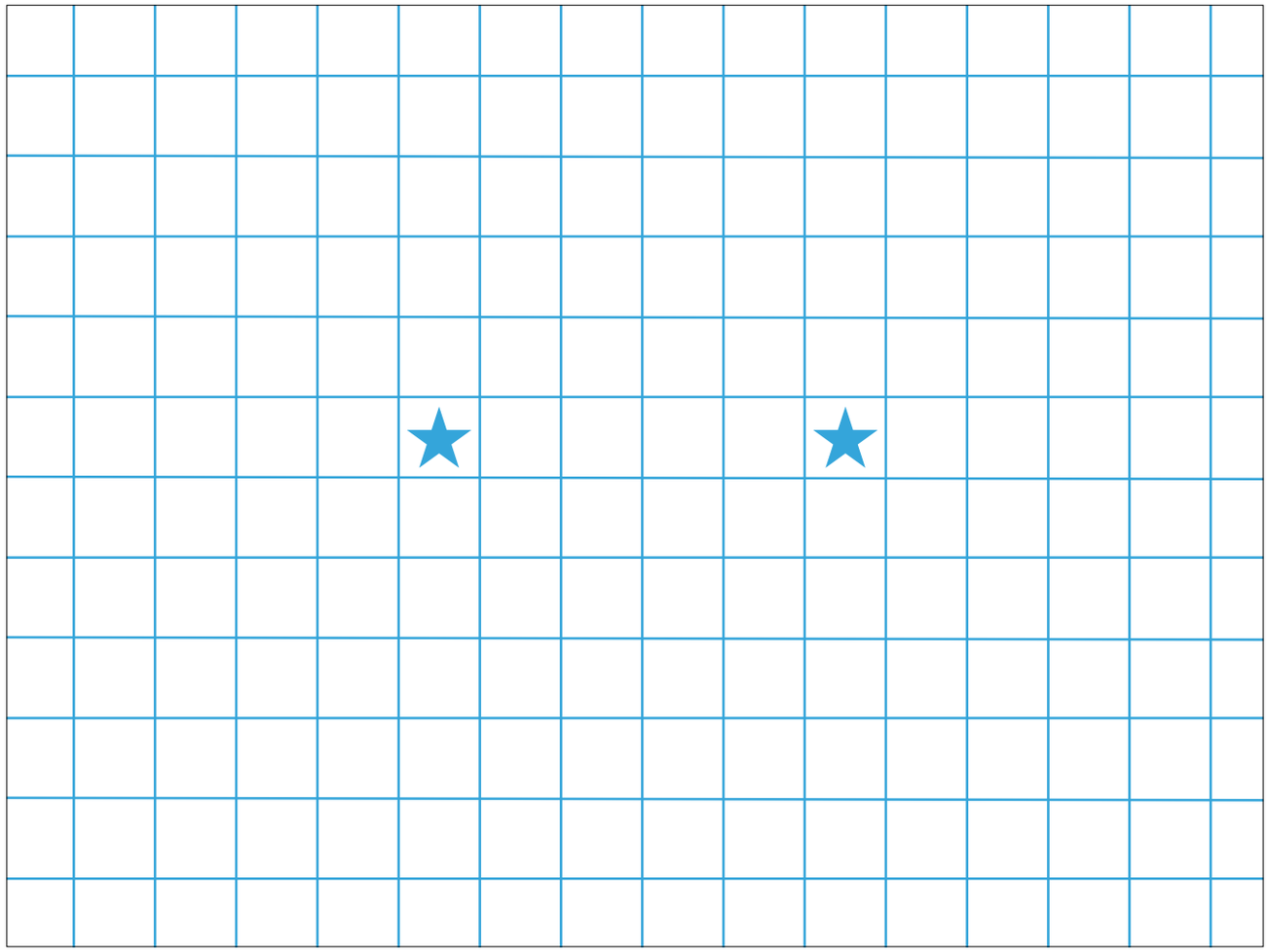
One other important thing to know is that Dijkstra's doesn't work well in graphs with negative edges. In this graph, Dijkstra's is going to prioritize the weight of the edge between A and B, and then A and B and Goal over the edge with weight 5. But ultimately, with the negative weight, that second path would be the better one.

In practice, there aren't too many graphs with negative weights – you're never going to go somewhere where you gain time back by going there – but it's important to keep in mind.

NEGATIVE CYCLES

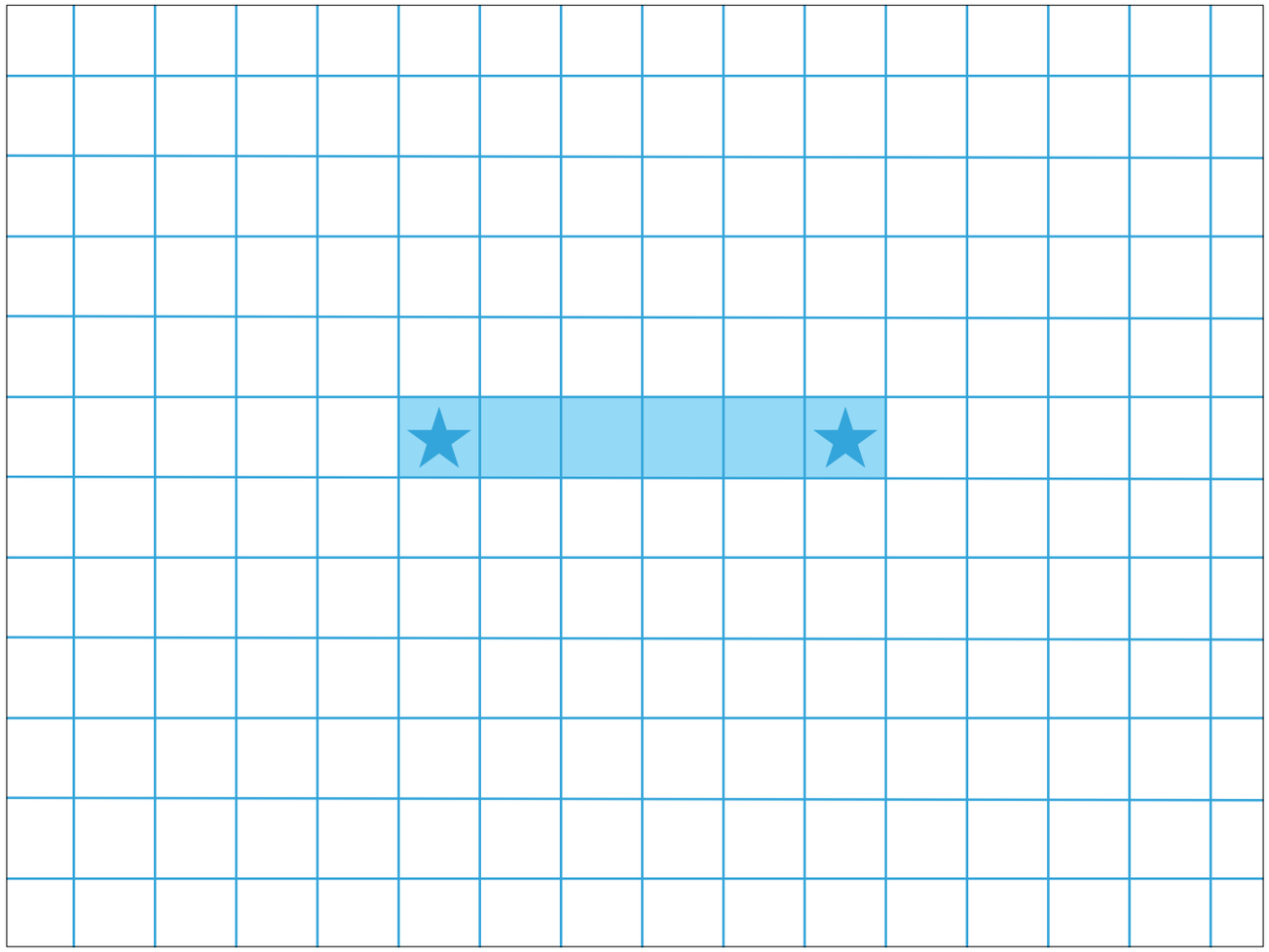


Dijkstra's also has problems when there are negative cycles - given free rein, it'll go around the cycle again and again and again.



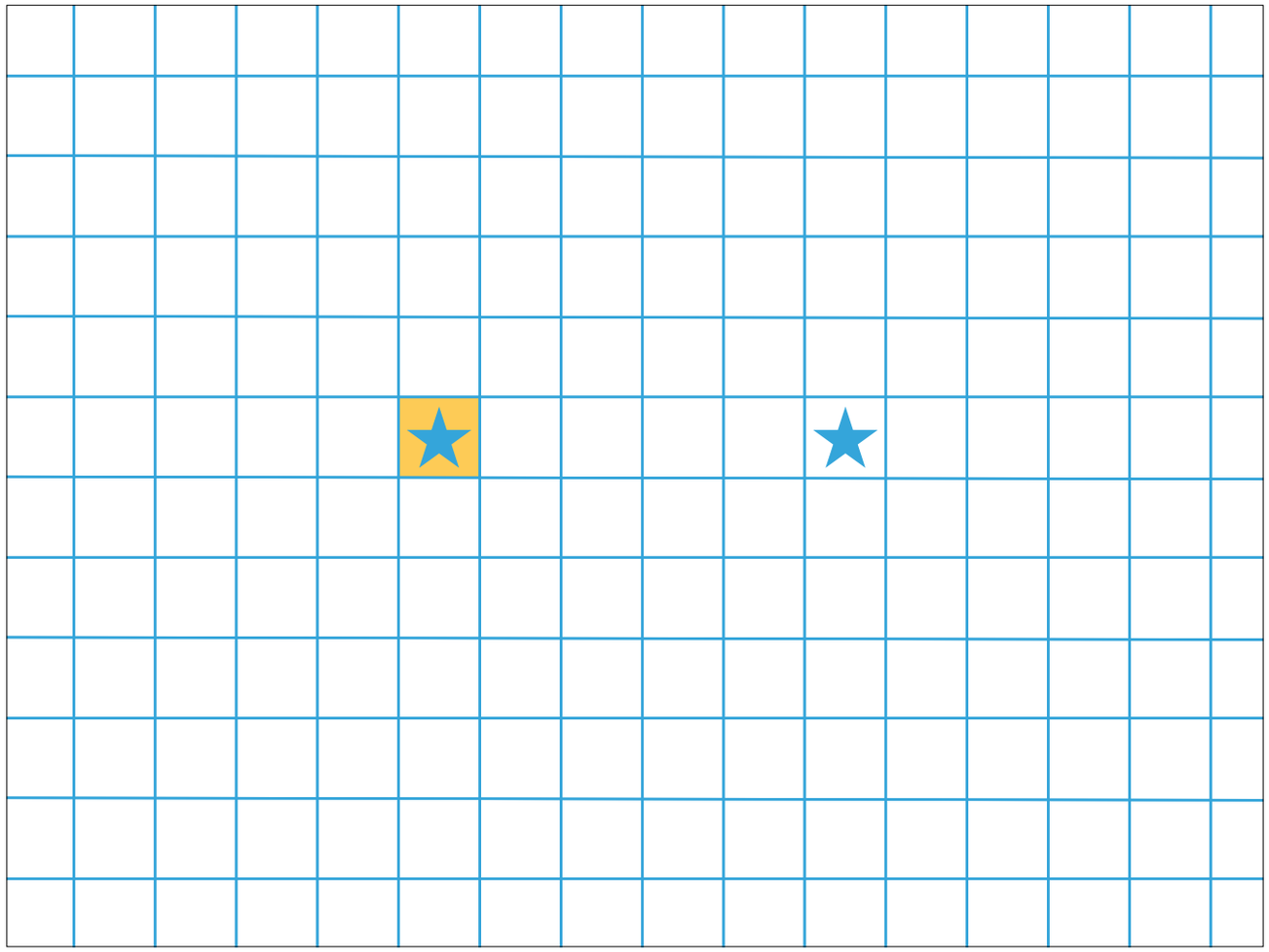
There are some other graphs where Dijkstra's doesn't perform very well. In particular, it doesn't work very well on graphs where all the edges have equal weights. Take for instance this grid; each cell is a vertex, and each cell has four outgoing edges, to each of the cells north, south, west, and east. We'll assume for now they each have weights as 1.

What's the shortest path between the two stars on this graph?

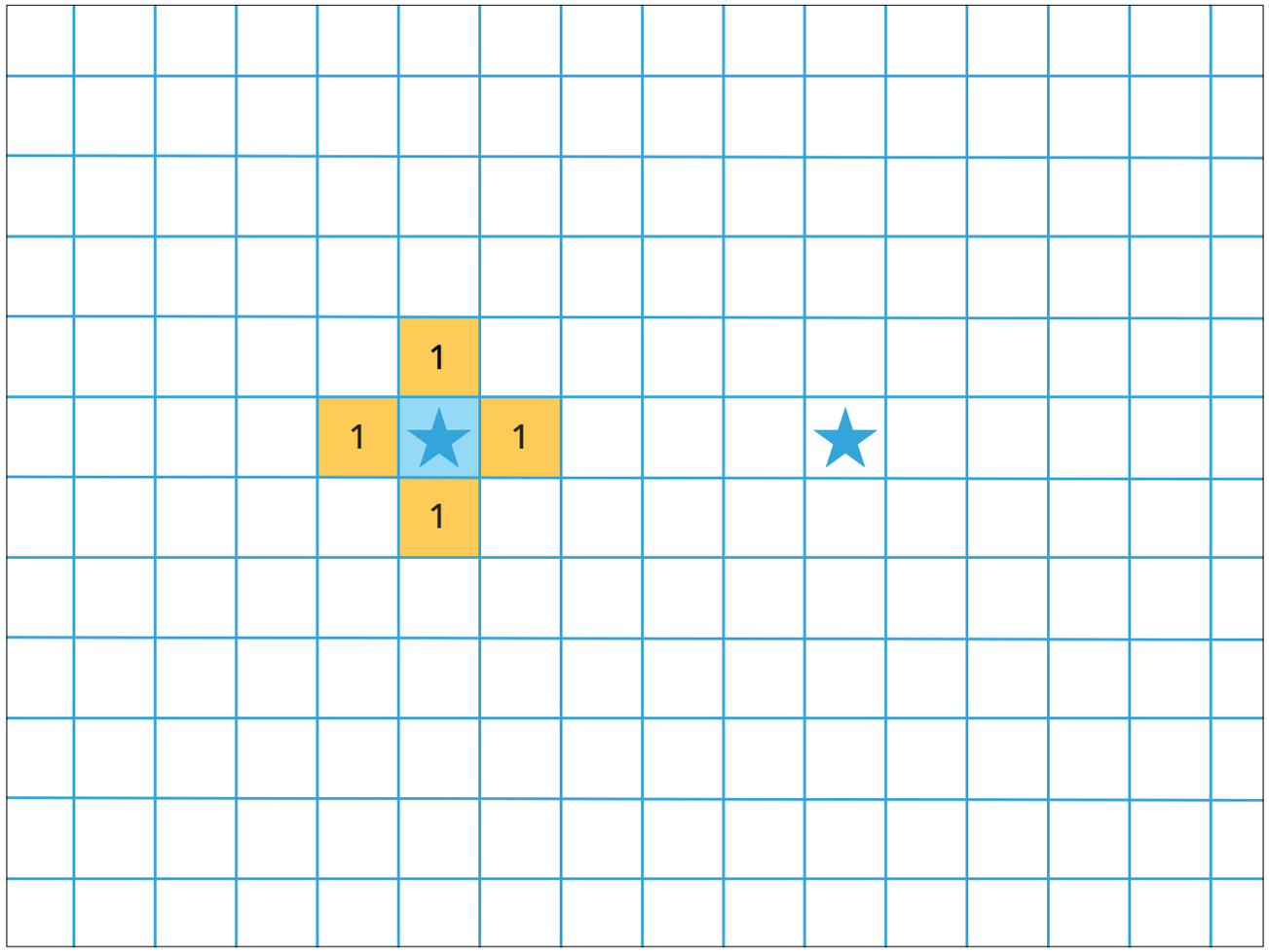


It's pretty straightforward! Let's see how Djikstra's does trying to find this path.

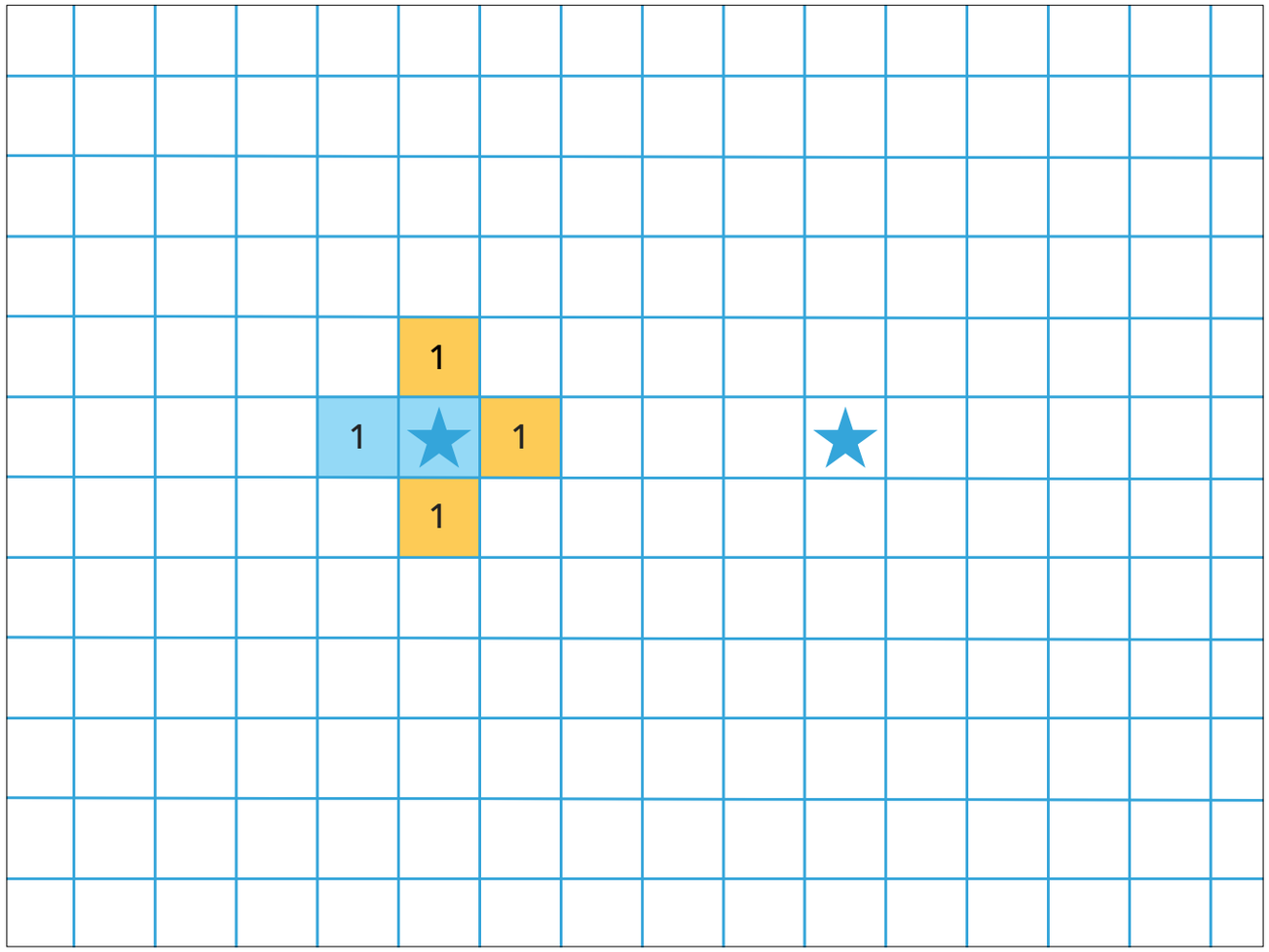
(For consistency, we'll process nodes with the same weights from left to right, top to bottom)



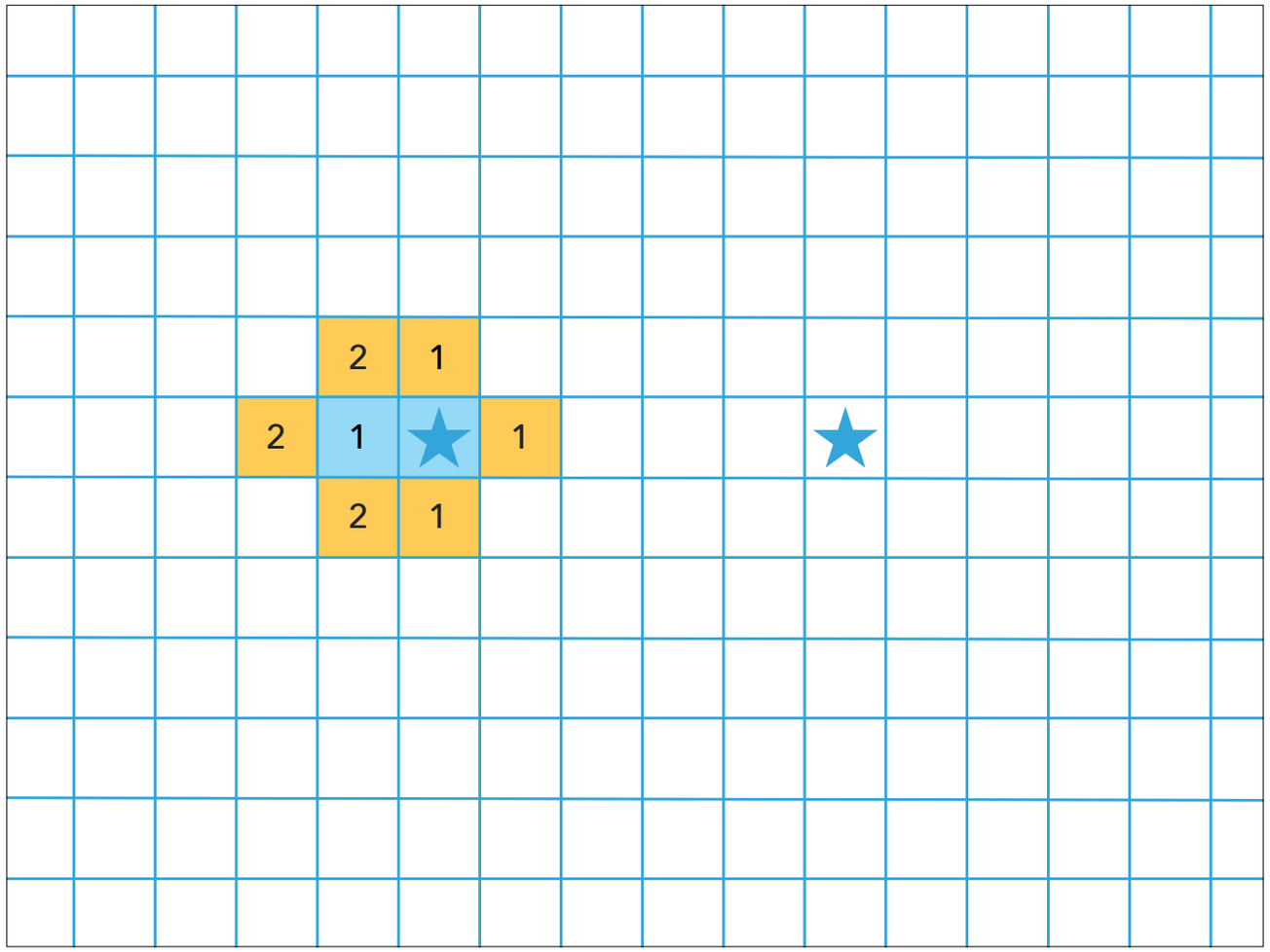
So we start by enqueueing the starting node



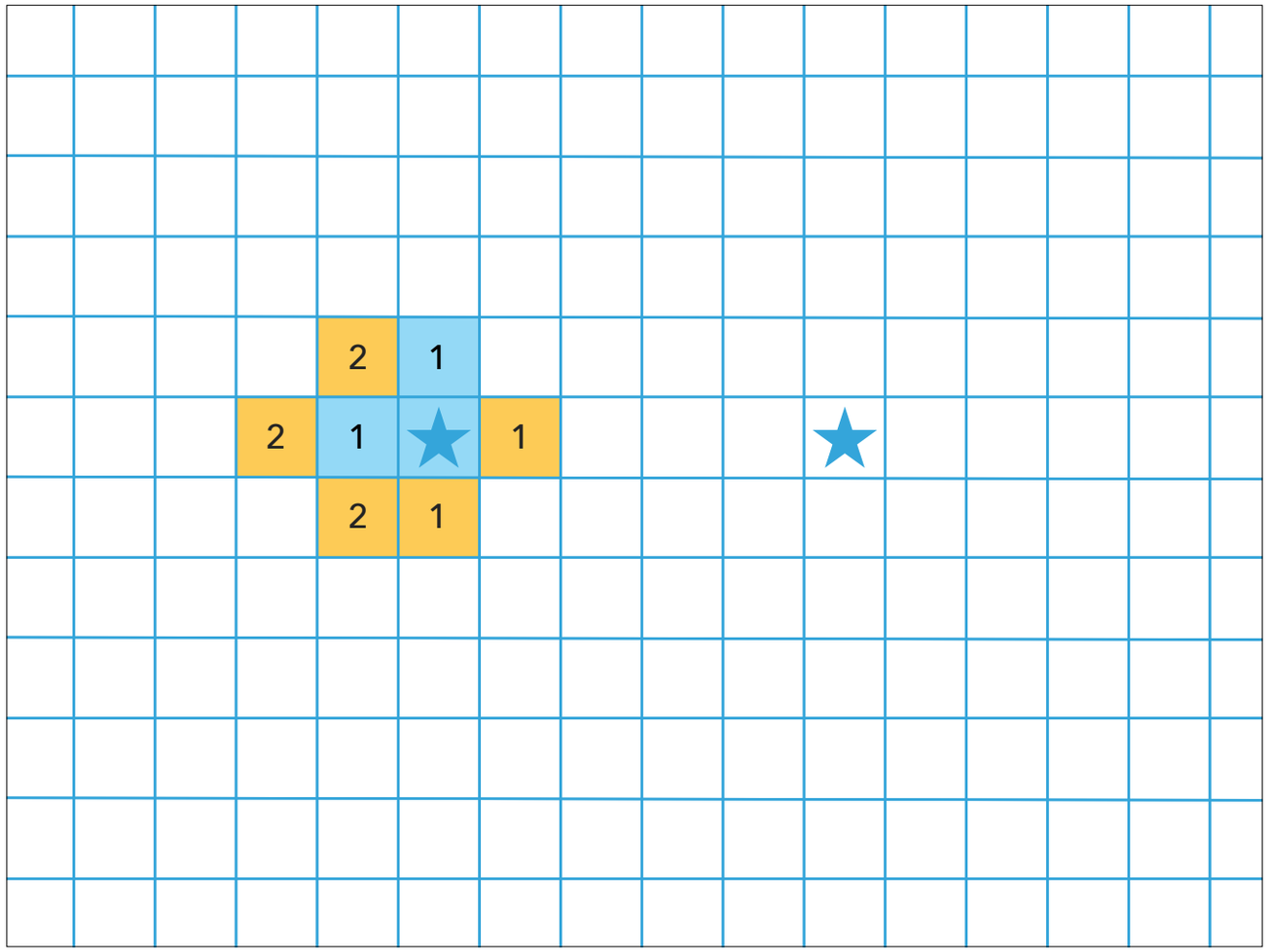
And then we dequeue it and enqueue its neighbors all with path weights of 1



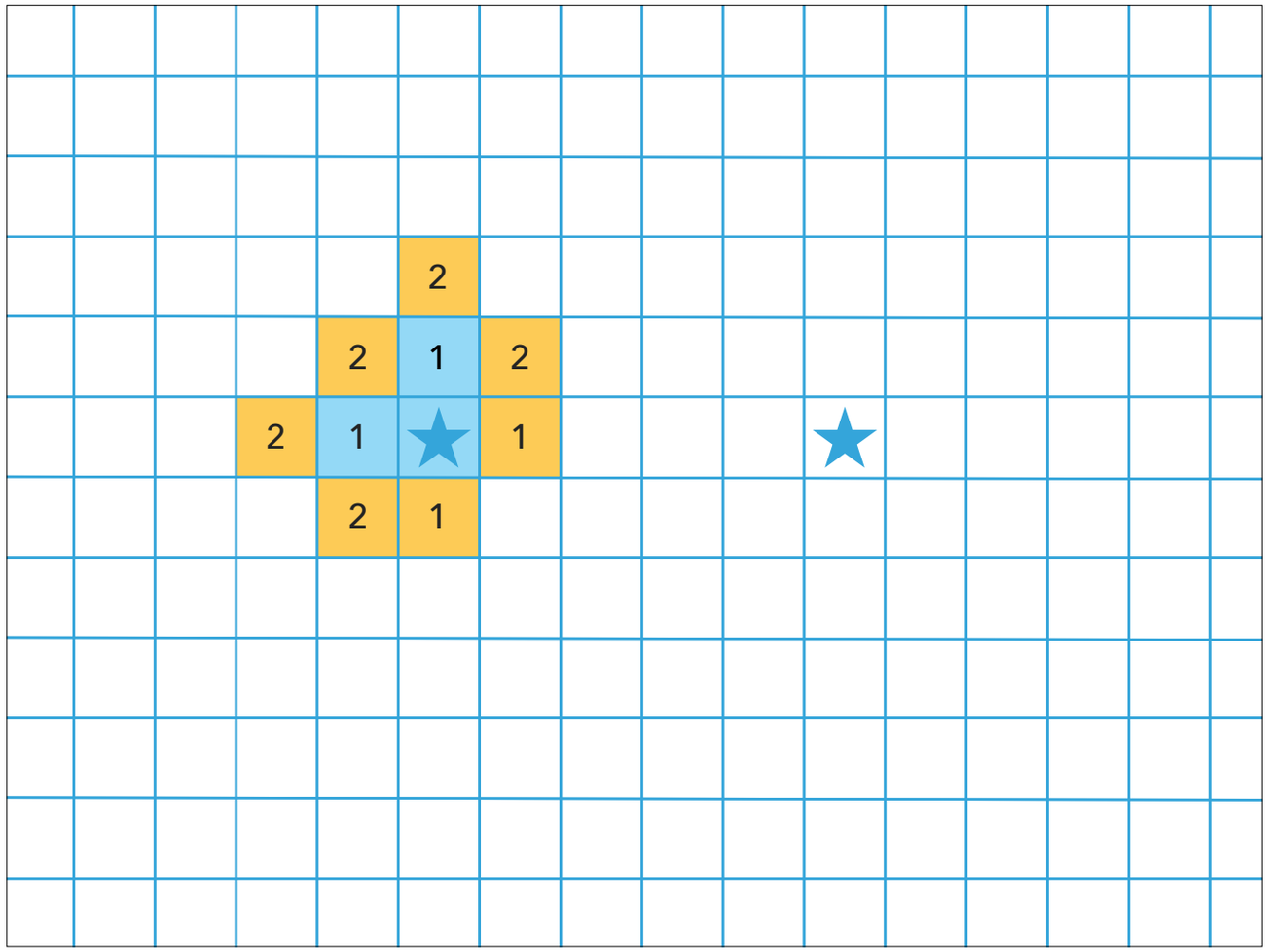
We dequeue the first path with the lowest weight



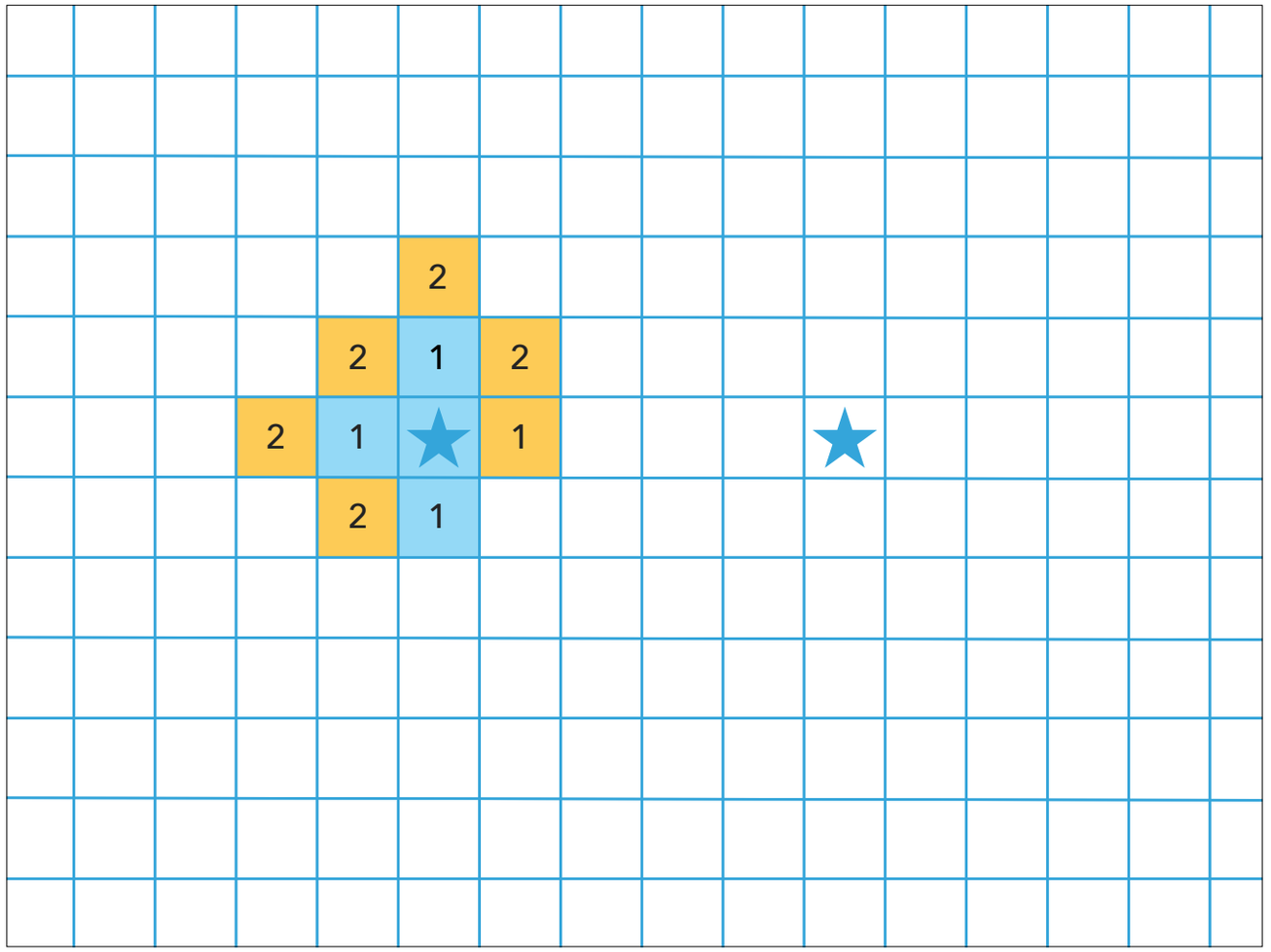
And we enqueue its neighbors as paths of length 2



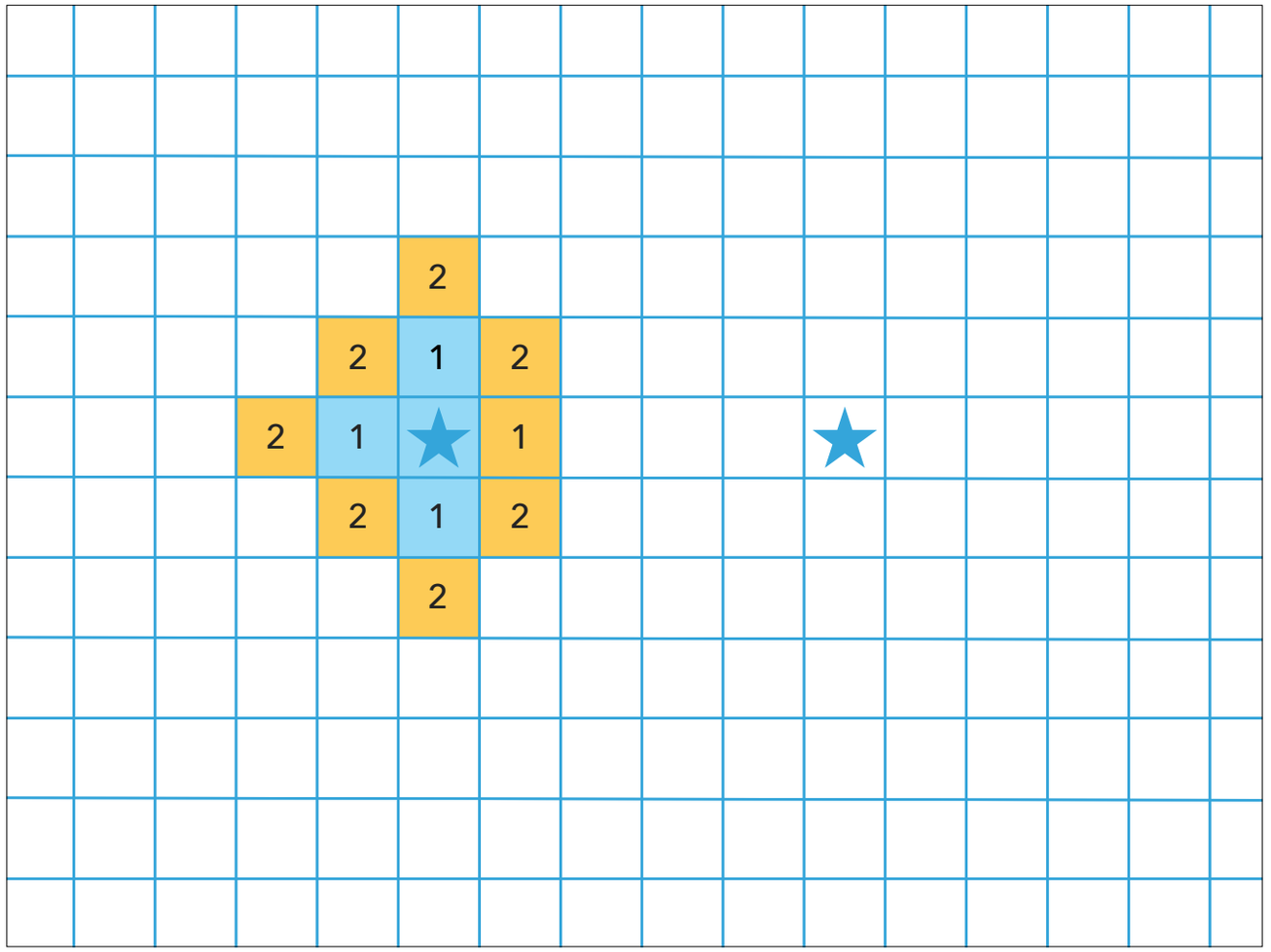
We dequeue the next cheapest path



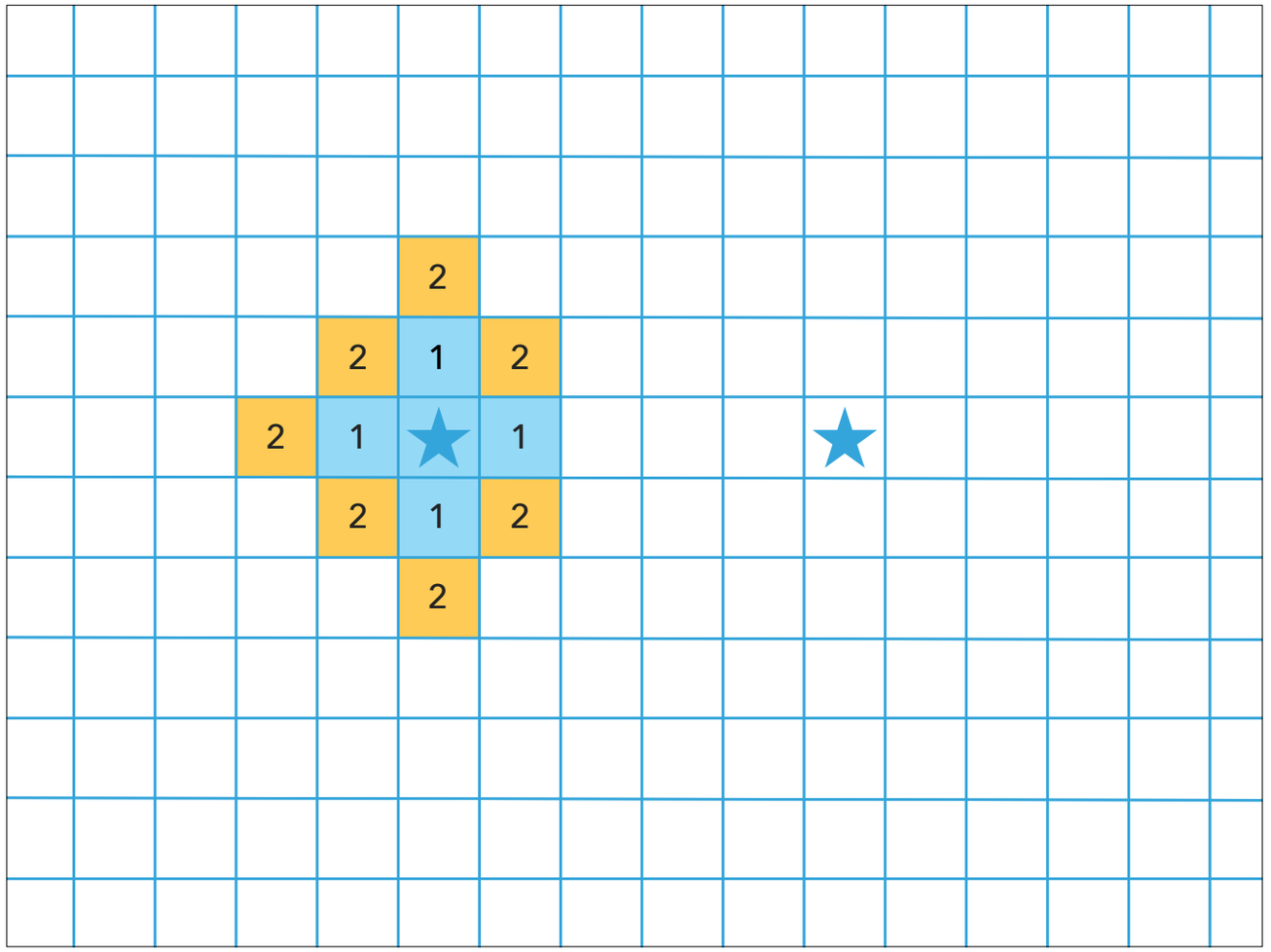
And then we enqueue its neighbor paths with weights of 2



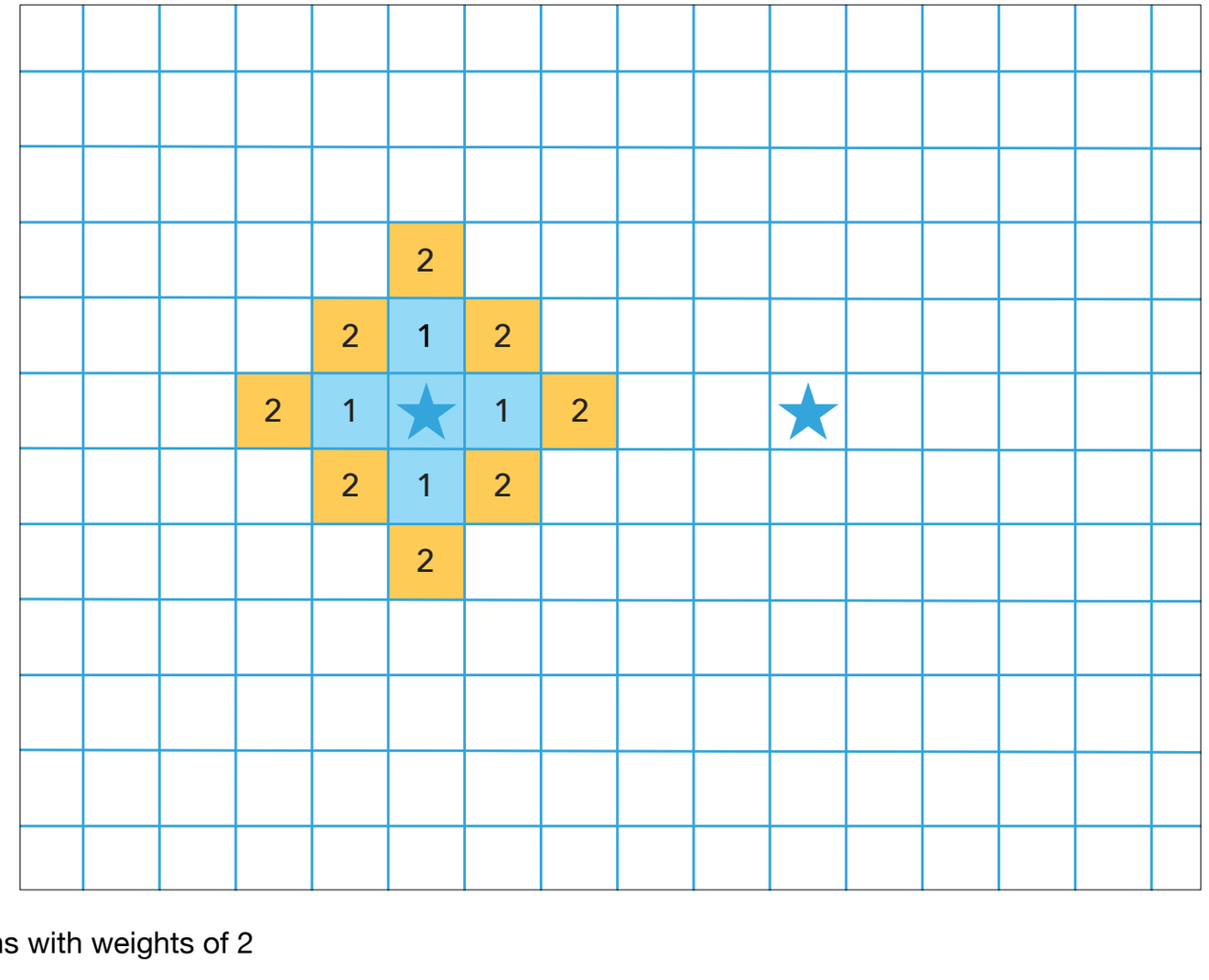
We dequeue the next one



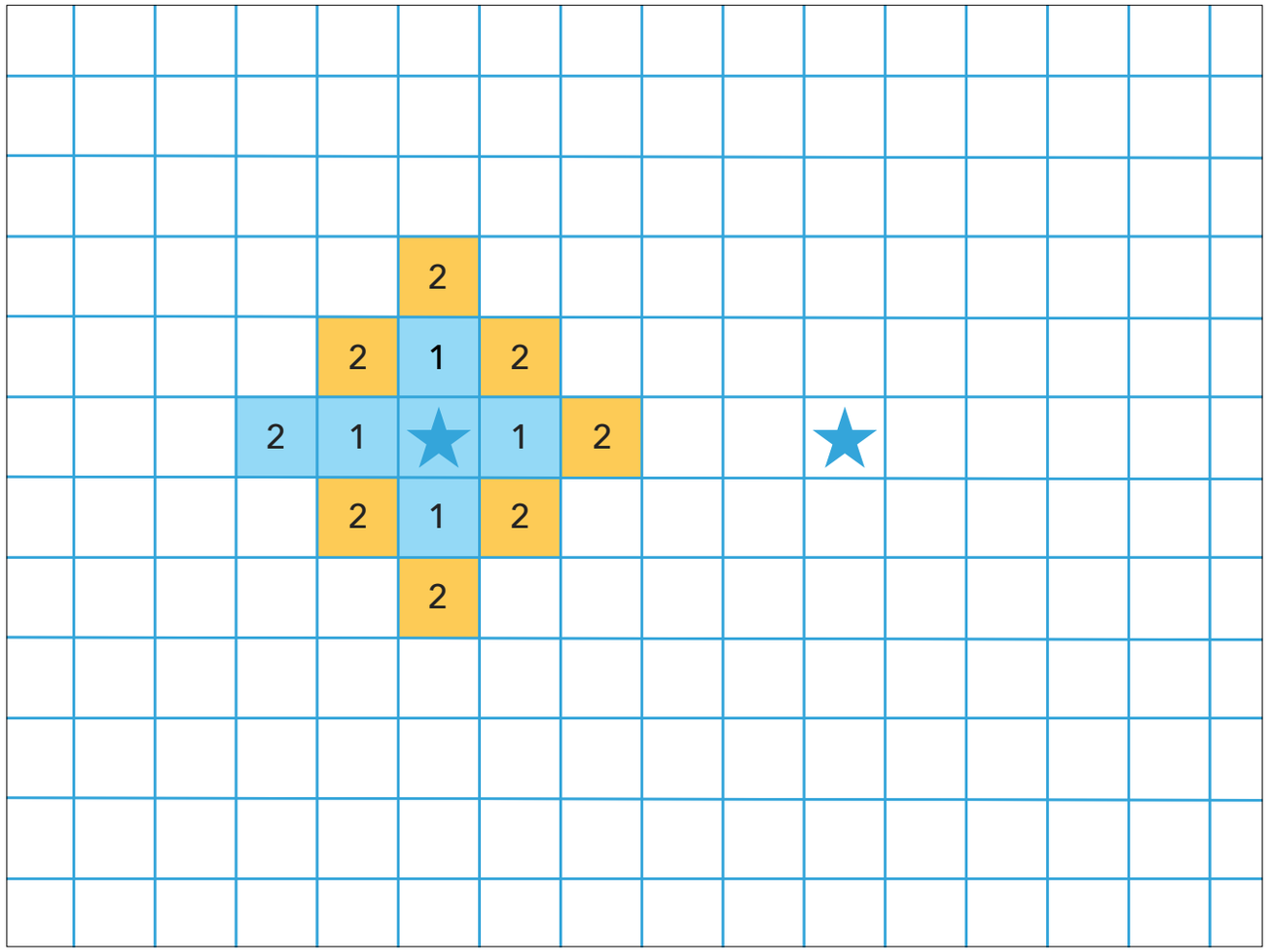
And then we enqueue its neighbor paths with weights of 2



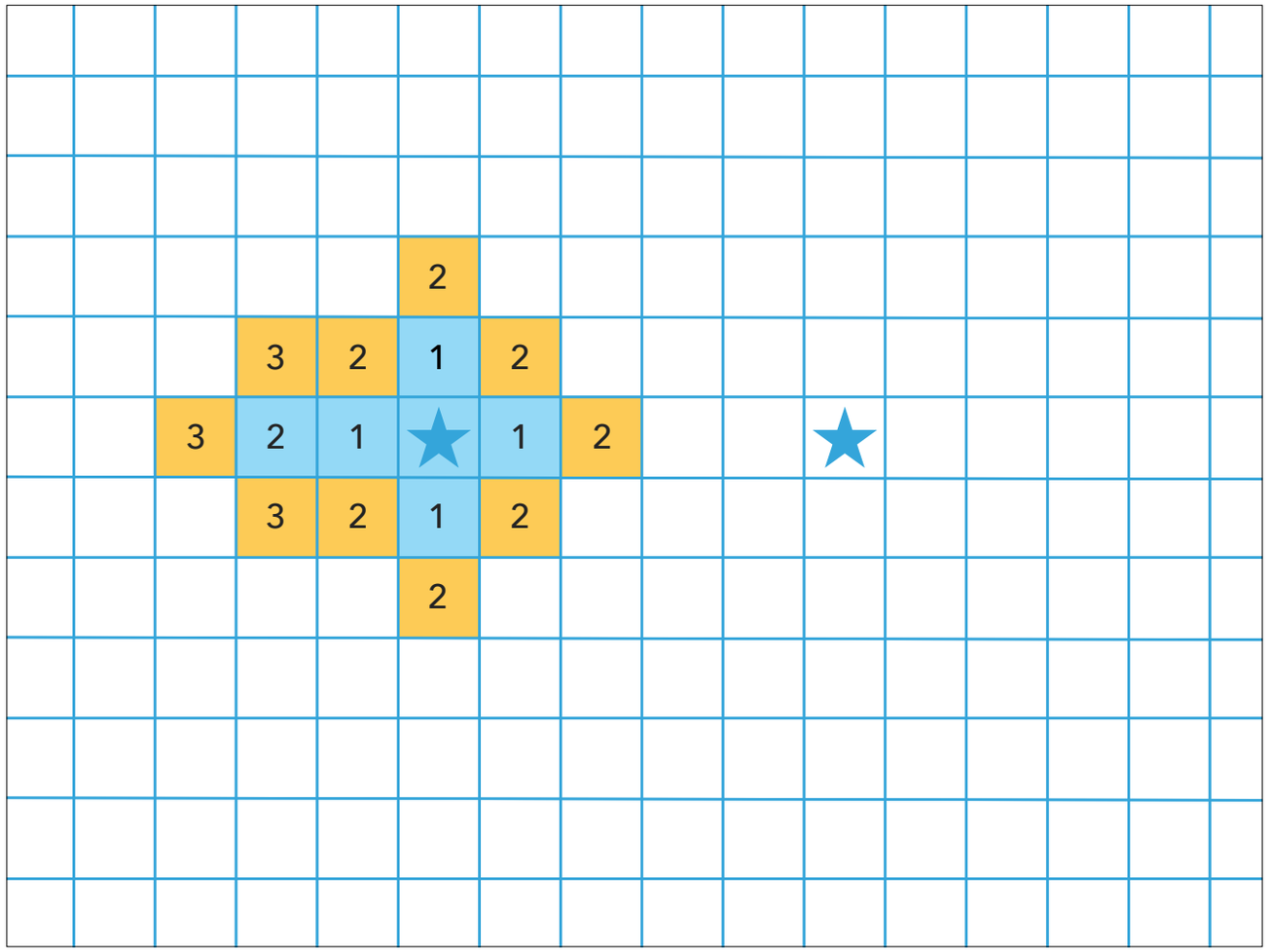
And we dequeue the last path with a weight of 1



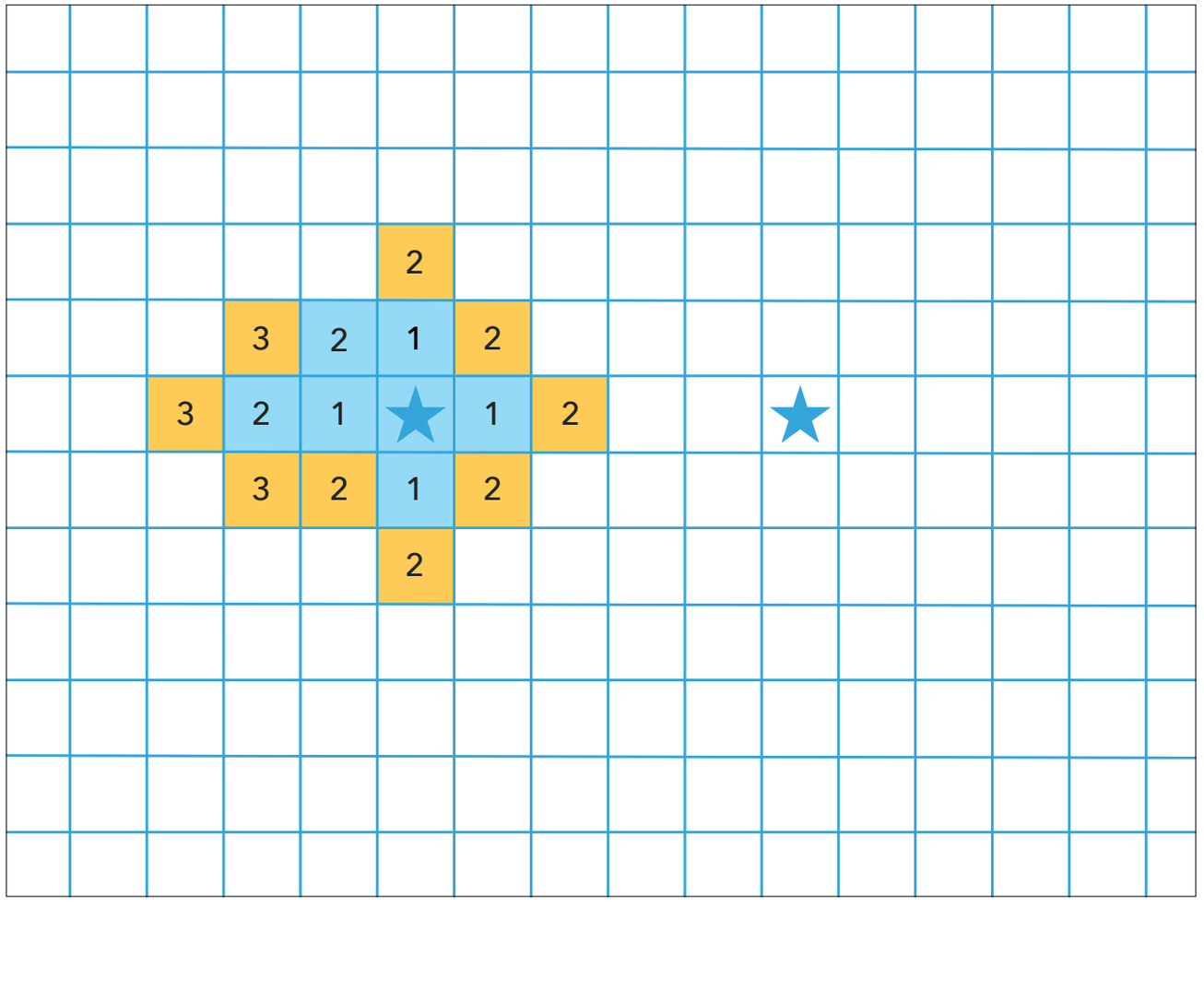
And then we enqueue its neighbor paths with weights of 2



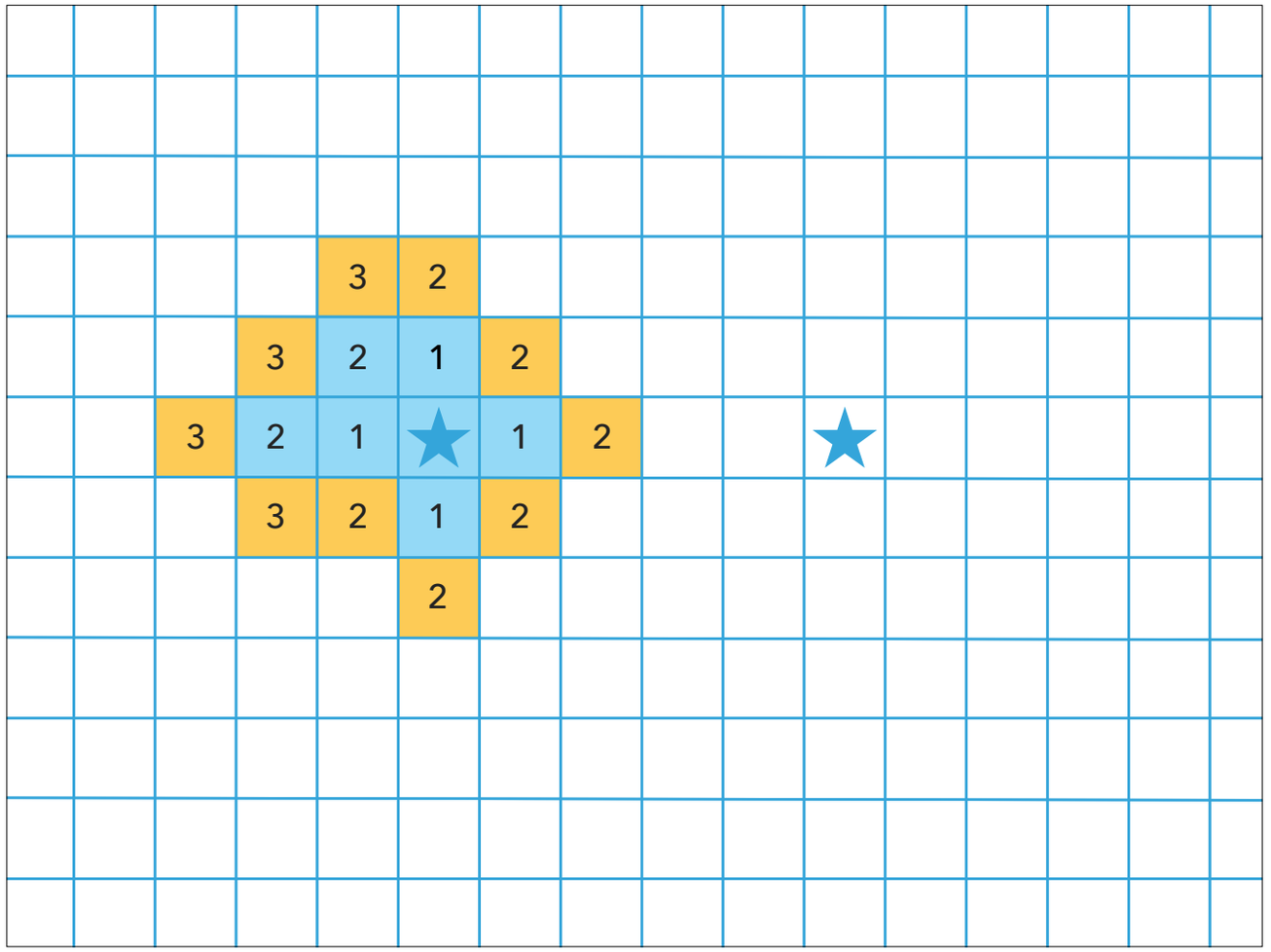
Now the cheapest path has a weight of 2; so we dequeue it



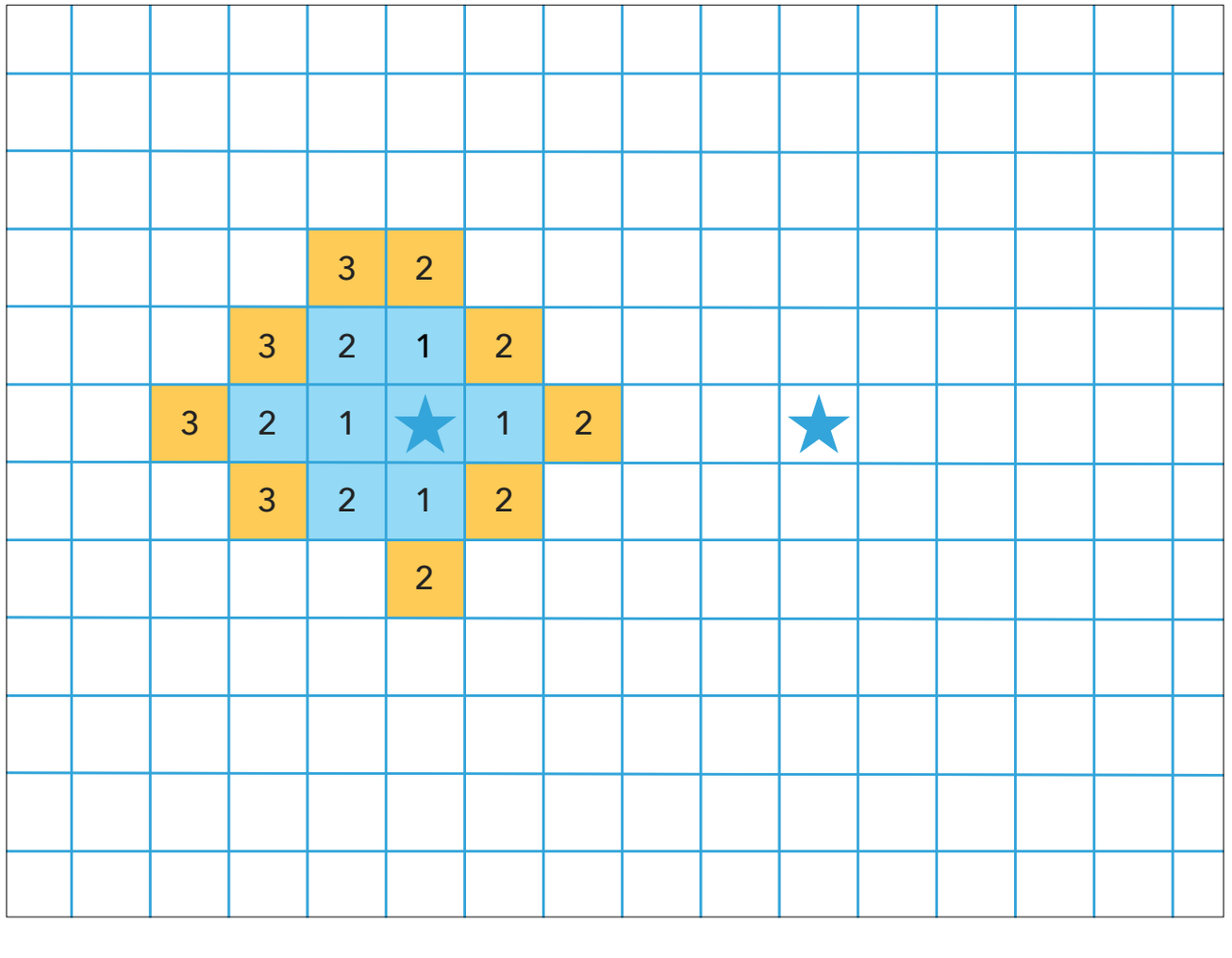
And we enqueue the paths to its neighbors with weights of 3



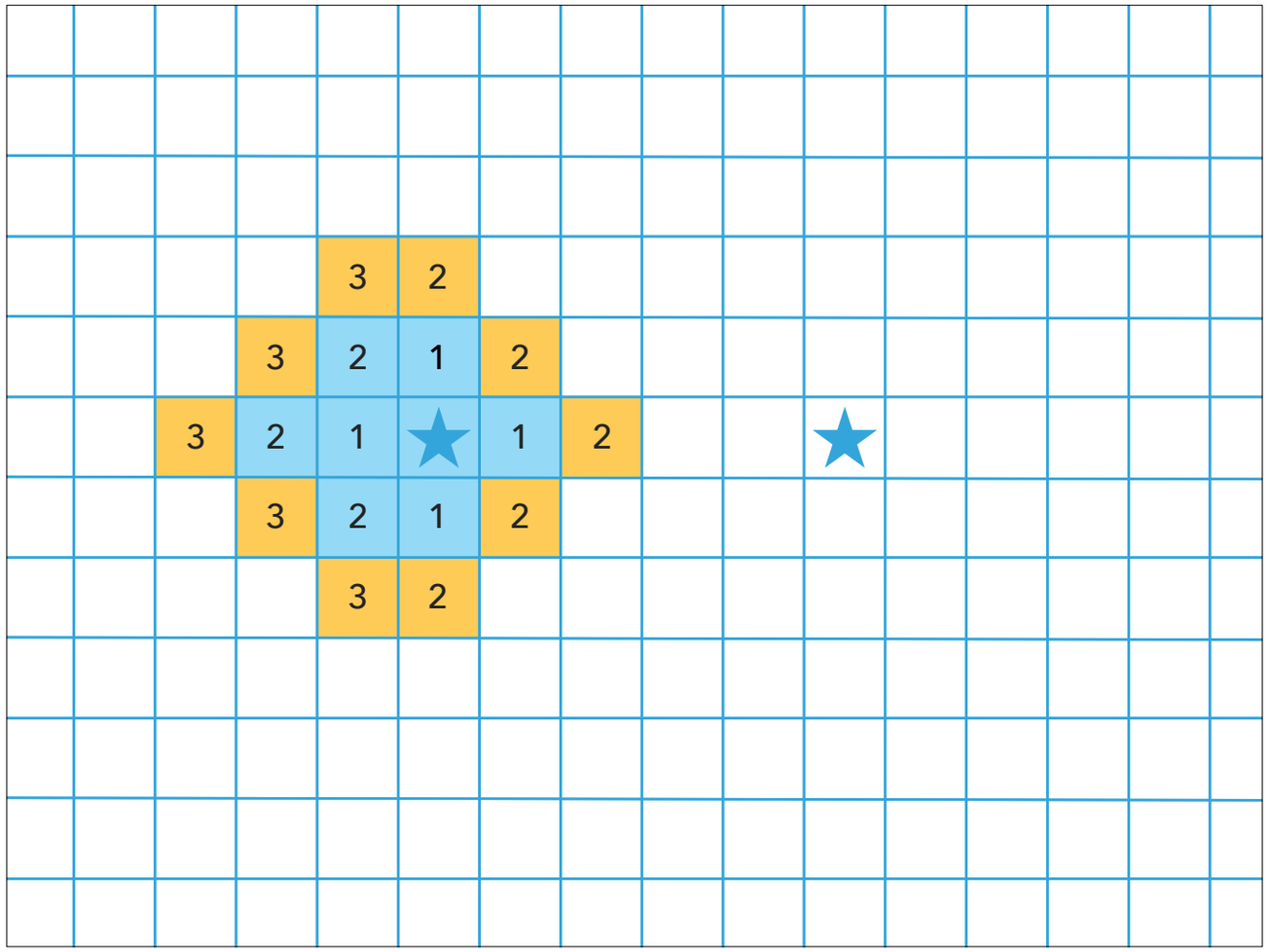
Dequeue the next one



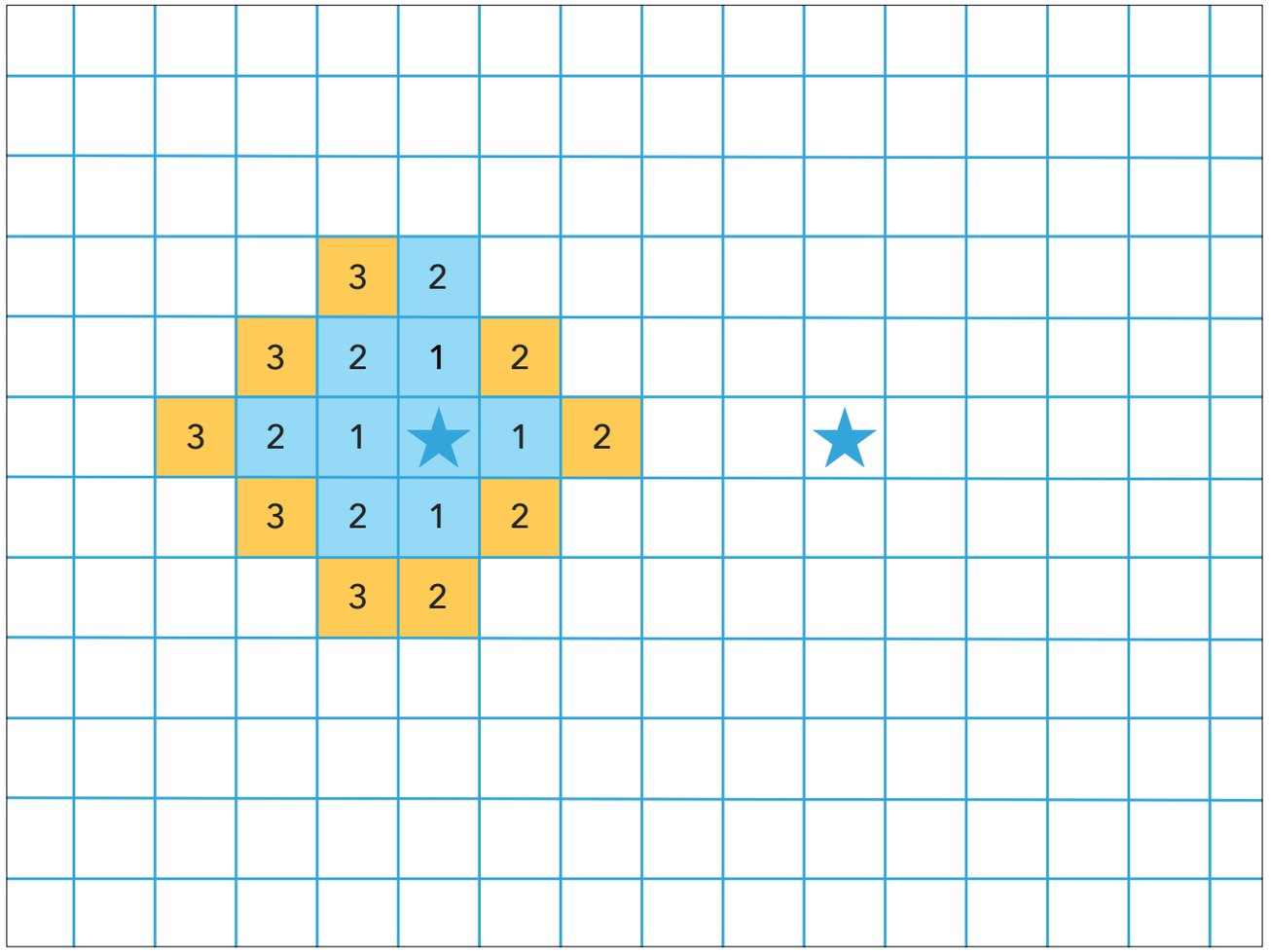
And we enqueue the paths to its neighbors with weights of 3



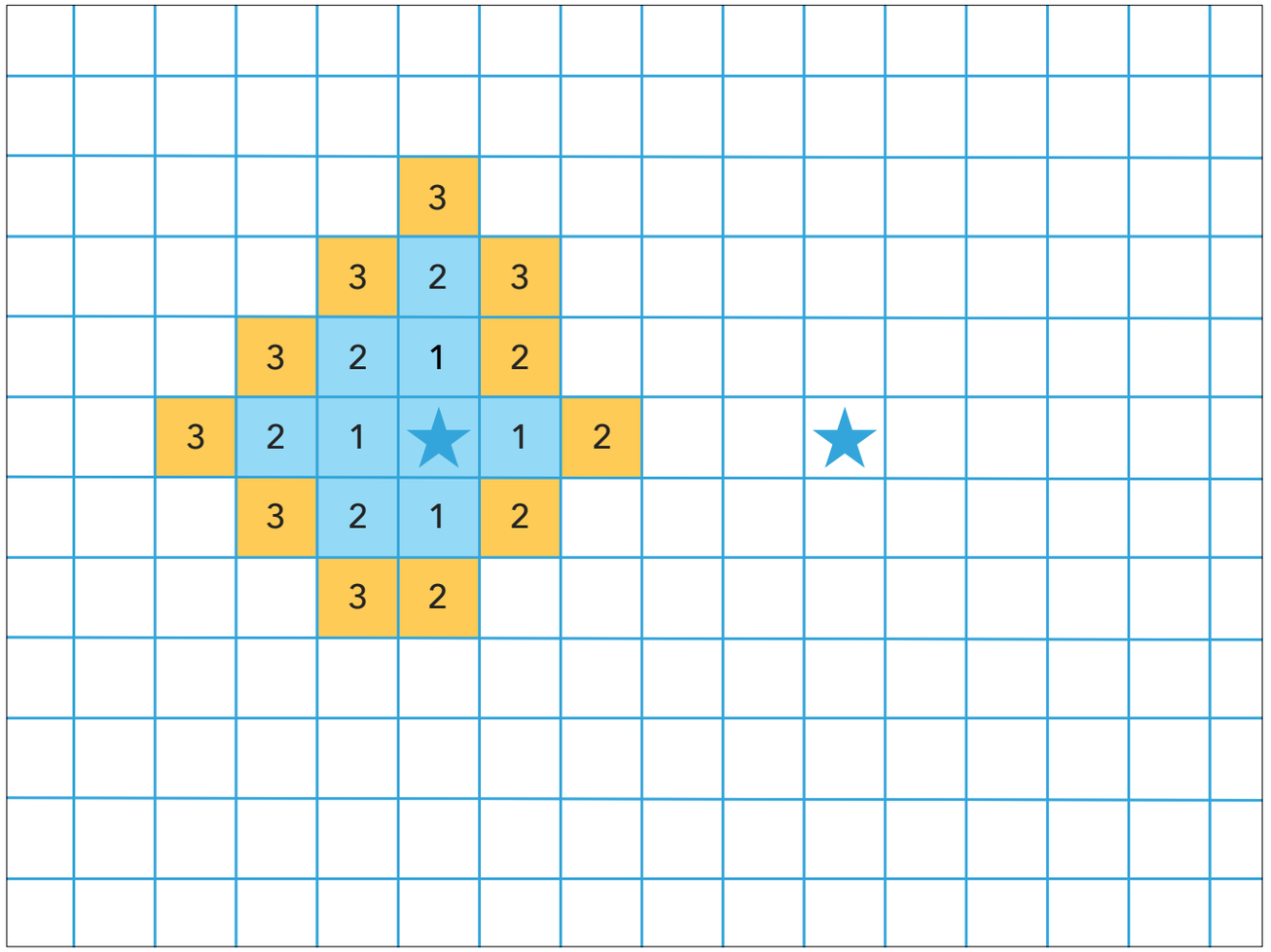
Dequeue the next one



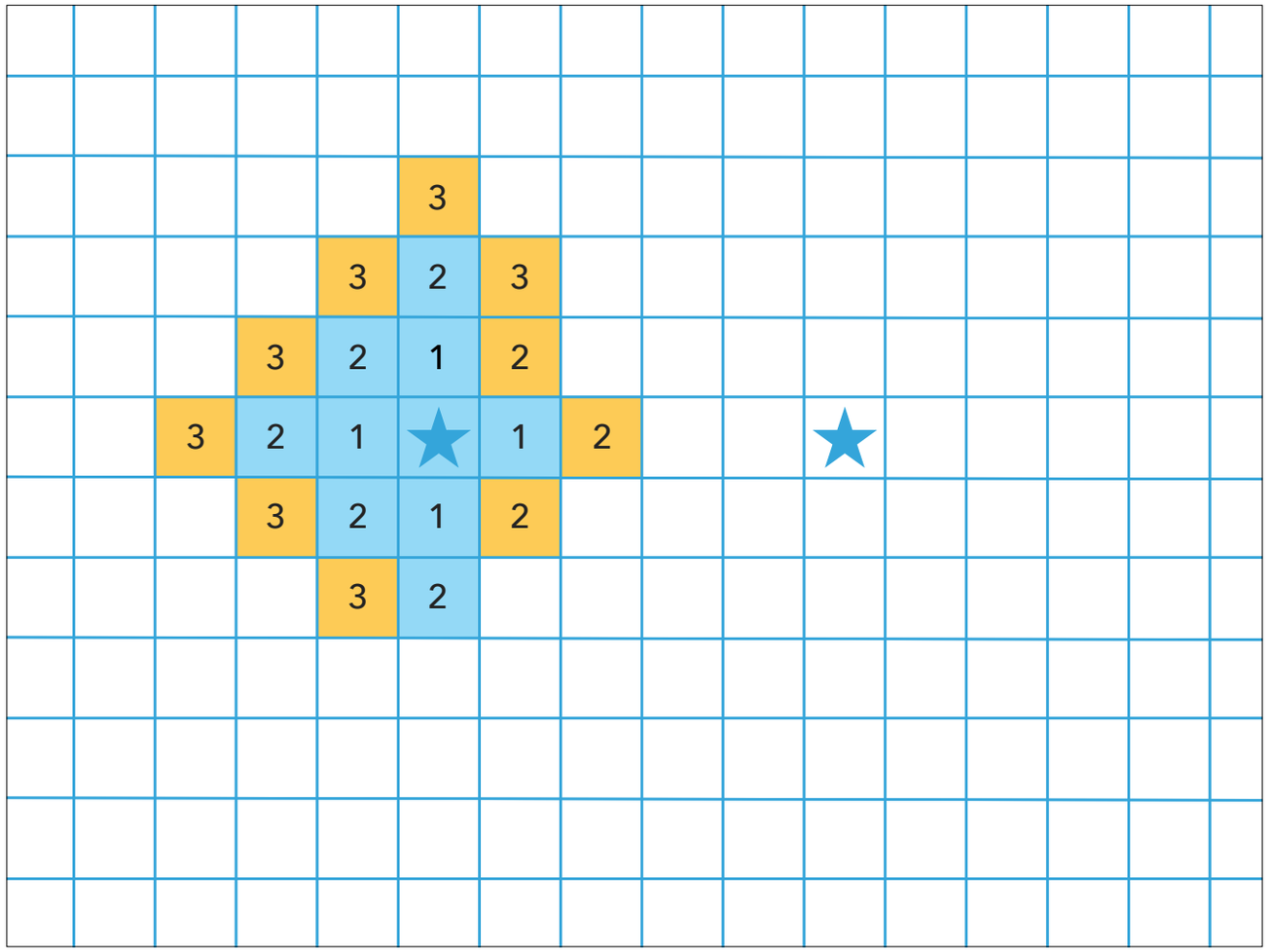
And we enqueue the paths to its neighbors with weights of 3



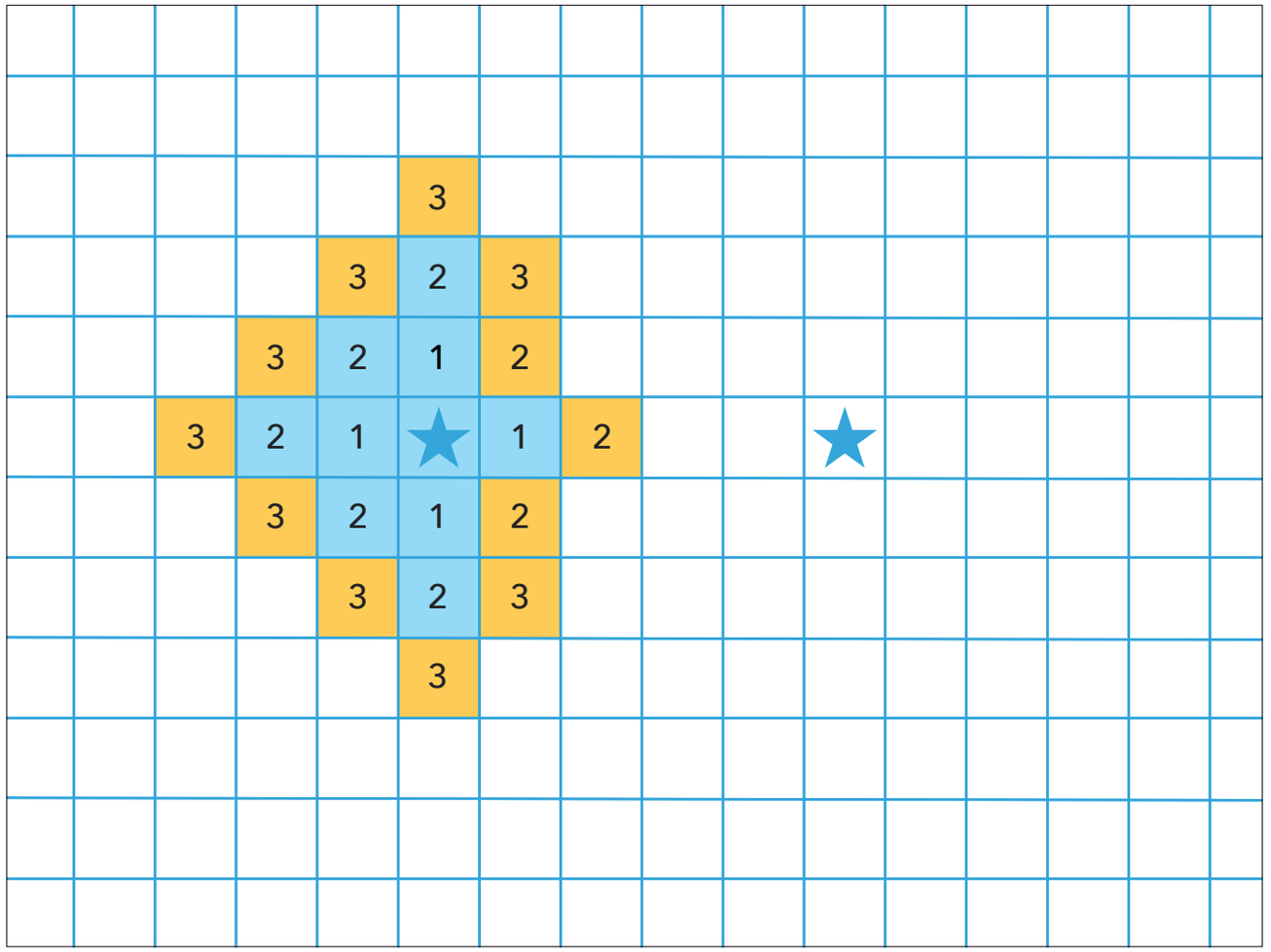
Dequeue the next one



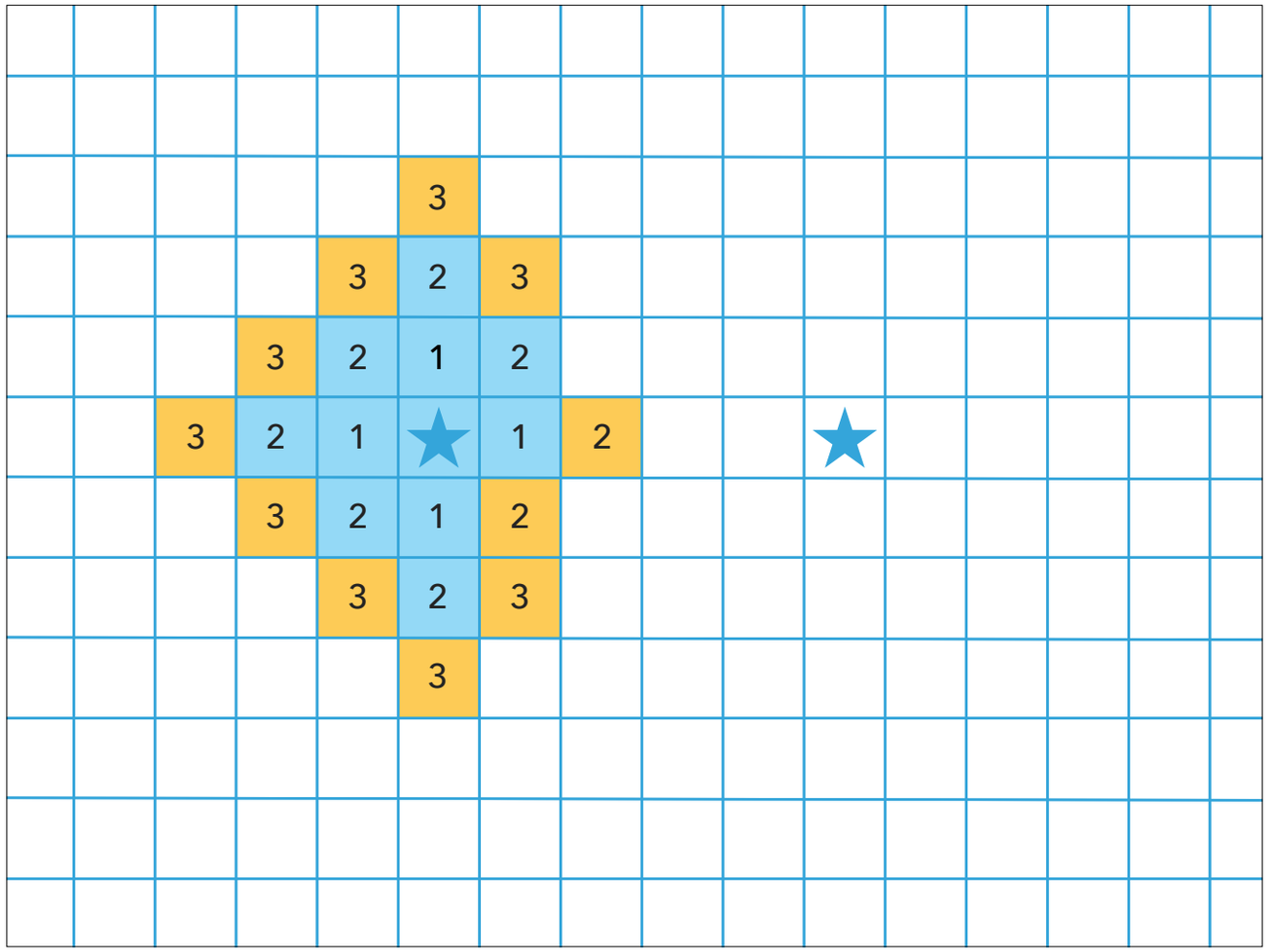
And we enqueue the paths to its neighbors with weights of 3



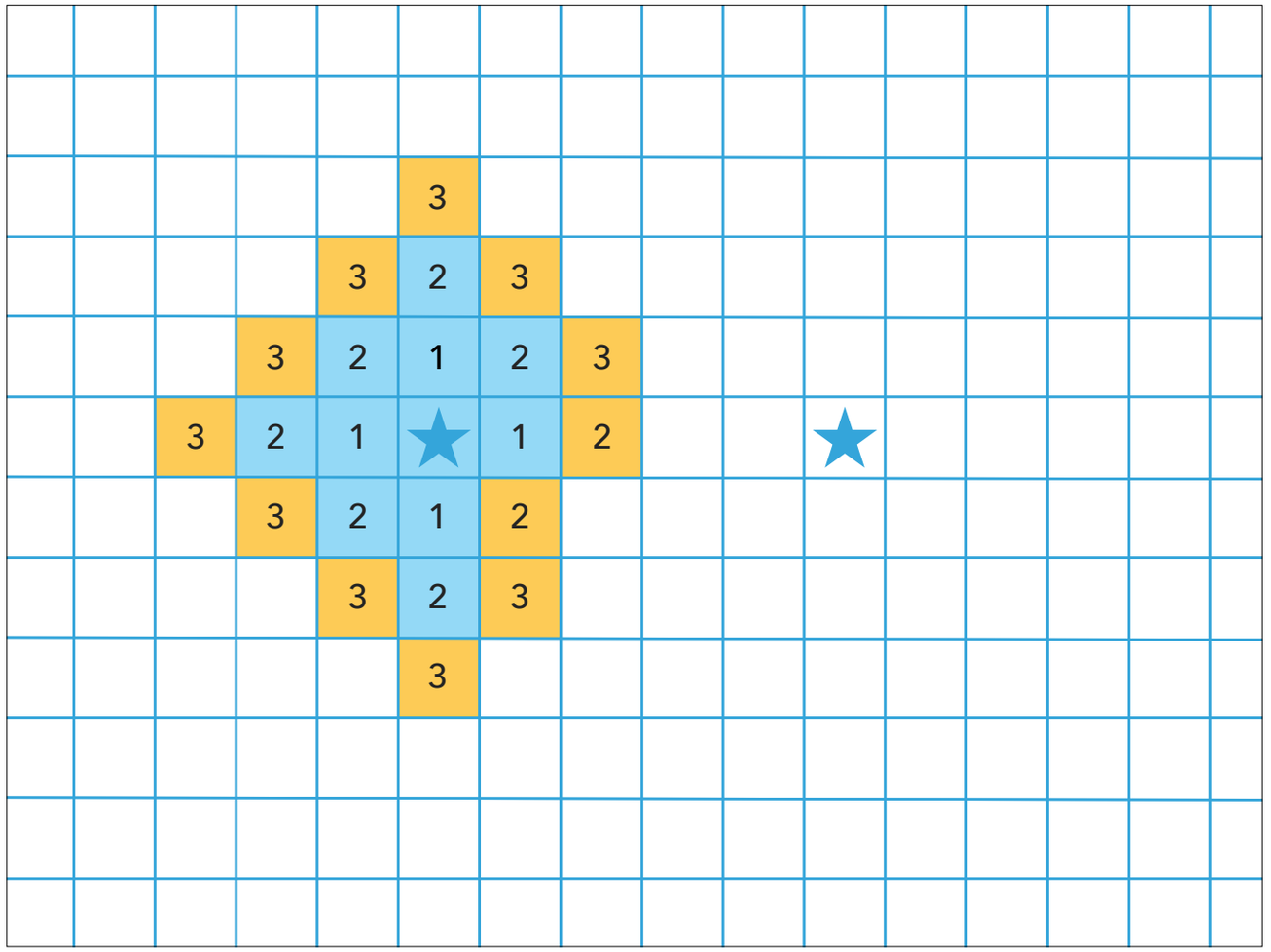
Dequeue the next one



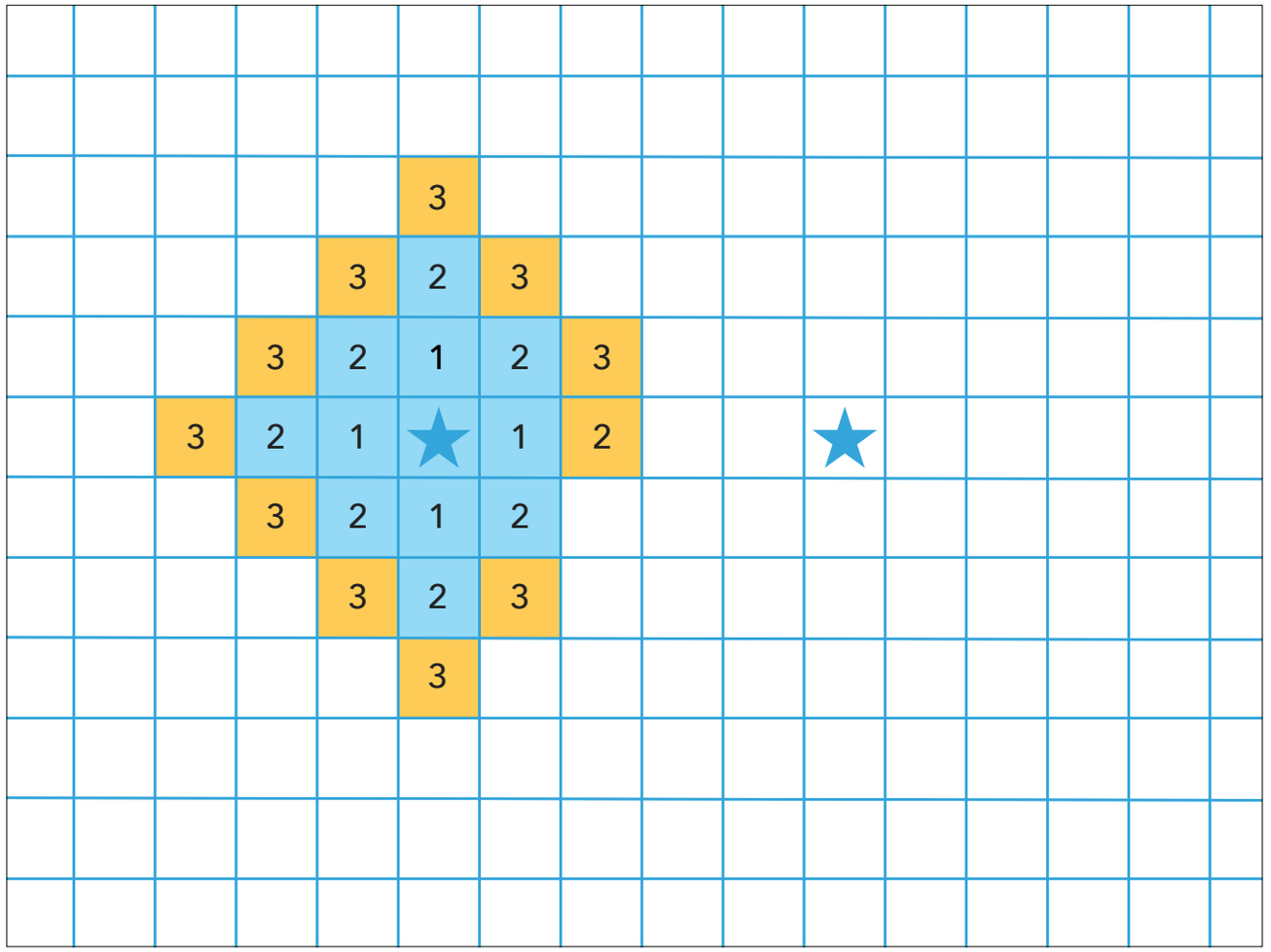
And we enqueue the paths to its neighbors with weights of 3



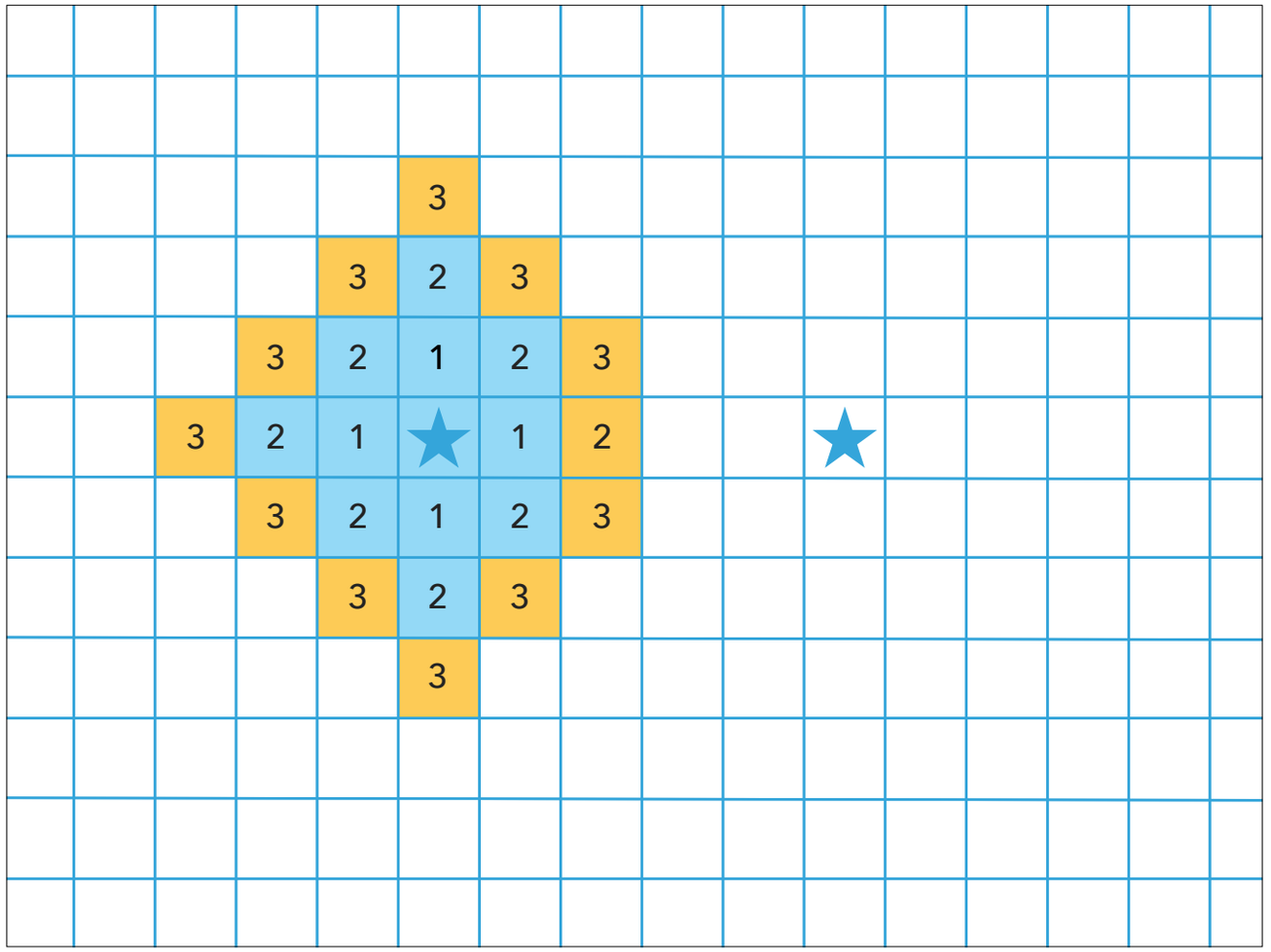
Dequeue the next one



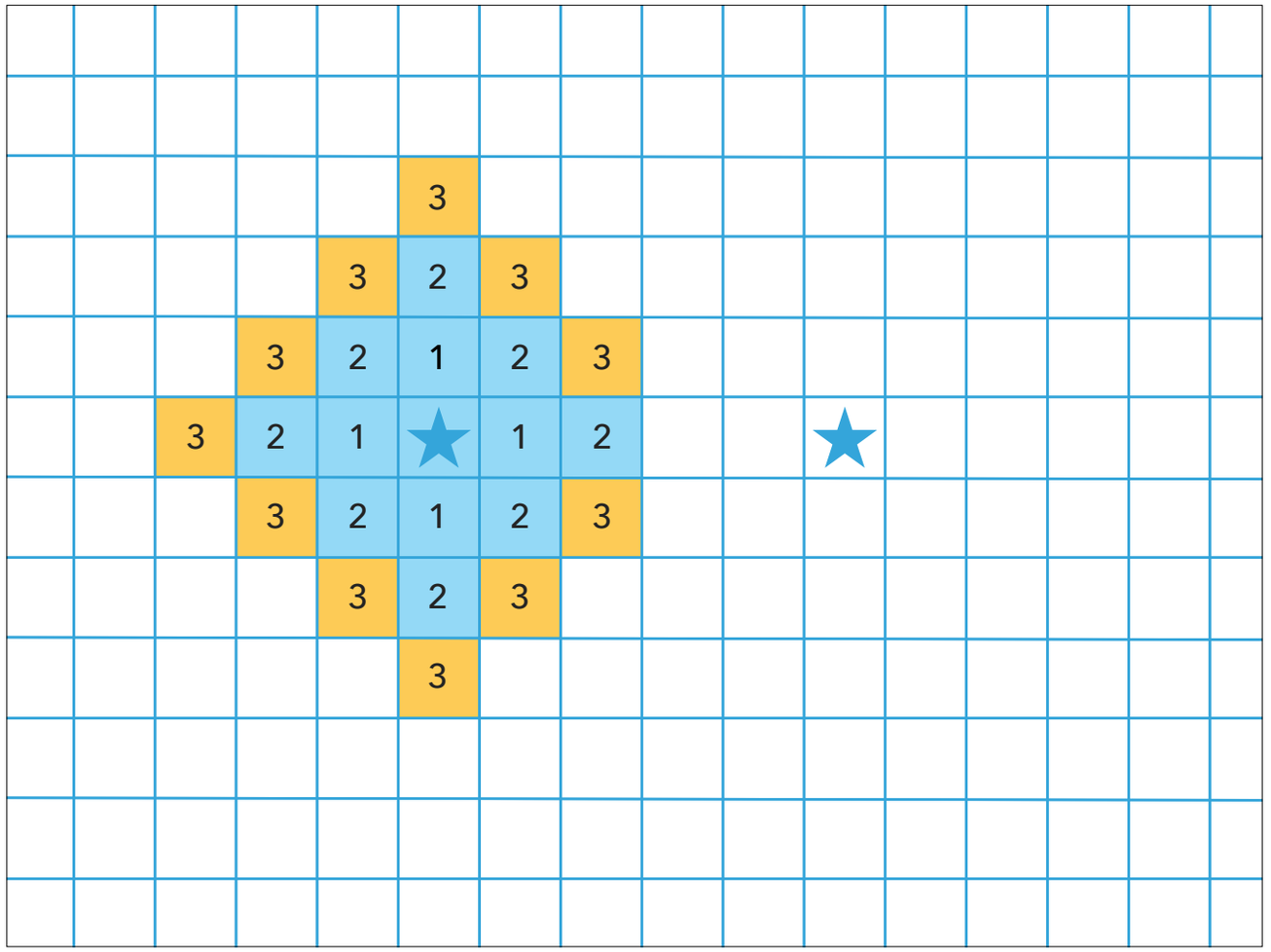
And we enqueue the paths to its neighbors with weights of 3



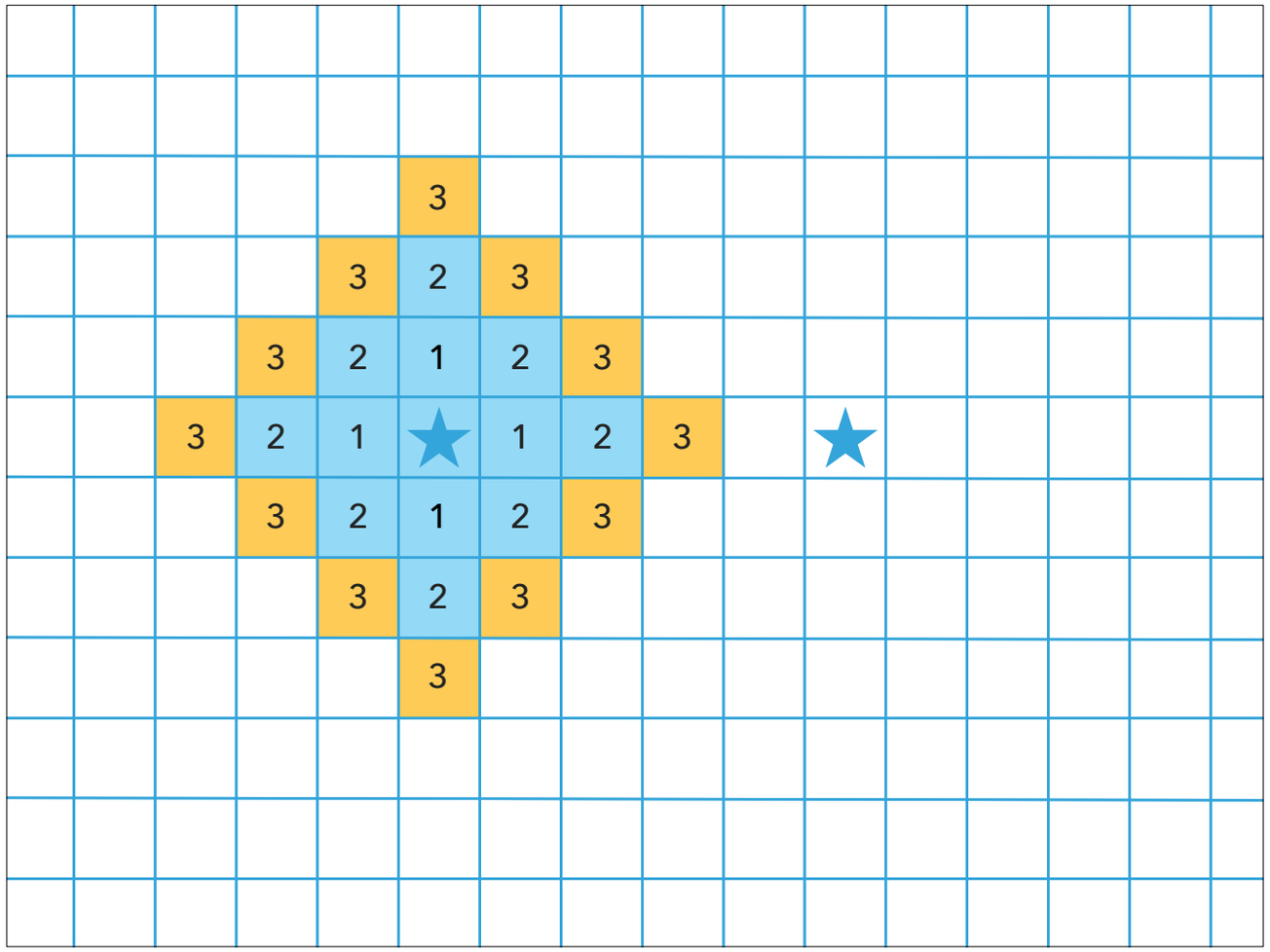
Dequeue the next one



And we enqueue the paths to its neighbors with weights of 3



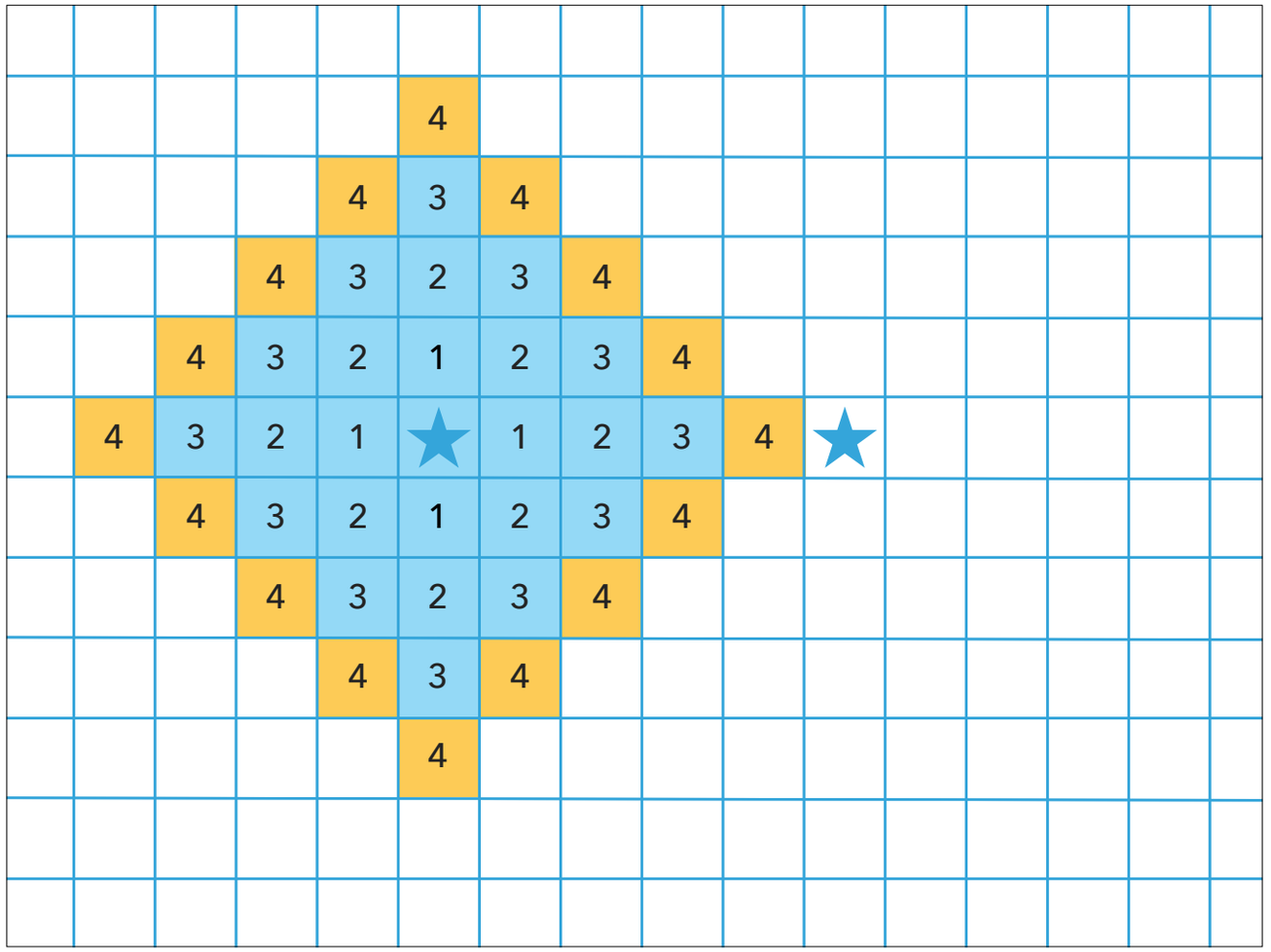
Dequeue the next one



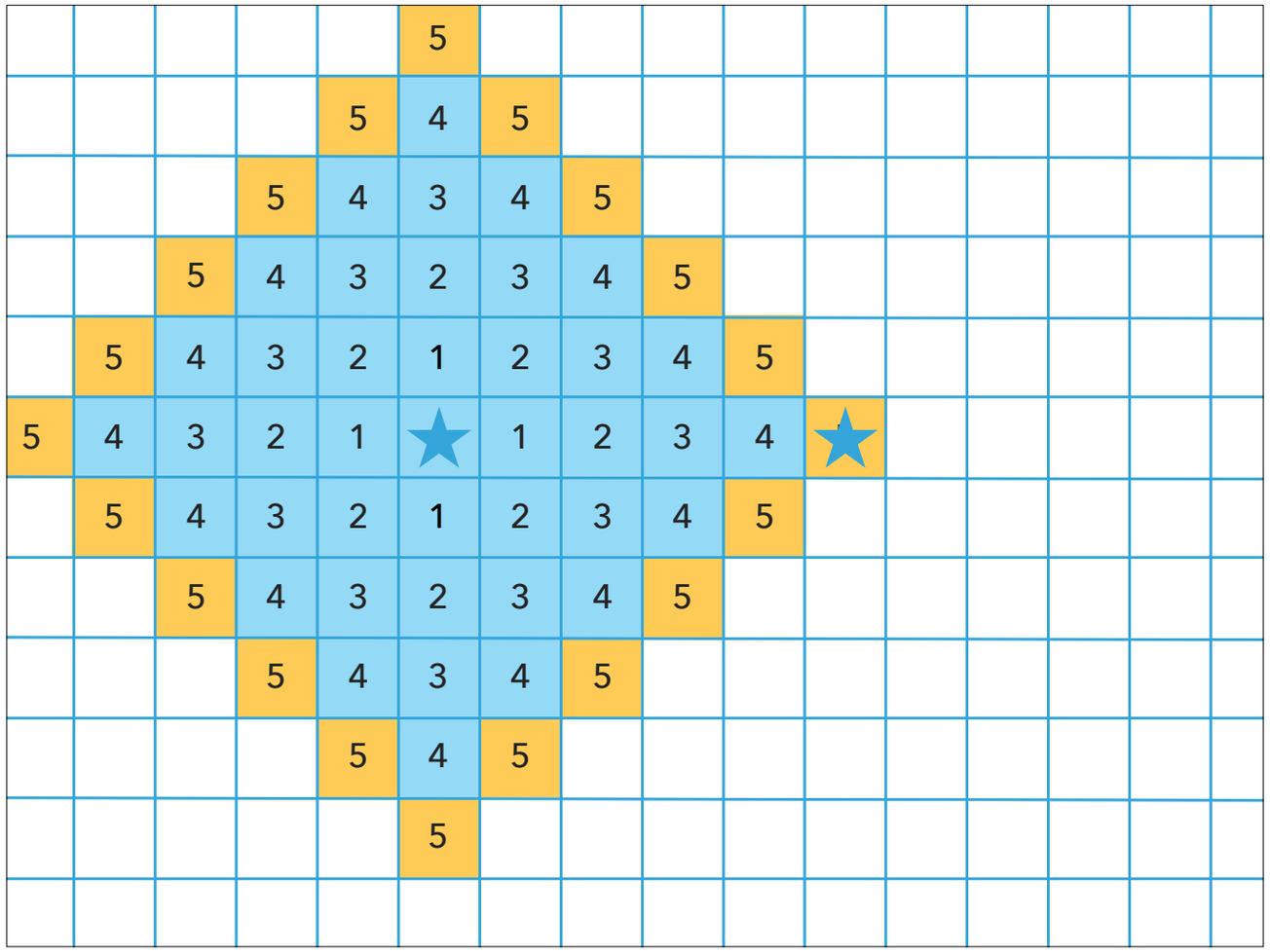
And we enqueue the paths to its neighbors with weights of 3

If it feels like I'm spending a little bit too much time on this, that's kind of the point. Dijkstra's isn't going to find this relatively straightforward path very quickly.

But just to move things along, let's skip ahead to having processed all the three nodes...

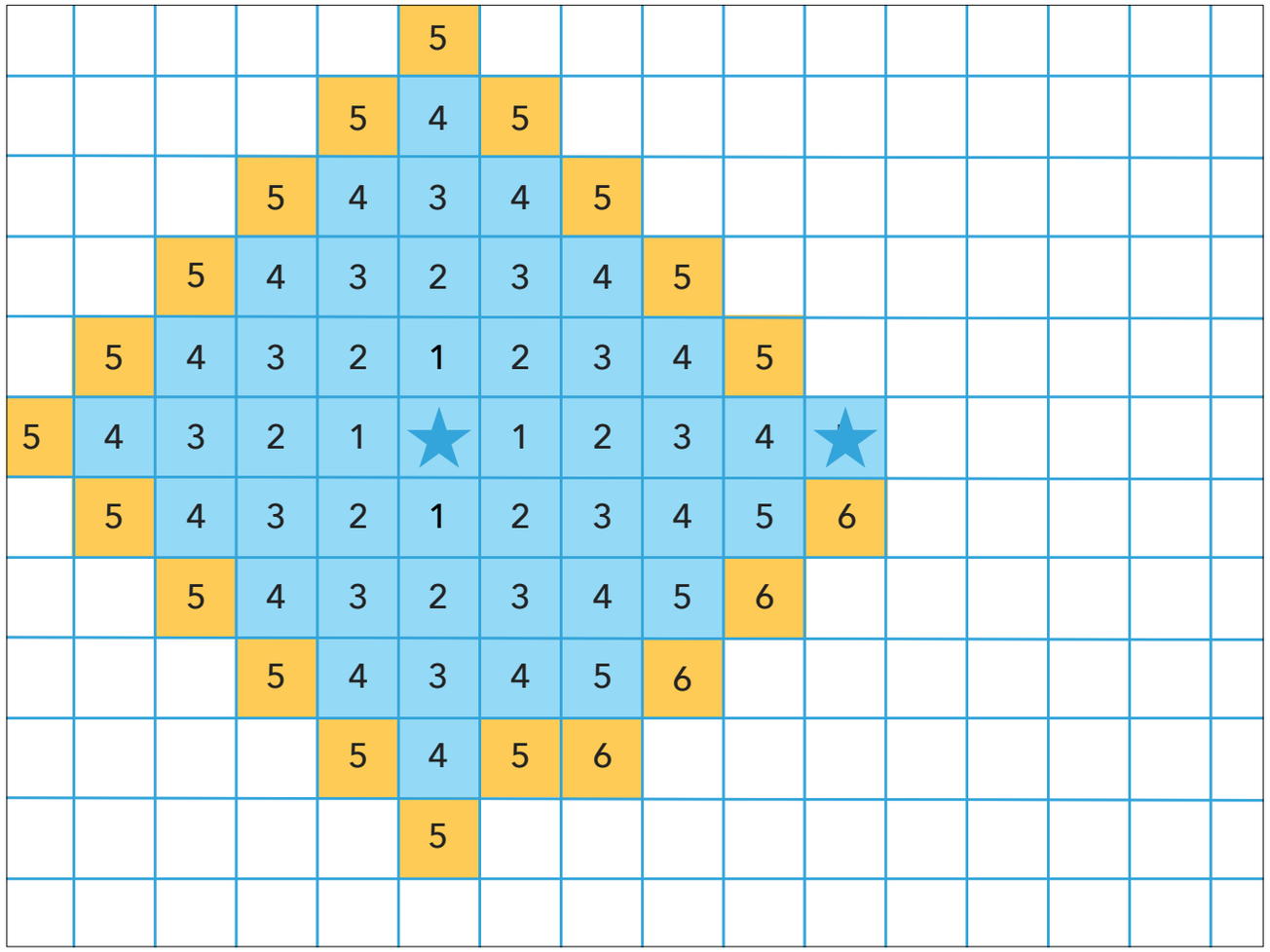


At which point our graph looks like this. We still don't have a solution. So let's process all of the four nodes...

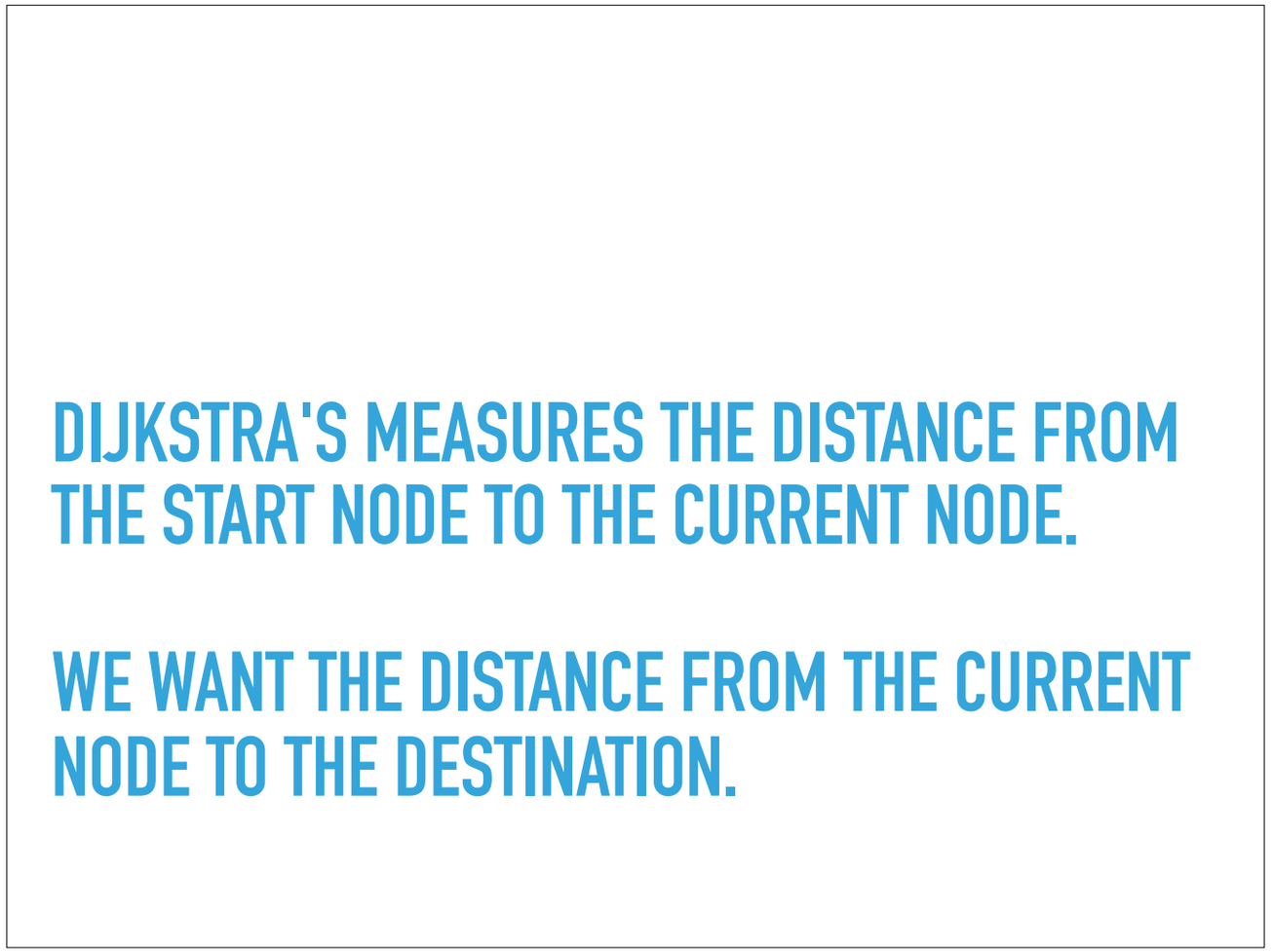


And our graph looks like this. Are we done yet?

Trick question - no we aren't. Remember that in Dijkstra's, it is important that you don't look at the end point until you've dequeued the path from the priority queue. You can be a little lax with this in BFS and DFS, but if you do it wrong here you might end up with a more expensive path. So we actually have to do one more round of this (or, depending on the order you process your nodes in, part of a round)



and then you'll find the path. But this begs the question, why does the algorithm waste all of this time searching over here on the left, when the destination isn't there? That's because Dijkstra's prioritizes paths depending on the cost so far, and doesn't care about what the rest of the path might look like. Simply put–

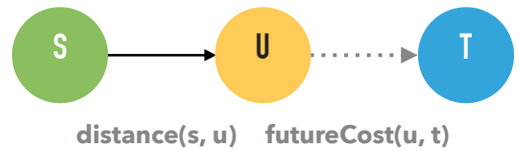


Dijkstra's measures the distance from the start node to the current node. We want the distance from the current node to the destination.

And that means if we want to find the fastest path, we need to be able to see into the future.

SEEING THE FUTURE

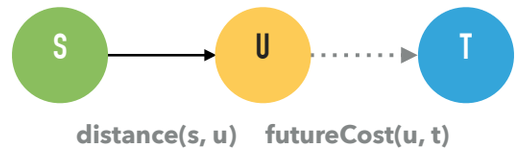
FORMAL DEFINITIONS



The way that this is formally defined is as follows:

We have three nodes, s (start), t (terminal), and u (the current one).

FORMAL DEFINITIONS

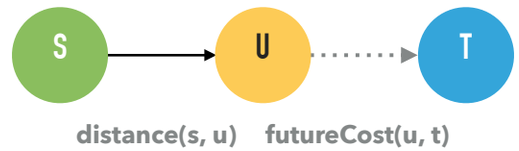


DIJKSTRA'S

priority(u) = distance(s, u)

In Dijkstra's algorithm, we define the priority of the path that starts at s and terminates as u as the distance between s and u.

FORMAL DEFINITIONS



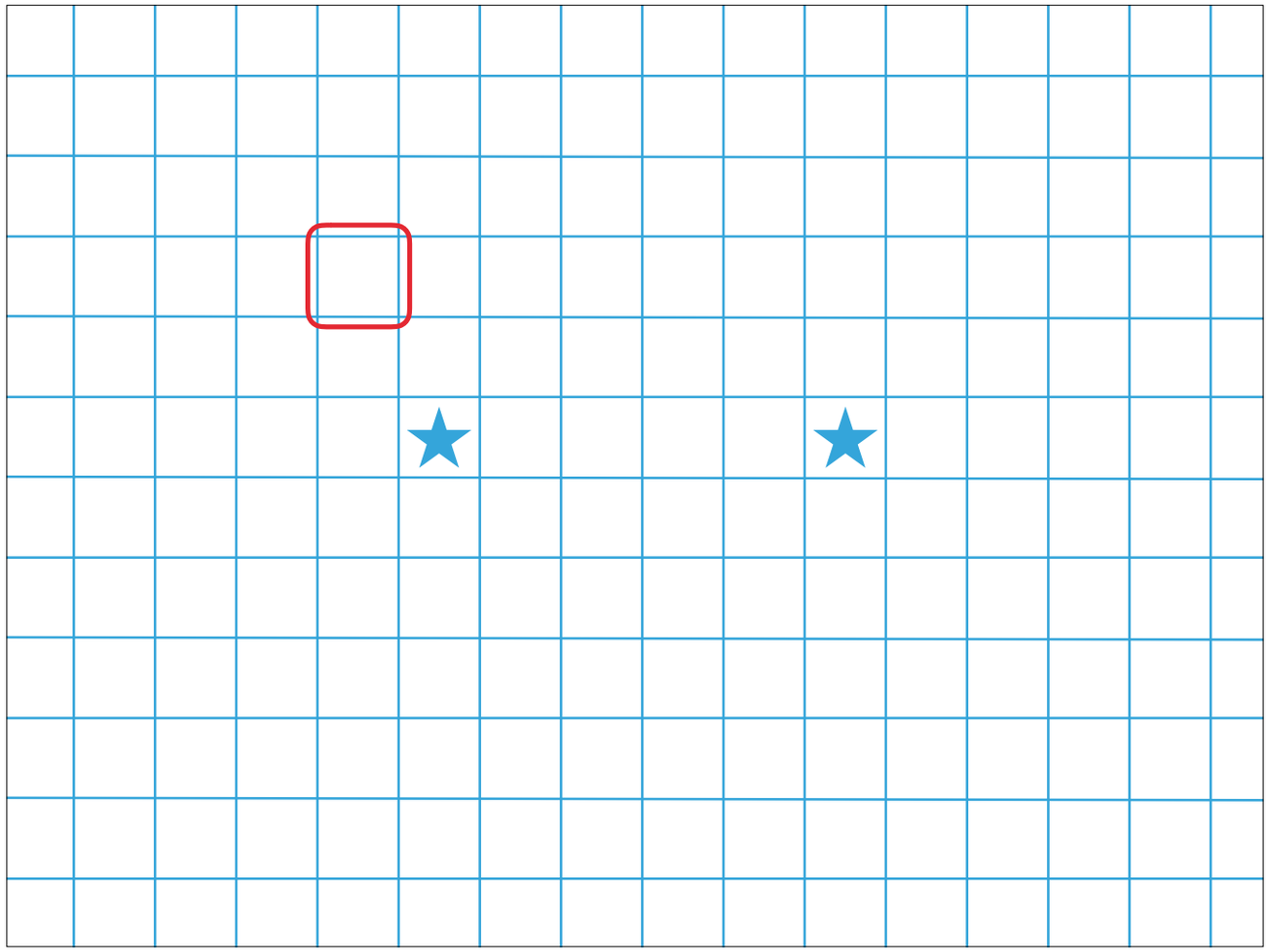
DIJKSTRA'S

priority(u) = distance(s, u)

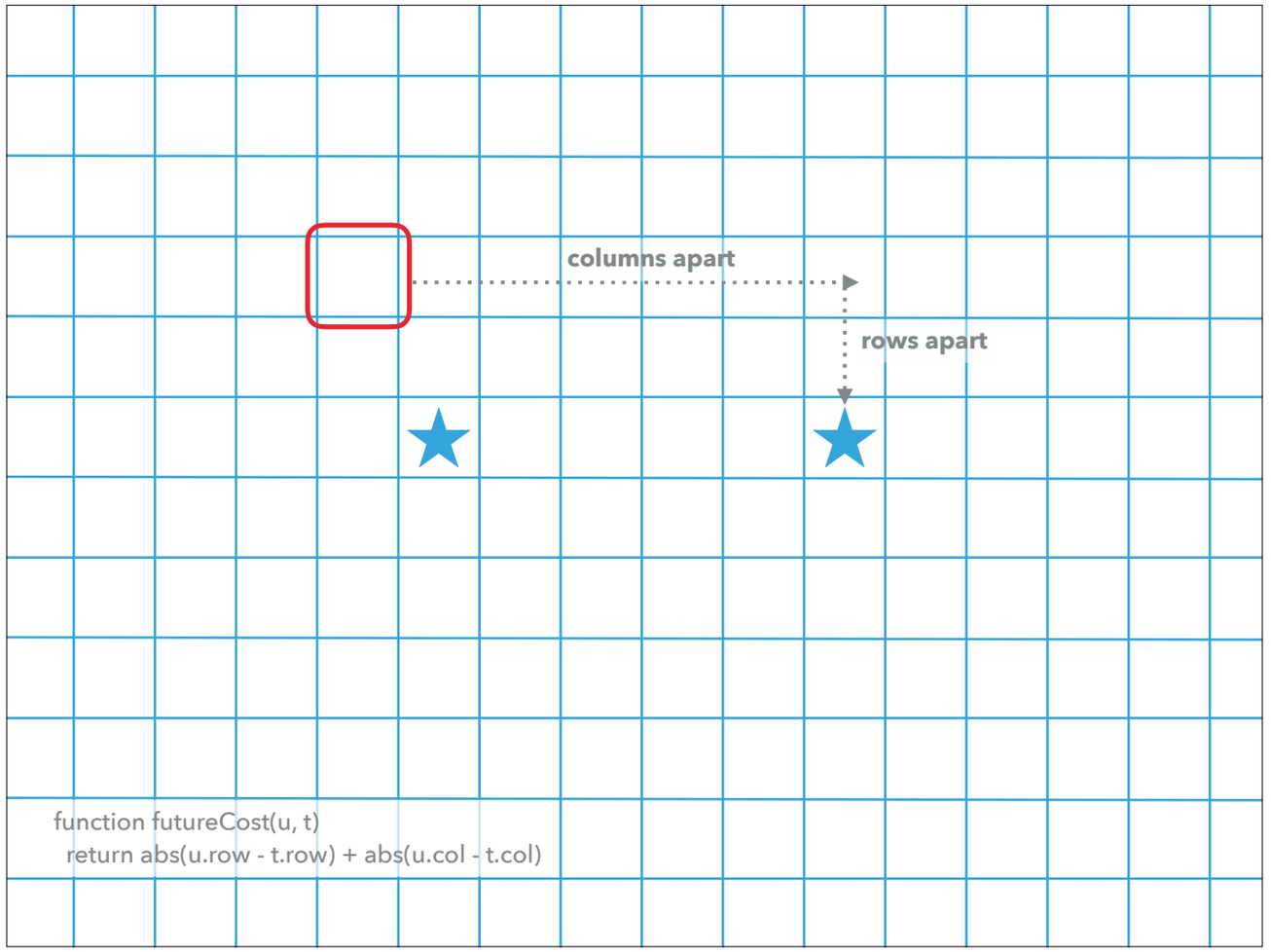
IDEAL

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

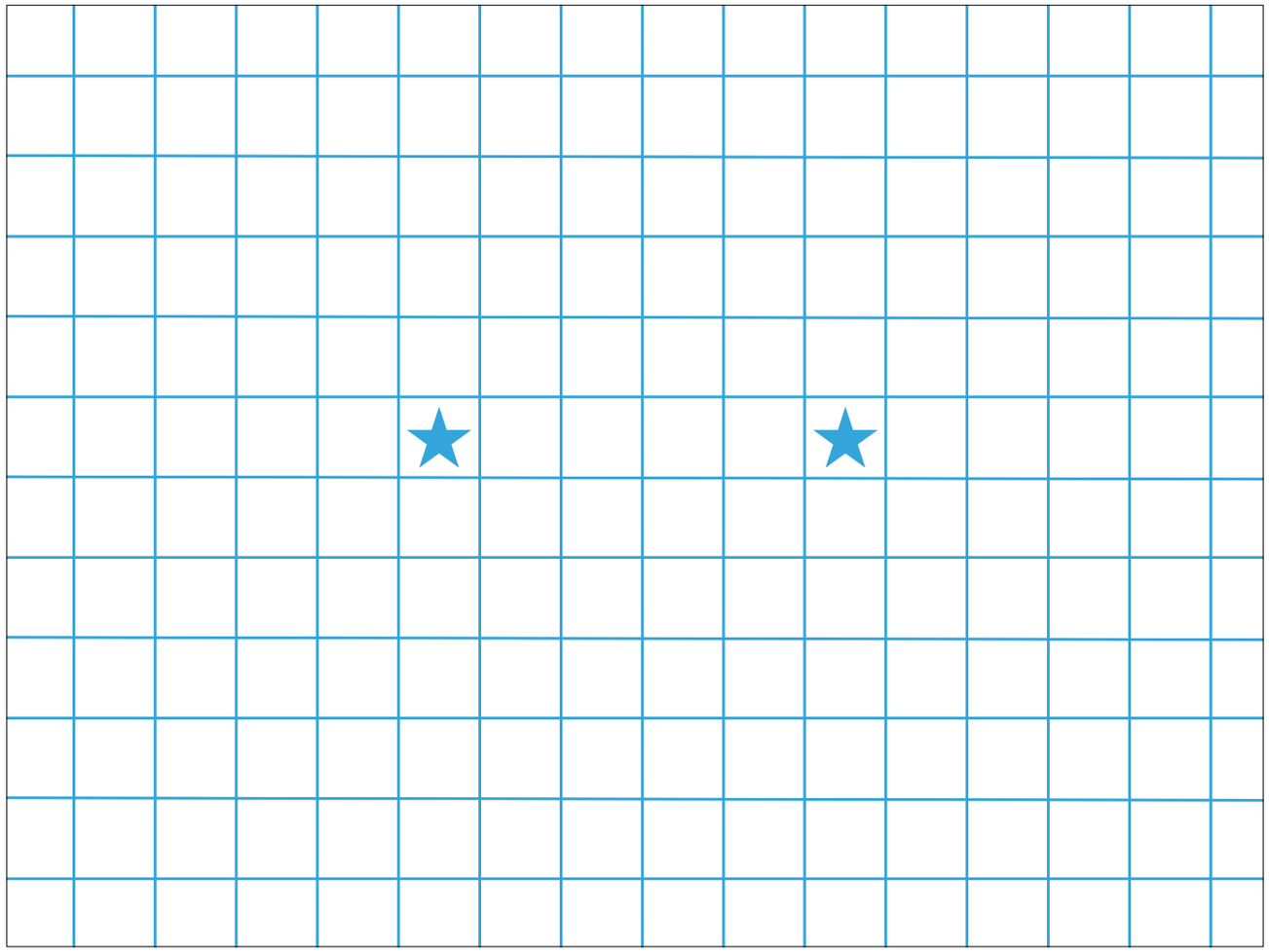
But in an ideal world, we'd be able to know how far u is from t, and take that into account in calculating the cost of the path.



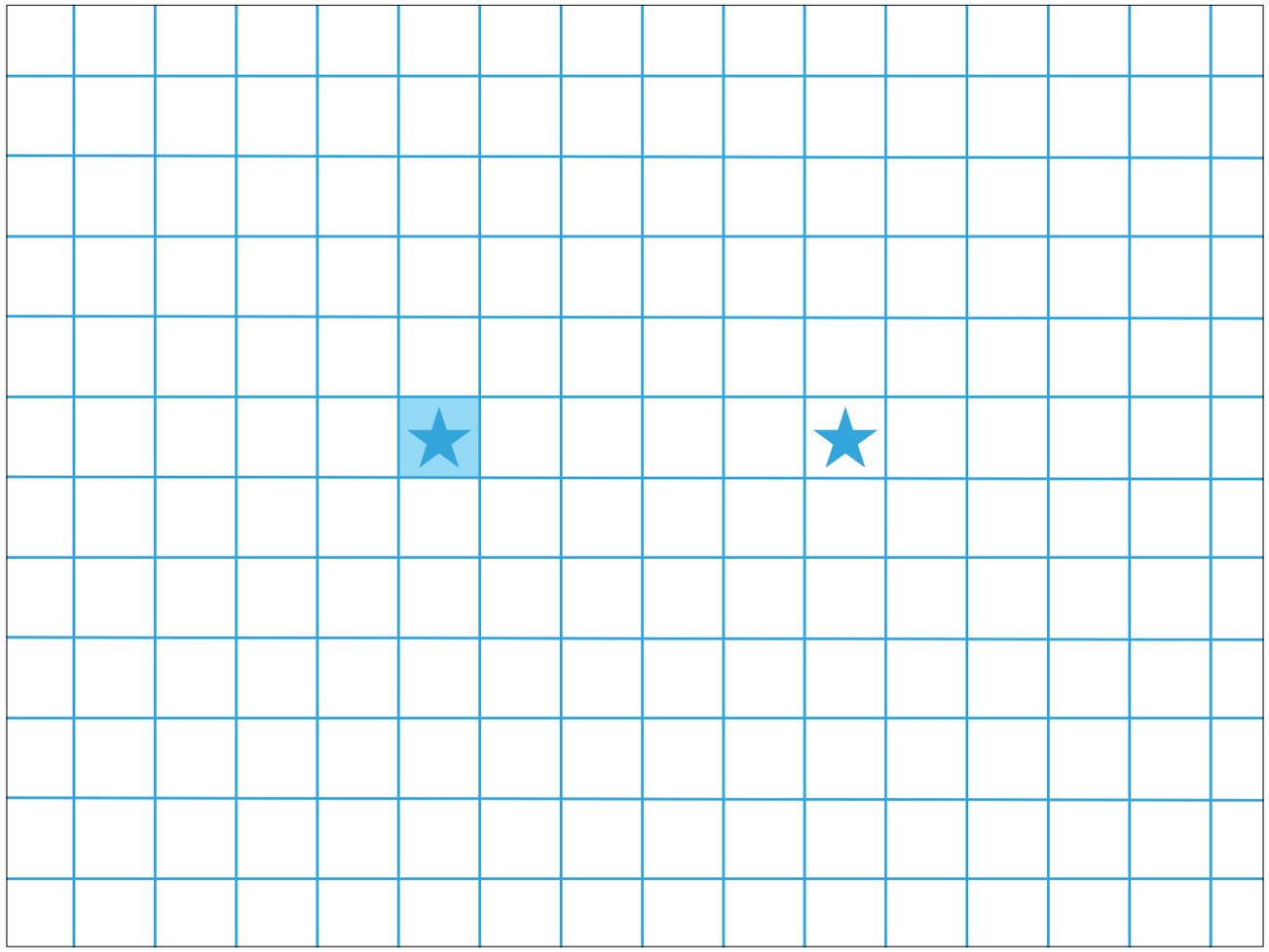
So how would we find the the future cost of traveling from the marked square to the destination?



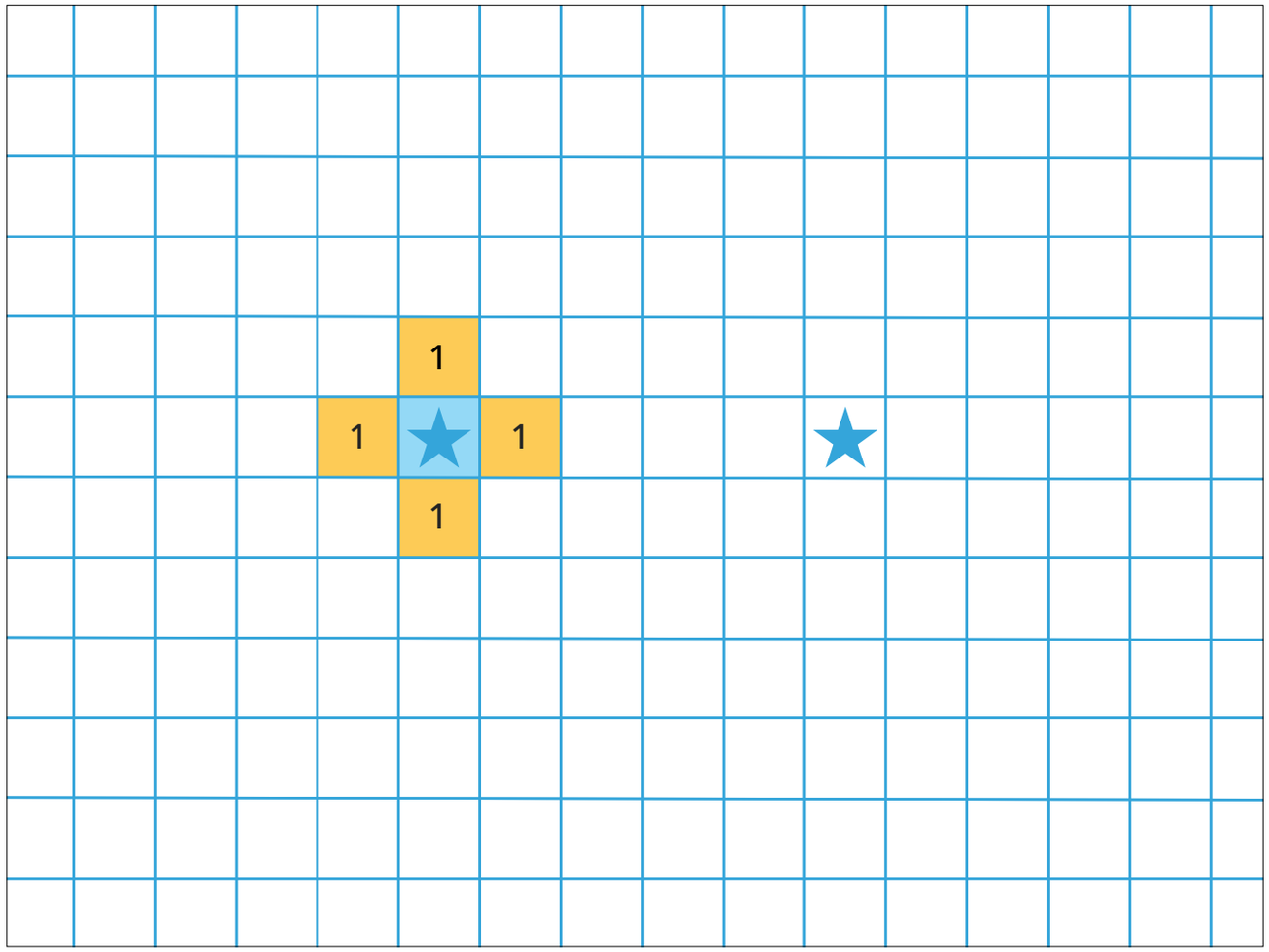
We can just count the number cells to travel.



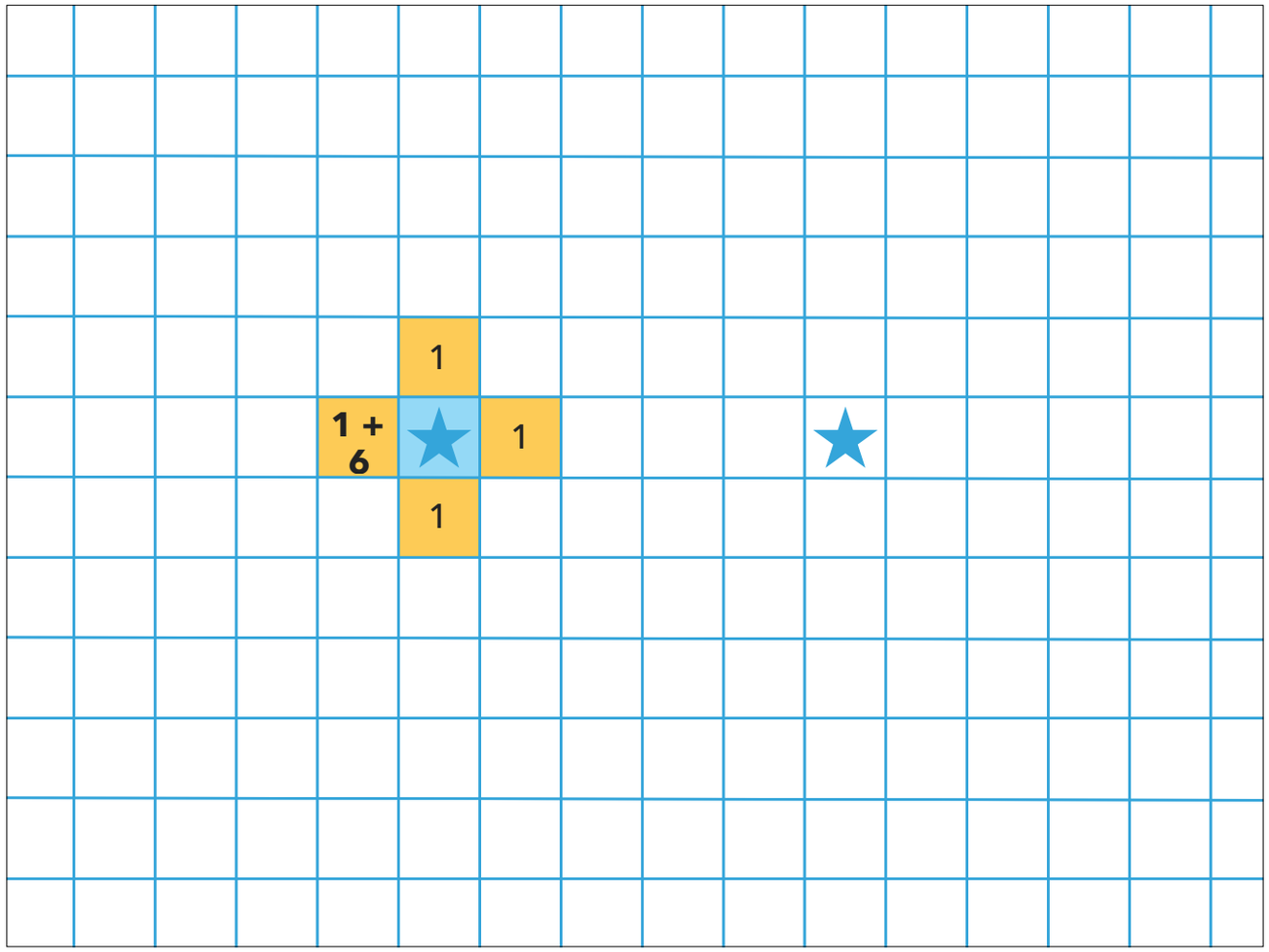
So let's try finding a path using this new ideal algorithm.



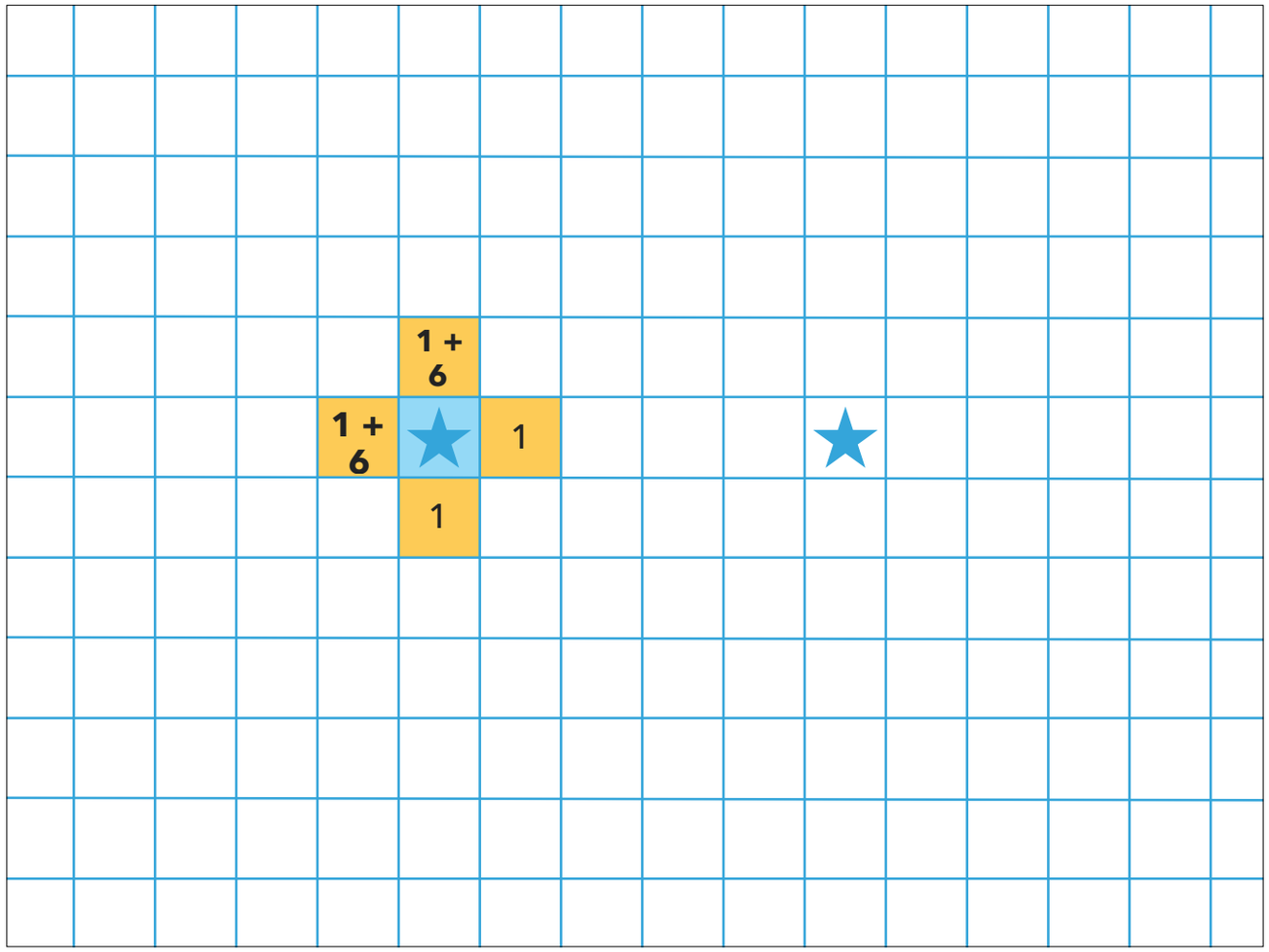
We start by enqueueing the start node.



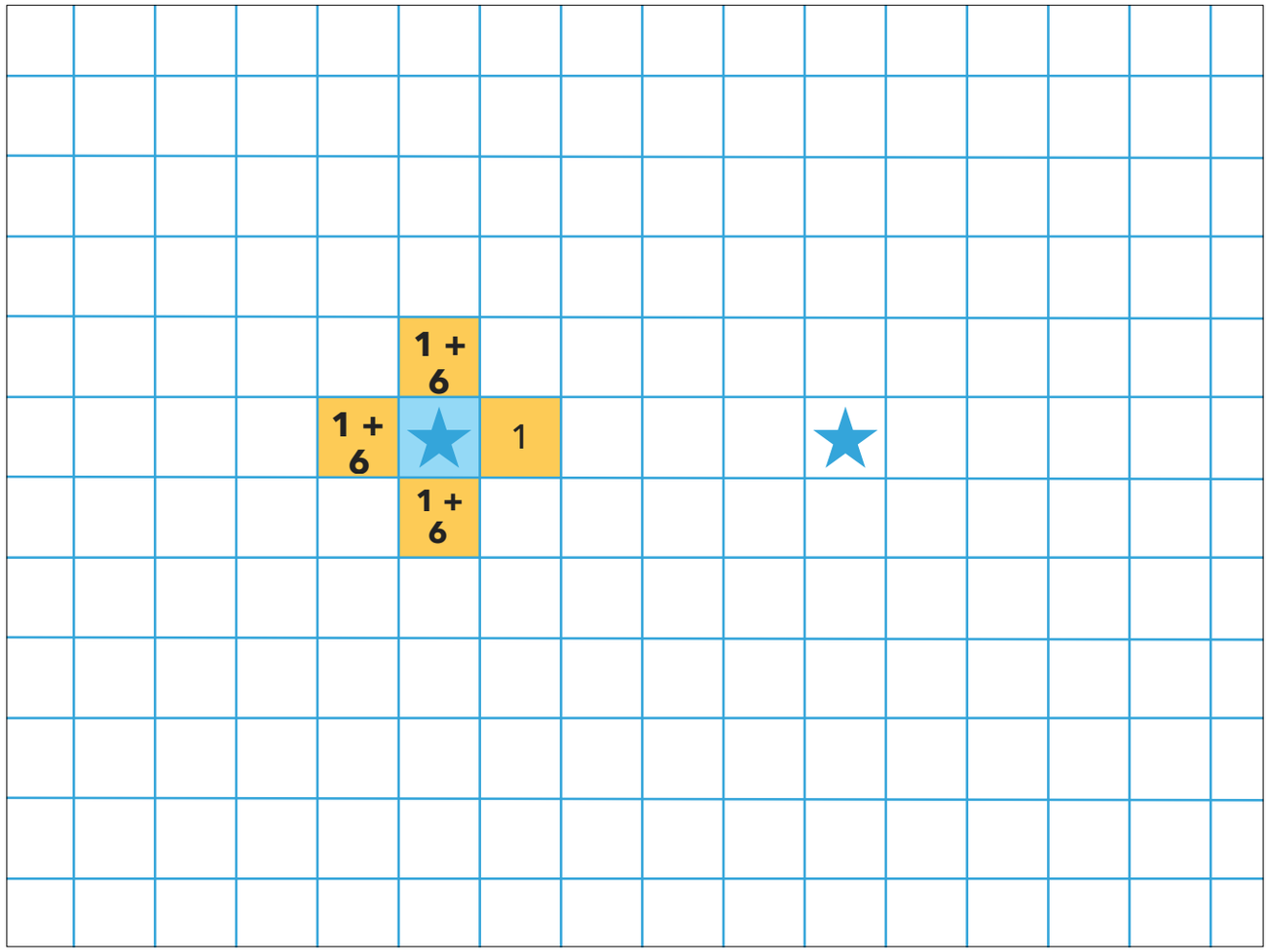
And then we dequeue it and add enqueue all of its neighbors with weights 1. But we also need to take into the future cost of each node a swell.



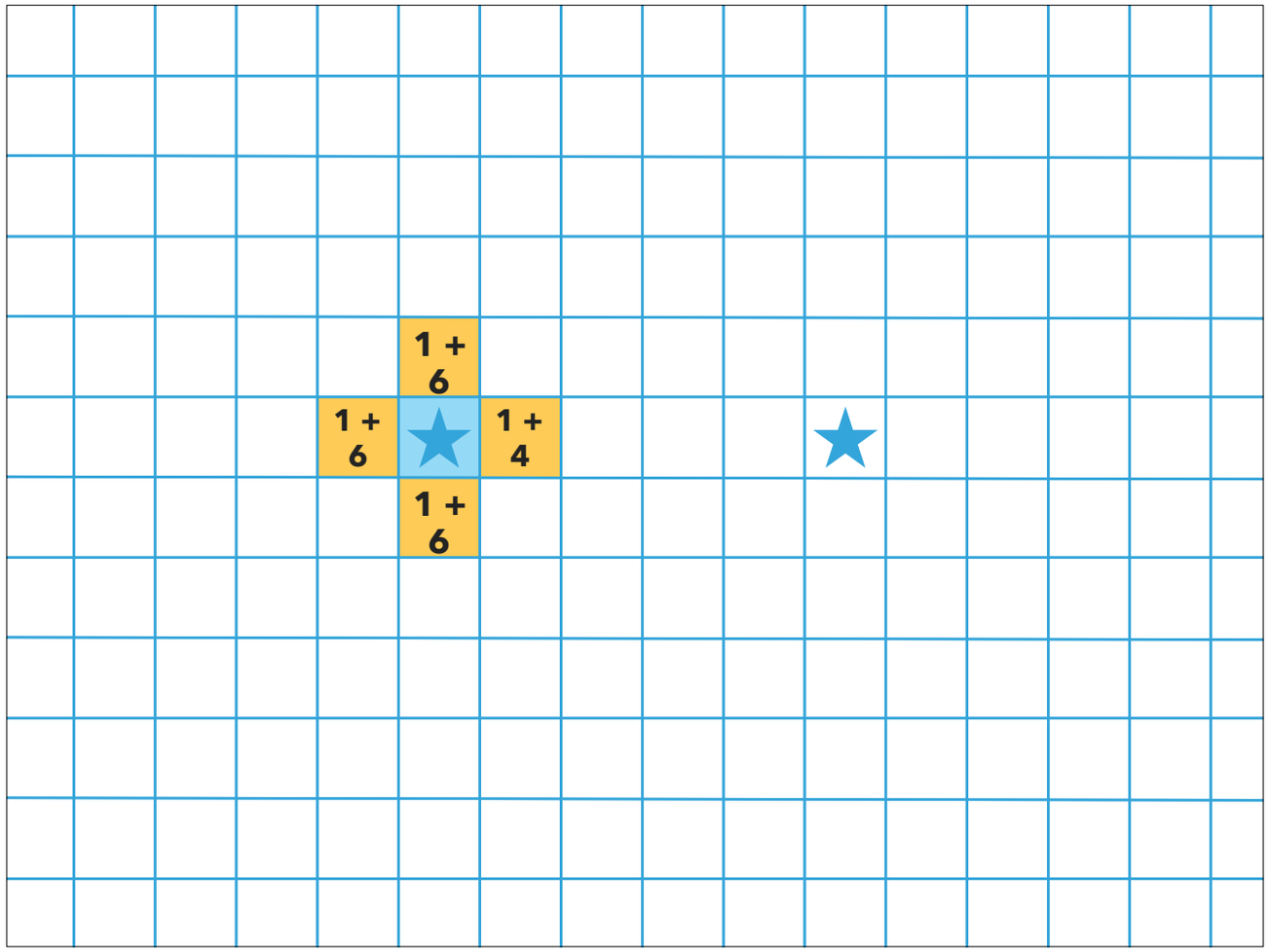
So we add the future cost of the first node



and the second

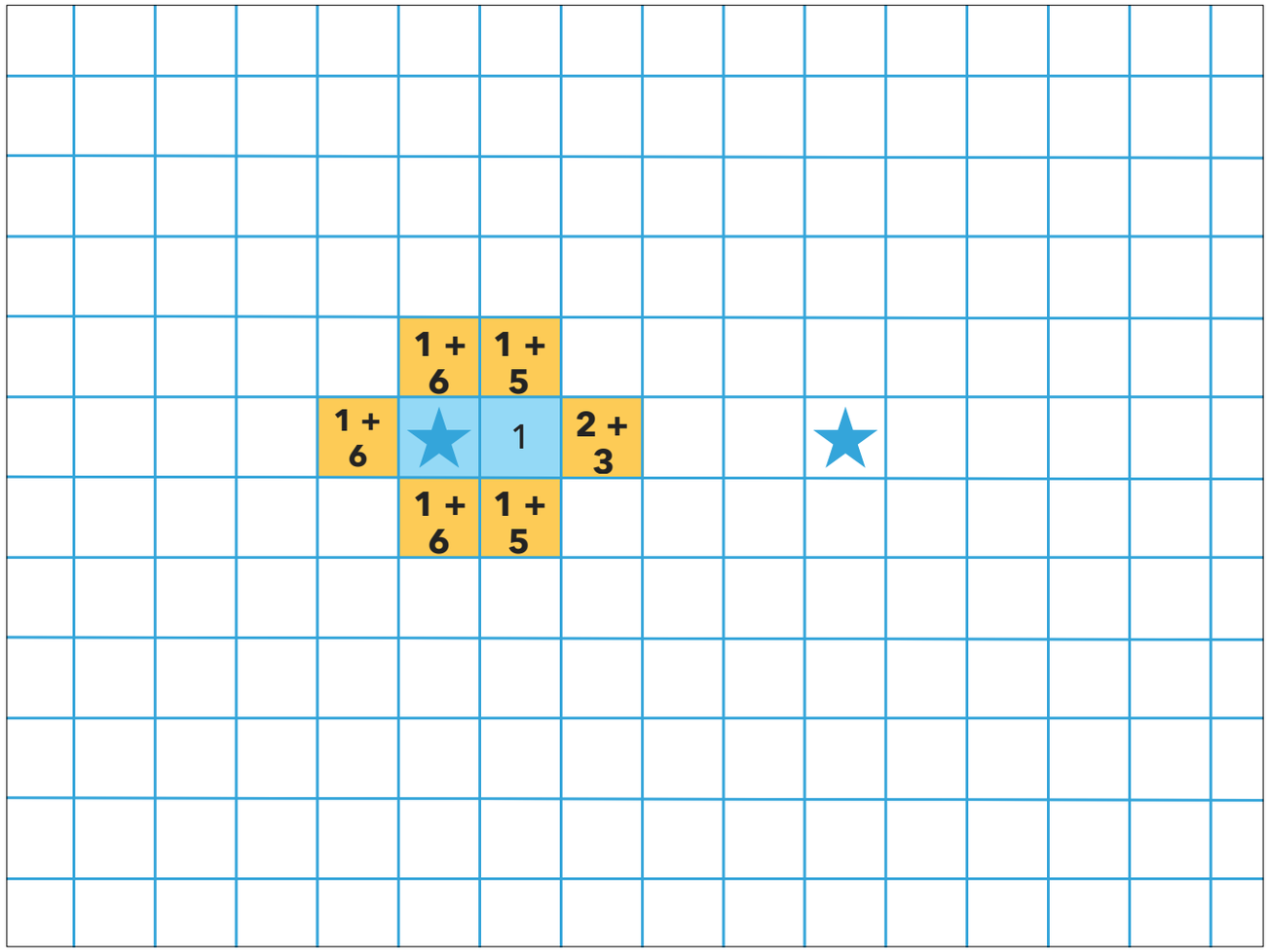


the third



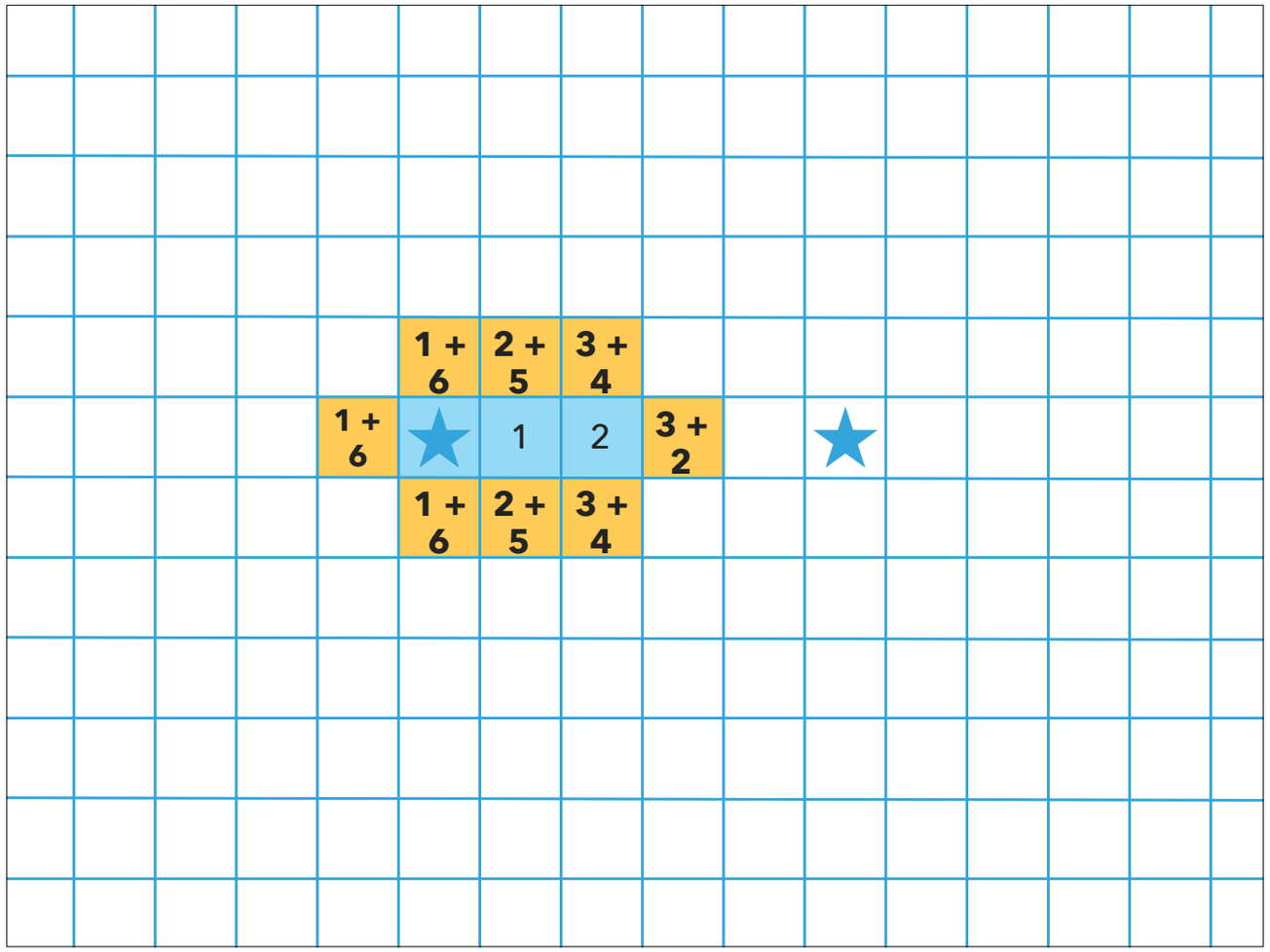
and the fourth.

And now we have a node that's clearly the cheapest possible node and the one we should consider first.

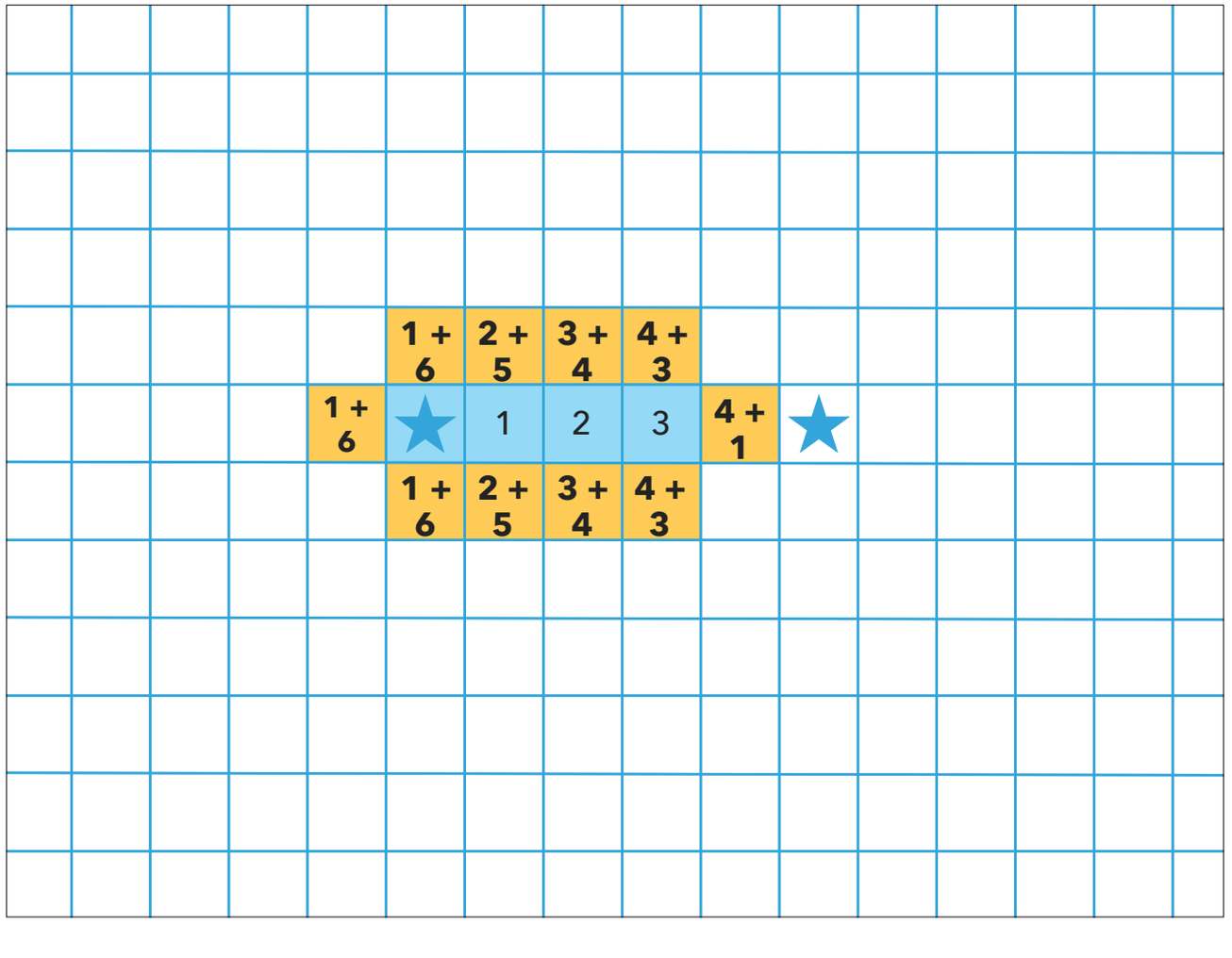


So we dequeue that one, and add its neighbors. And we still have a node that's clearly the cheapest.

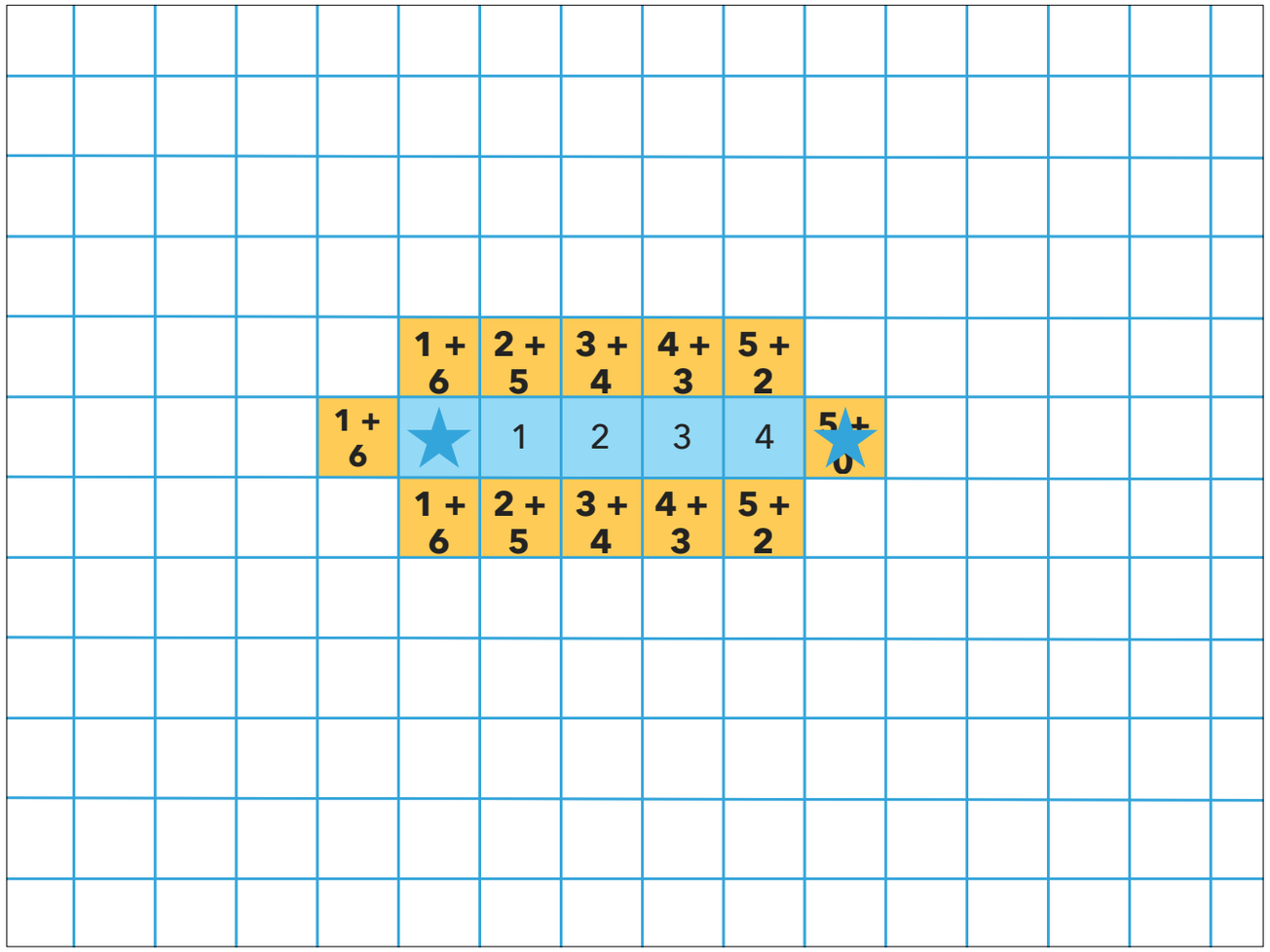
(One thing to note: the cost of the path to the dequeued node is *actually* cost 1, it doesn't include the future cost)



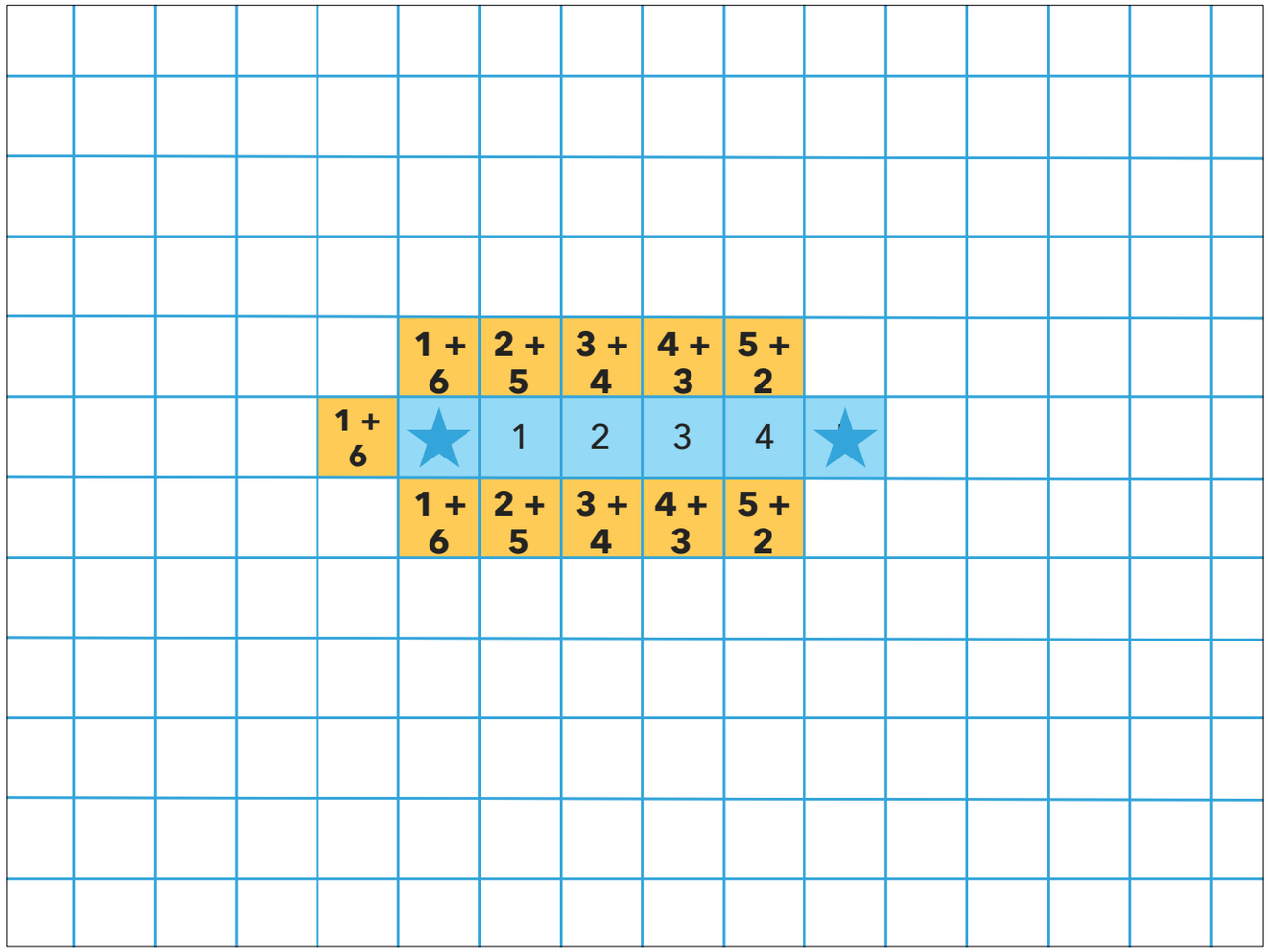
We dequeue that one, and we still have a node that's the cheapest.



We dequeue that one, and we still have a node that's the cheapest.

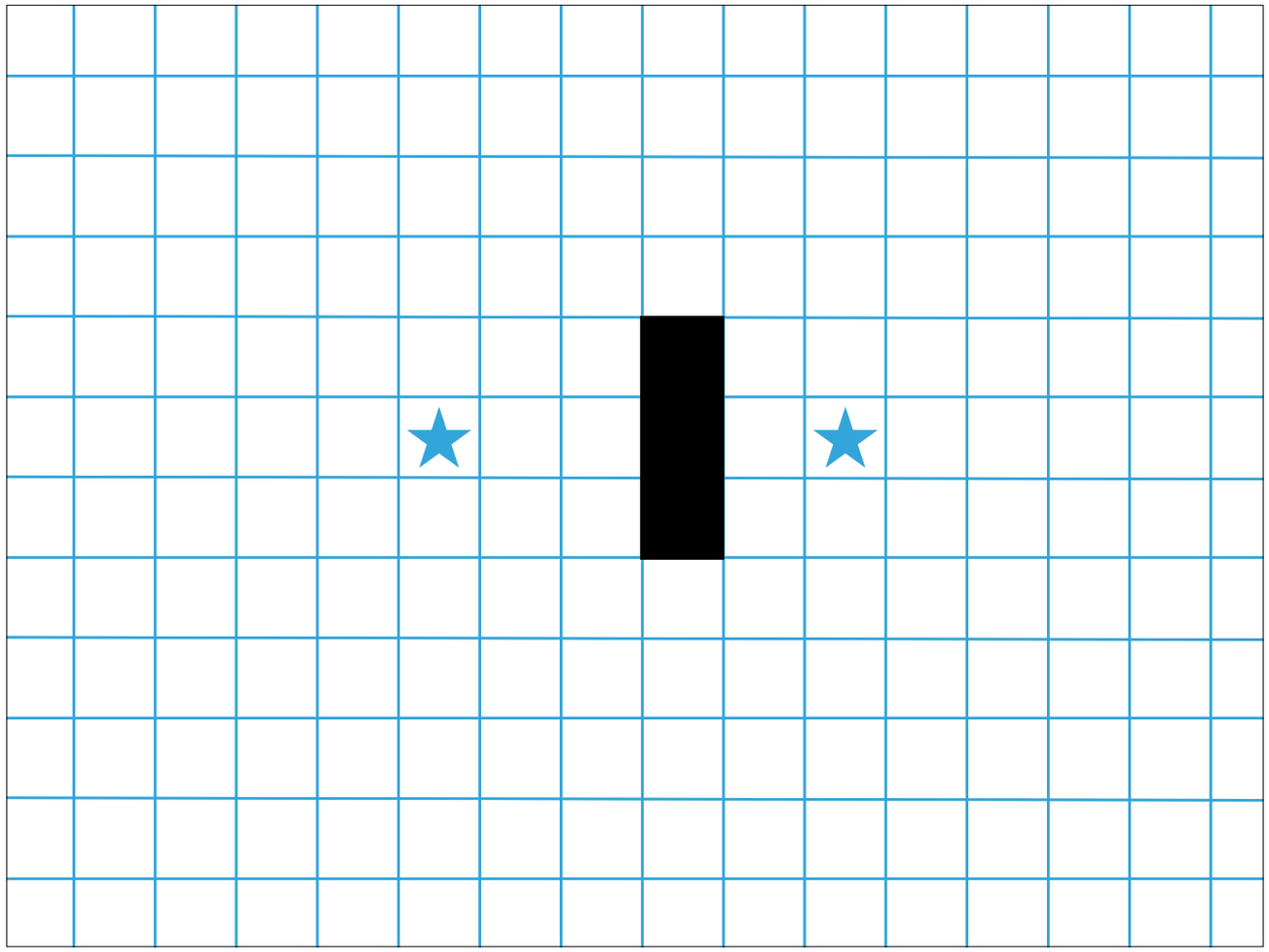


We dequeue that one, and we still have a node that's the cheapest. And now we just have one step left.



And we found the path! And we explored a lot less space than we did with Djikstra's. It was easy!

But the problem is here... you can't always calculate the future cost. It makes sense on a graph like this, but there could be some graphs where it isn't easy. For example

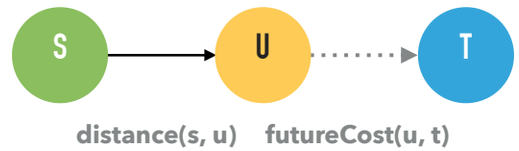


If there's a wall in the middle of the graph. You can't know for sure the distance between any point and the destination, because it's possible it will go around the wall. So if we can't see into the future, the best we can do is

MAKING GOOD LIFE DECISIONS

Make as good a decision as we possibly can

FORMAL DEFINITIONS

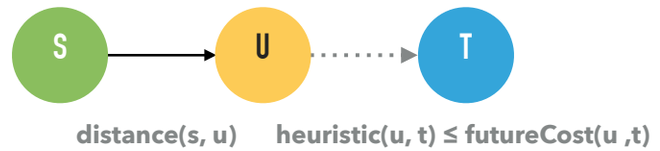


IDEAL

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

As we said earlier, the ideal priority would use the distance between s and u, as well as the future cost between u and t. But we can't always figure out the future cost. So instead, we're going to use what we call a heuristic.

FORMAL DEFINITIONS



IDEAL

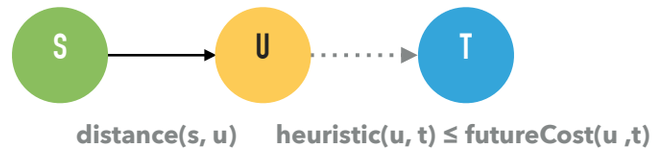
priority(u) = distance(s, u)
+ futureCost(u, t)

A*

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

A heuristic is a function that estimates the future cost. And we use that instead of the future cost when determining the priority of the future cost.

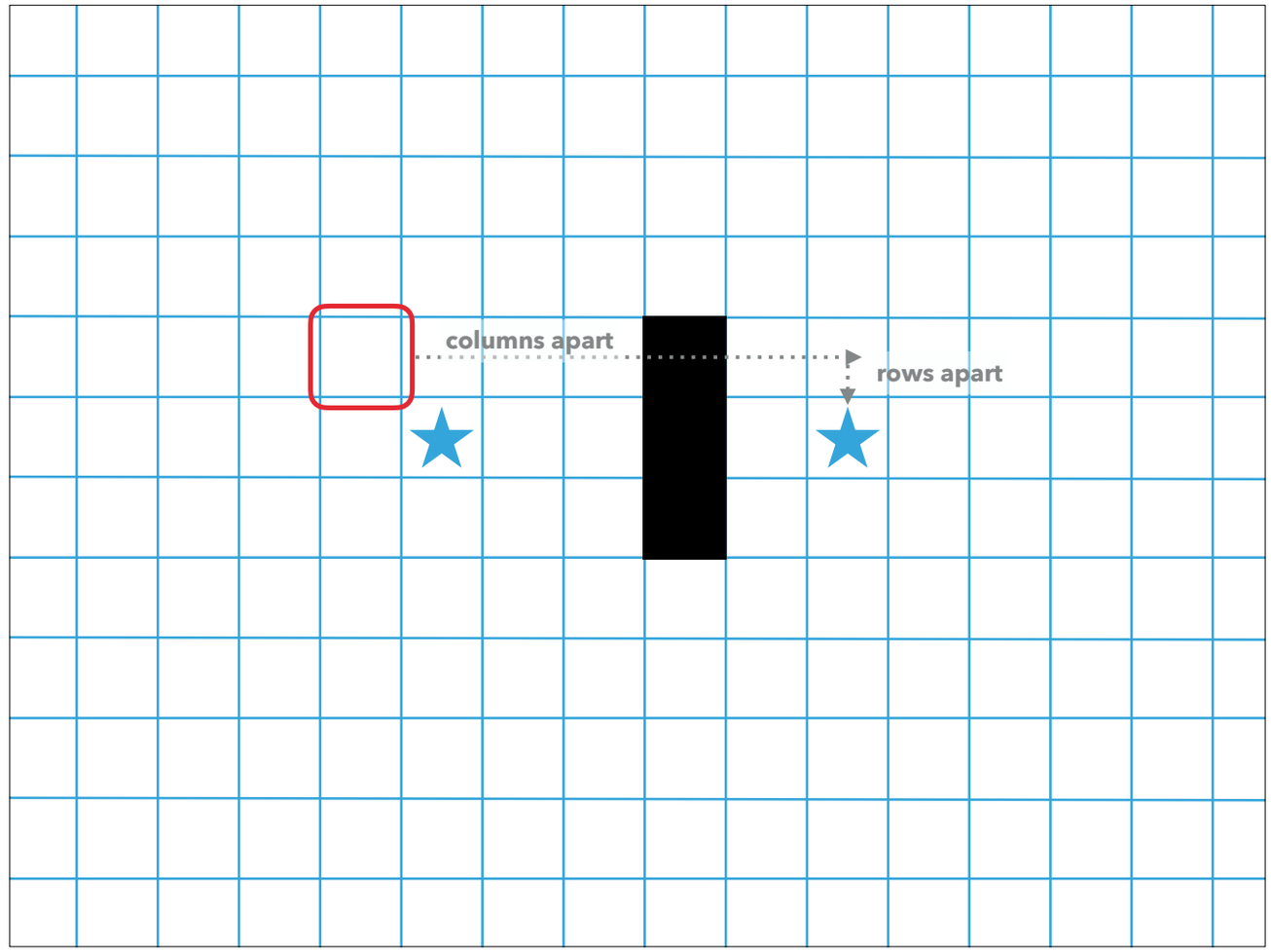
HEURISTICS



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

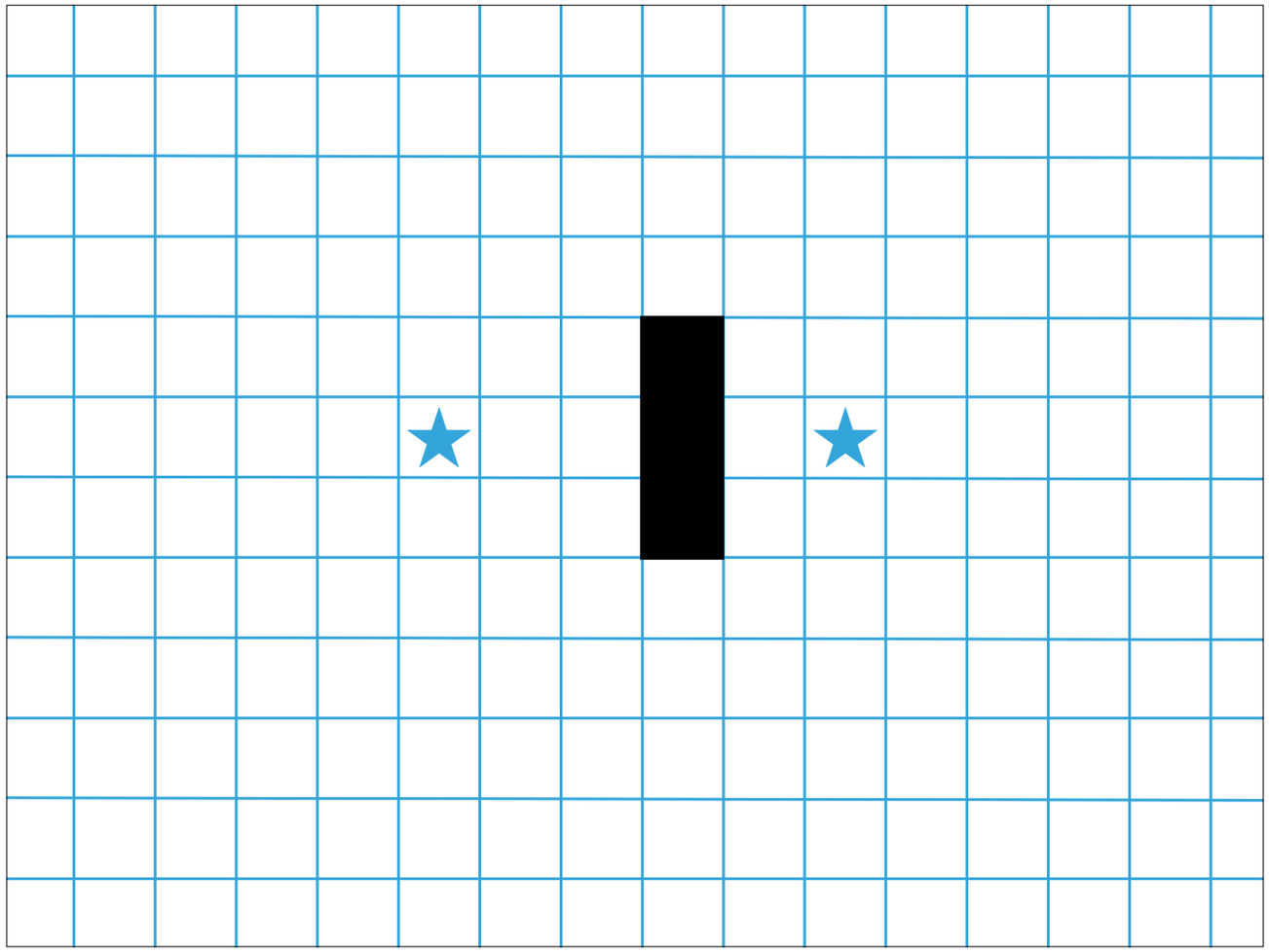
It's a "relaxation" heuristic.

In order for a heuristic to be valid, it needs to **underestimate** the cost of traveling from u to t. It must be less than or equal to the actual future cost. This is known as "relaxing" the constraint.

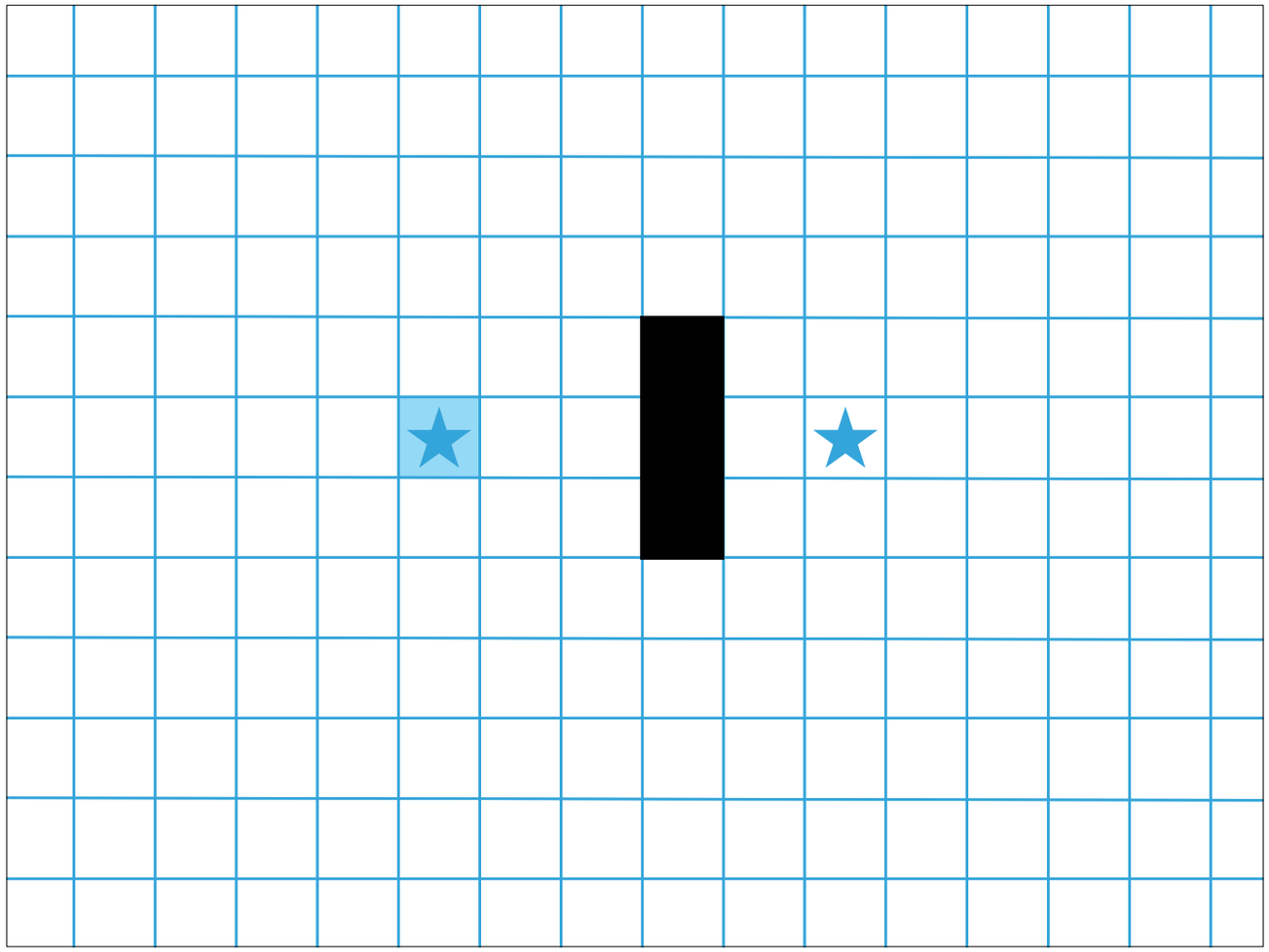


It turns out in this example, doing the manhattan distance is actually a good heuristic. In this example, the heuristic says the cost of traveling from the highlighted node to the destination is 7.

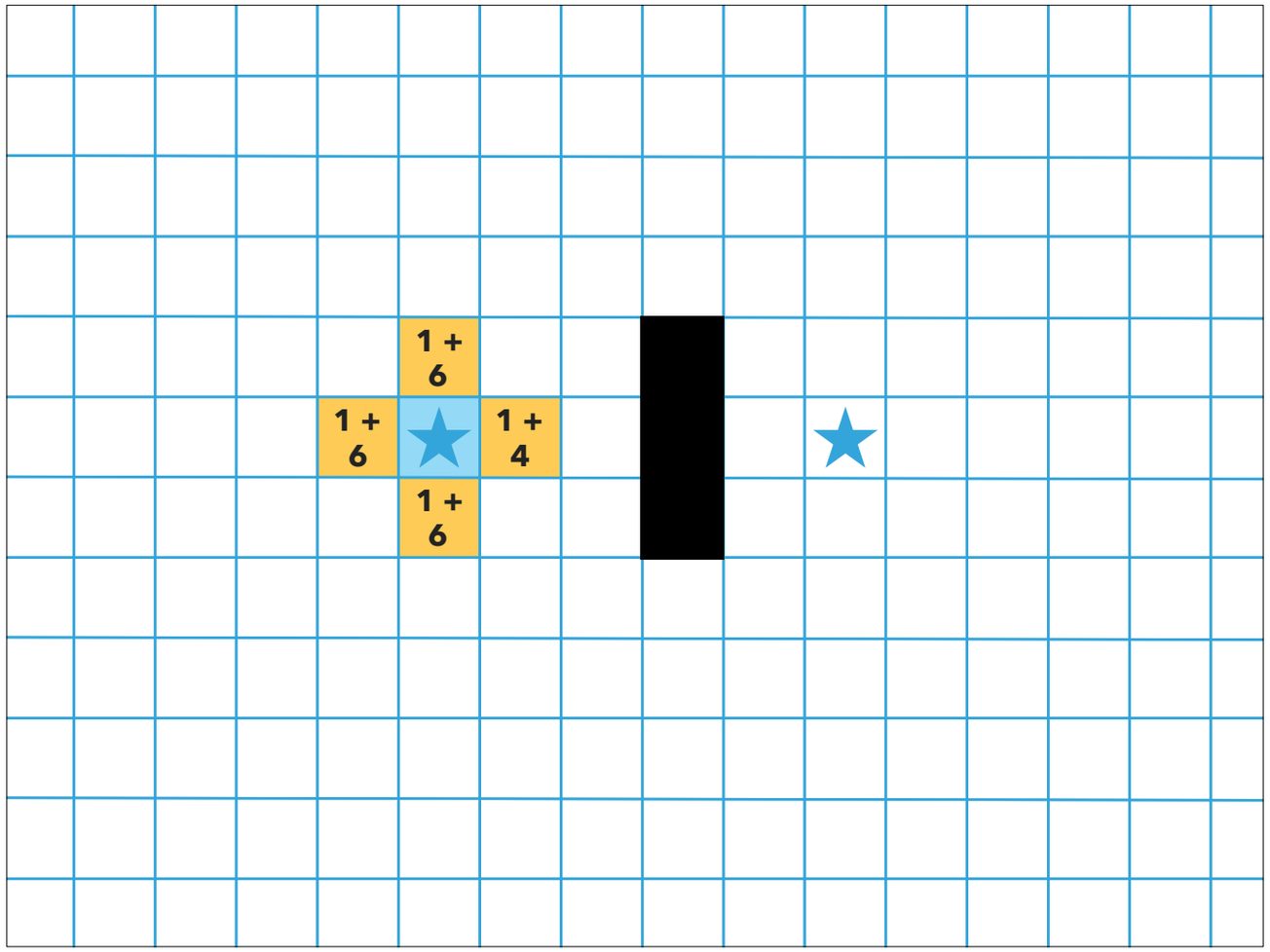
Now, the actual cost is more than that – the shortest path is cost 9. But by including the heuristic value of 7, we tell the algorithm that this is going to be a costly path, which causes it to prioritize it less. But we also make sure that the algorithm doesn't *avoid* this path if it ends up being the one that we want.



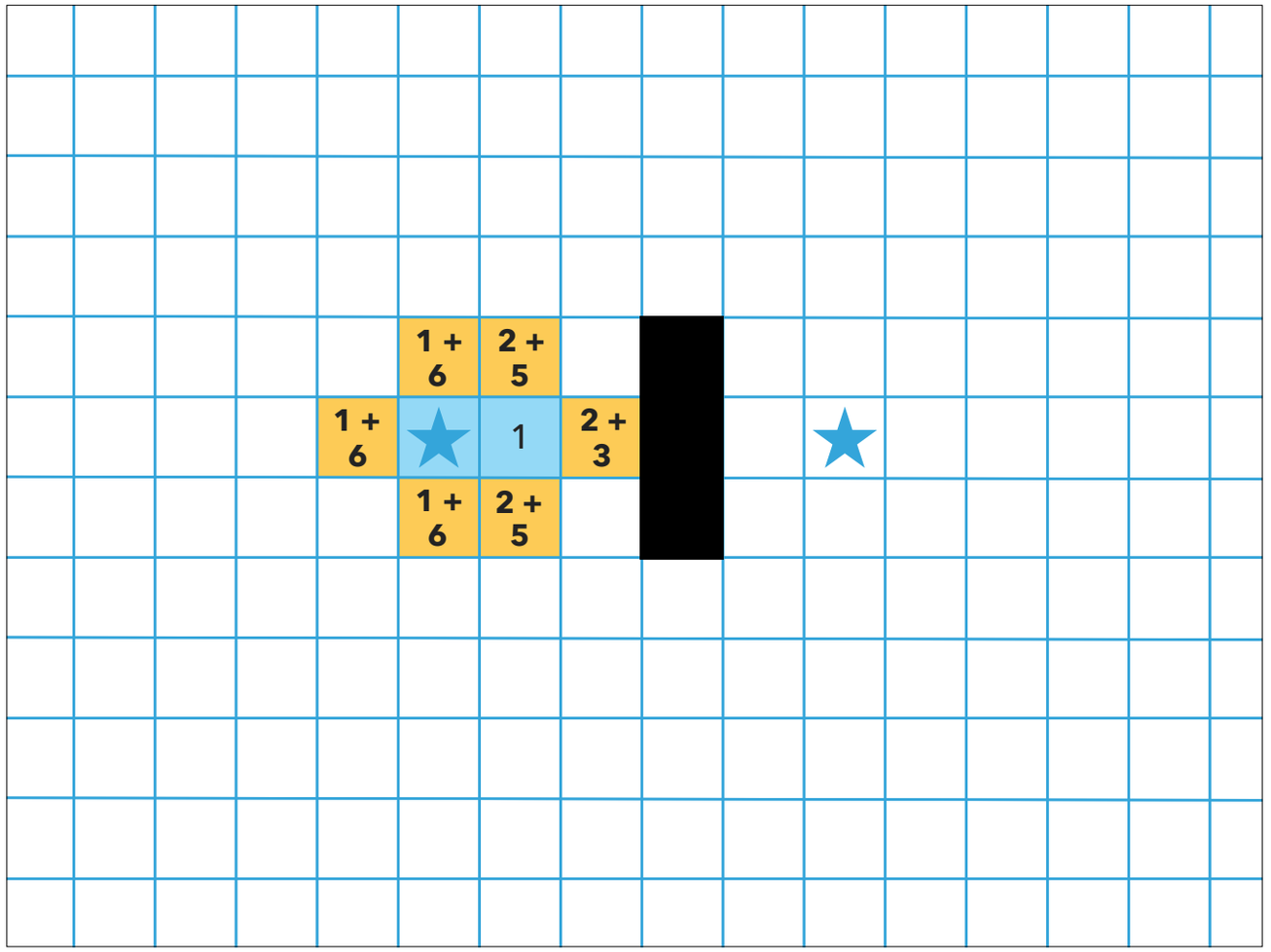
So let's run through this again.



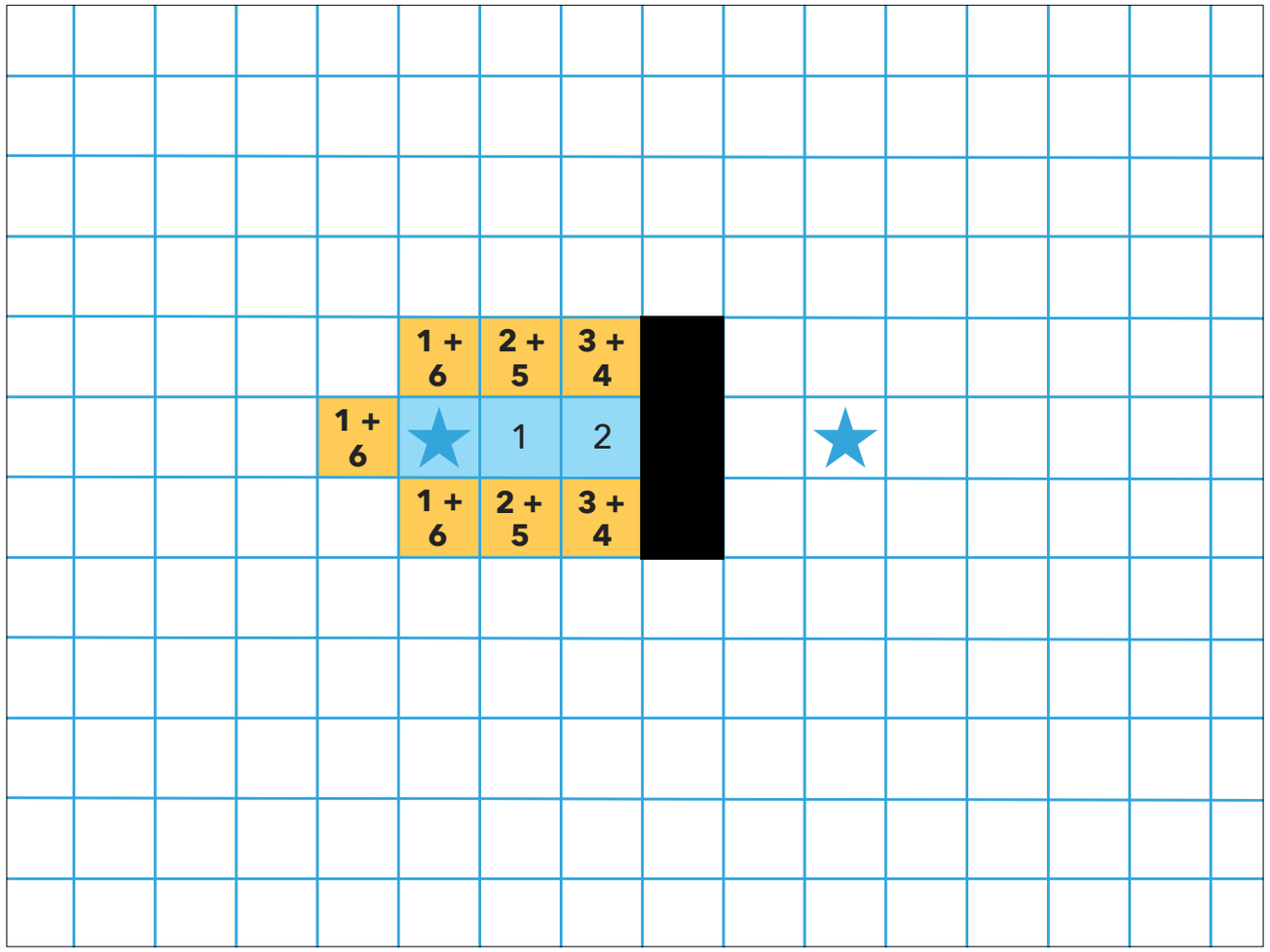
As always, you start by enqueueing the starting node.



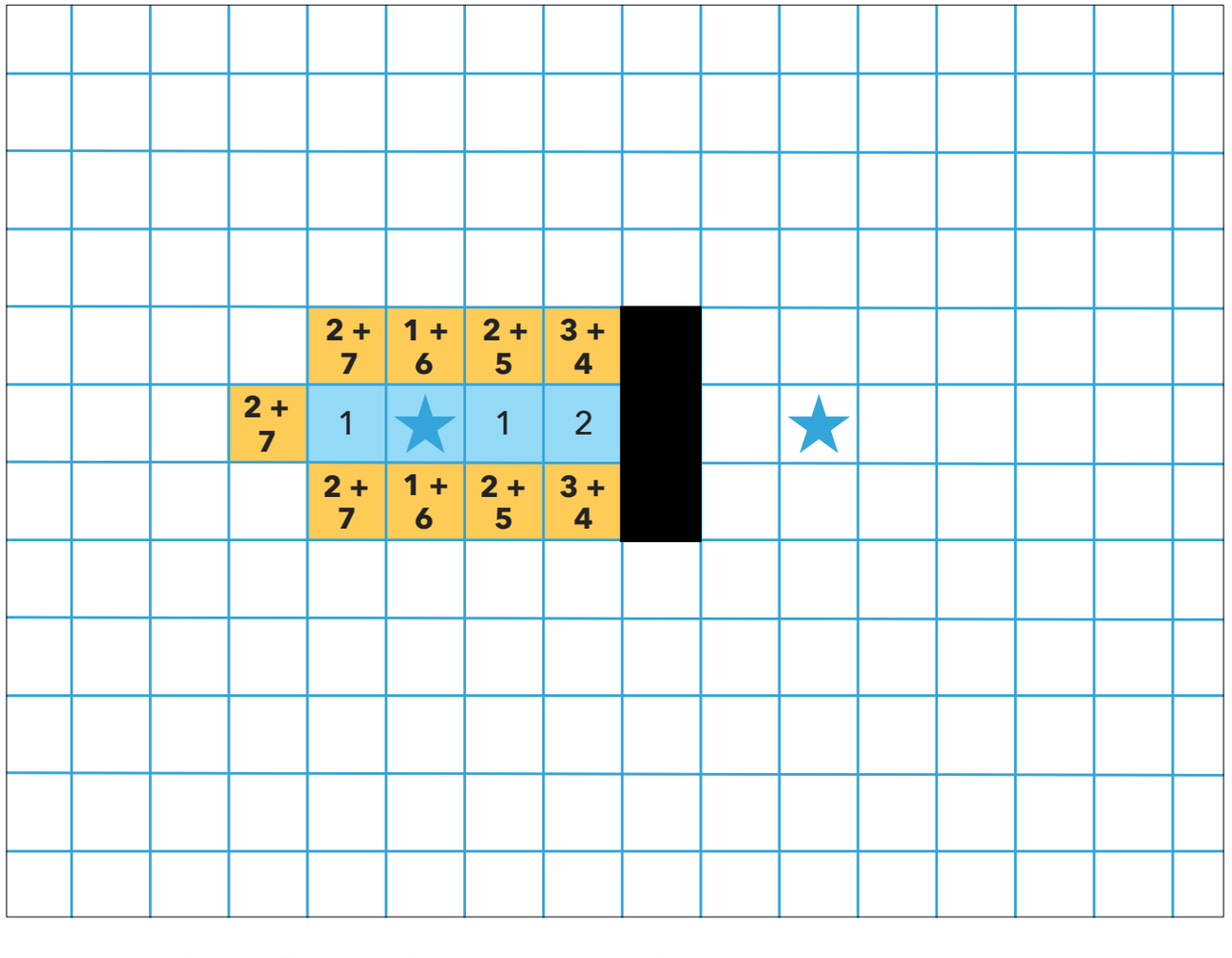
Then you dequeue it, and add its neighbors with weight 1 **plus** the heuristic. Note that the values are the same as they were on the last example.



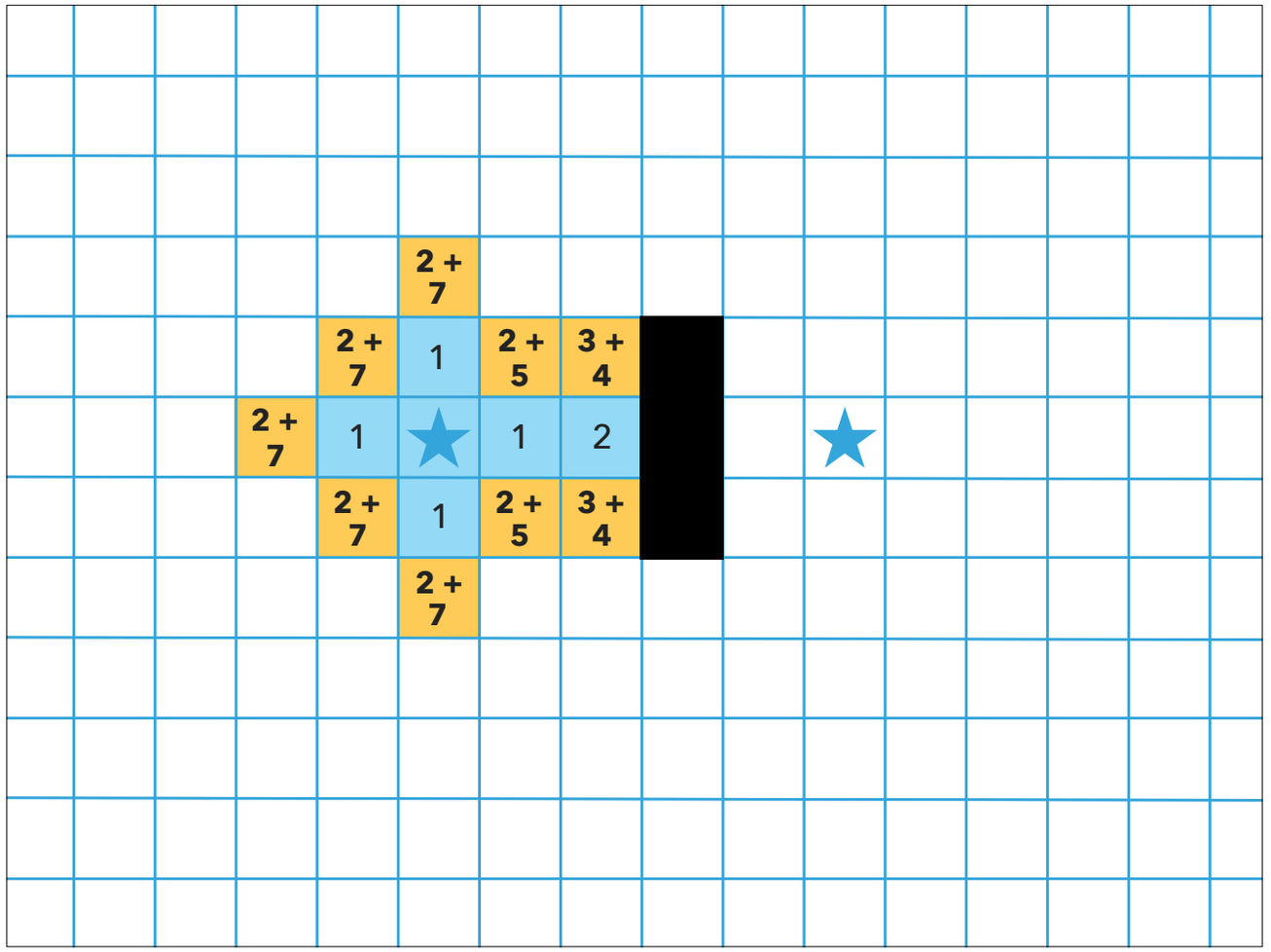
You dequeue the cheapest one, and enqueue its neighbors. And there's still a clearly cheapest choice.

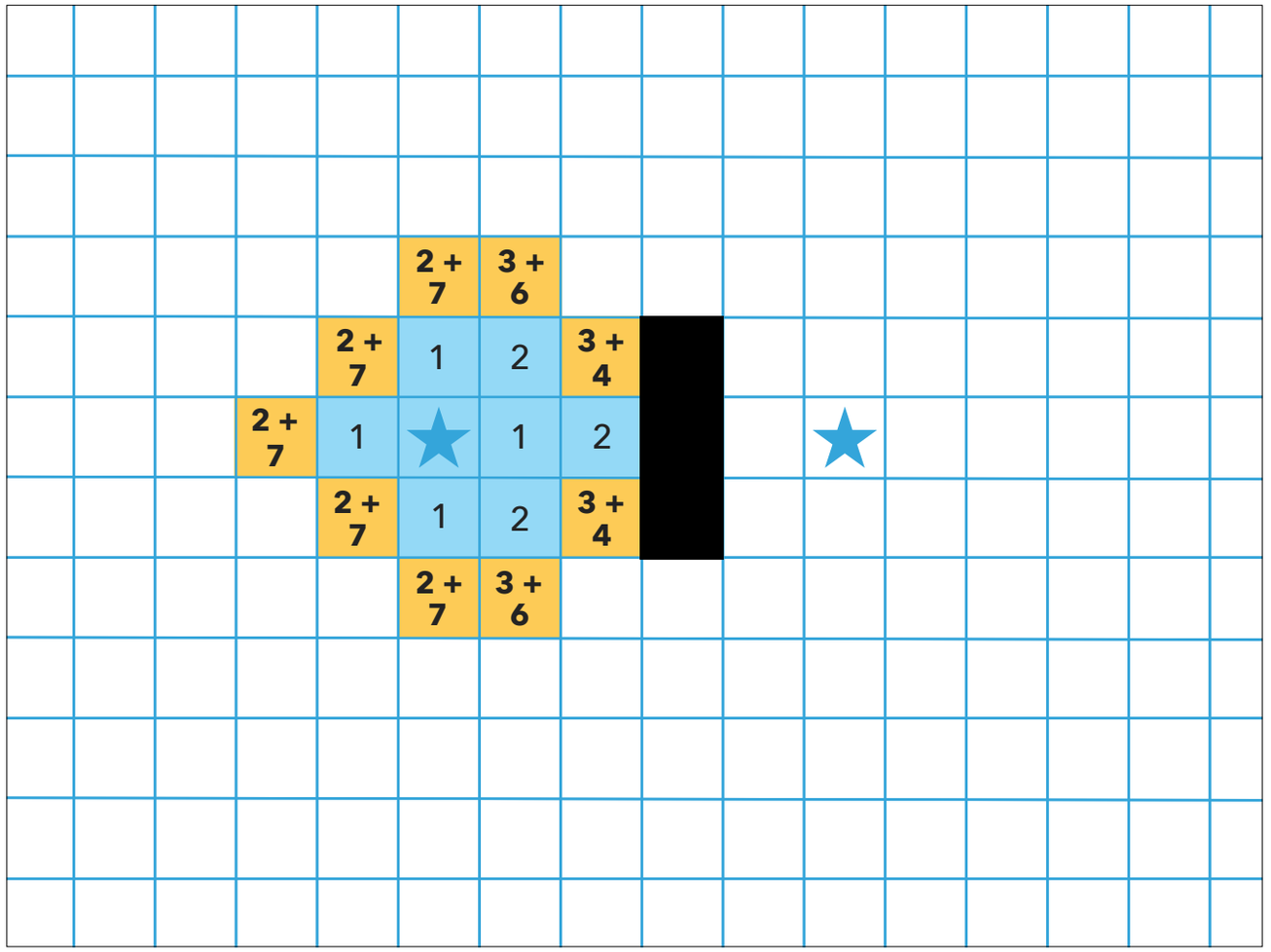


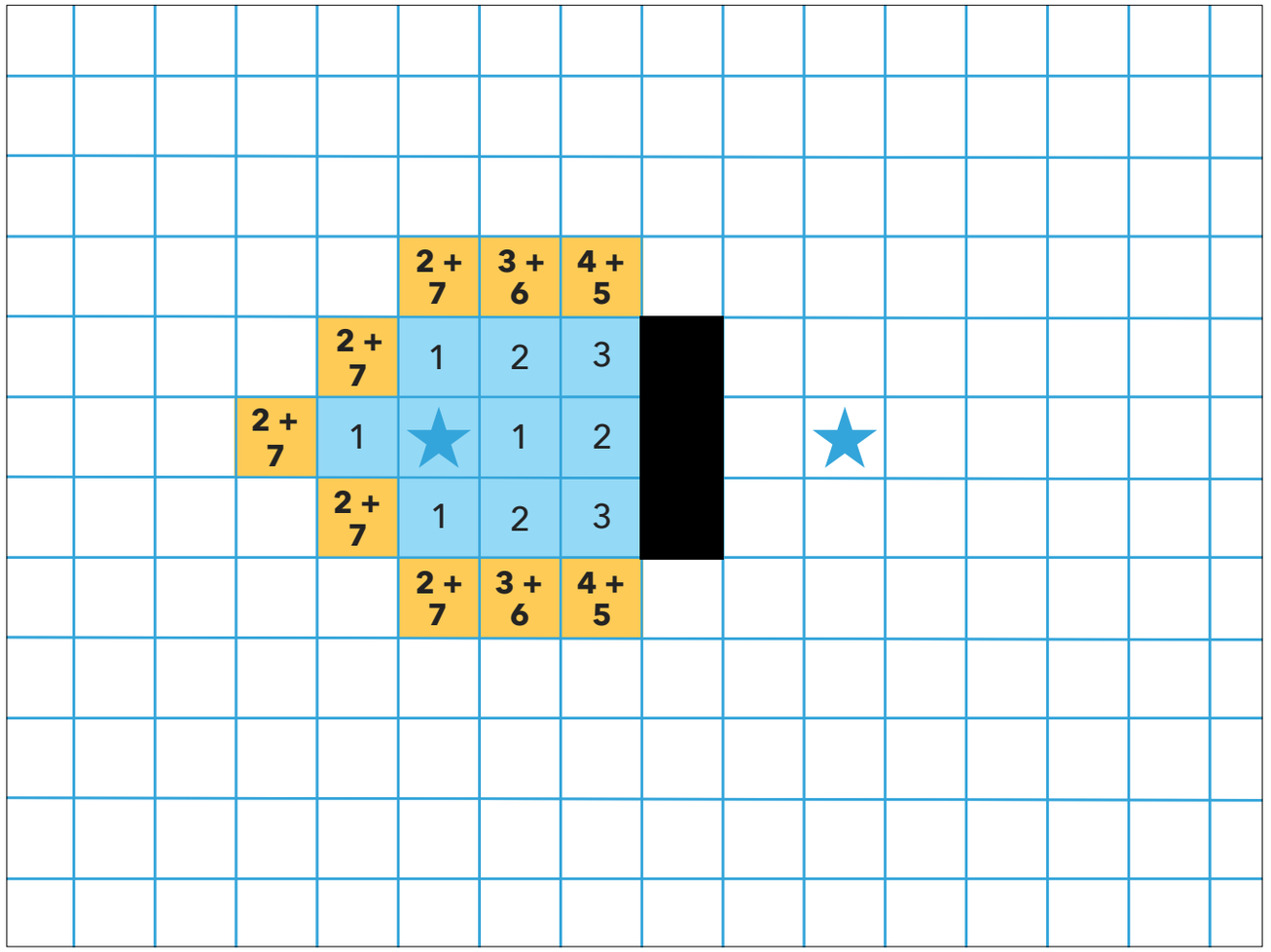
So we dequeue that one and enqueue its neighbors. And now we have a lot of possible nodes to pursue from here. So again, we're going to go from left to right, top to bottom.



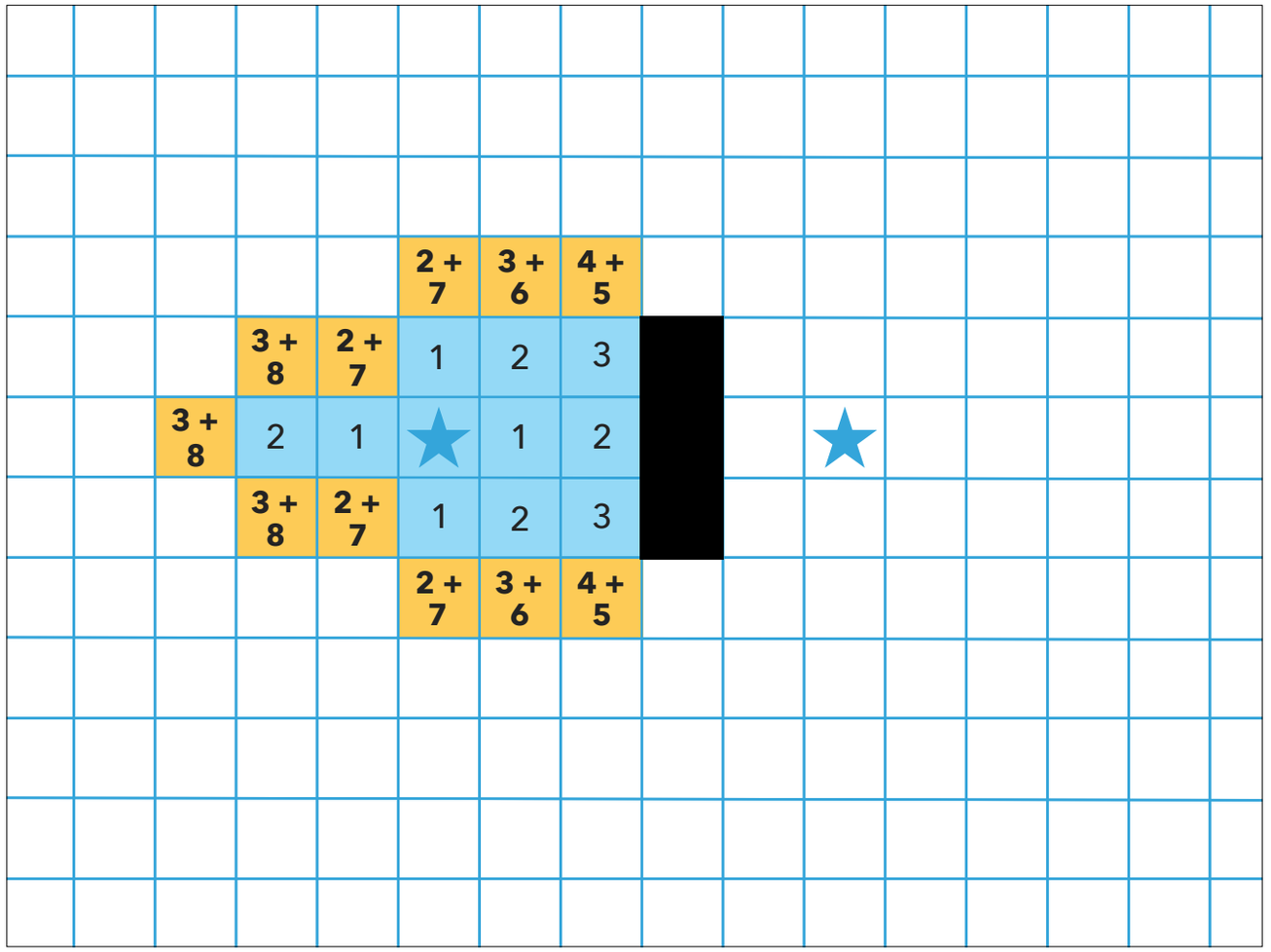
We dequeue the left most node, and enqueue its neighbors. Because it's neighbors are further away from the destination though, the heuristic is going to give it a fairly high cost. So we're going to keep going to the right from there.

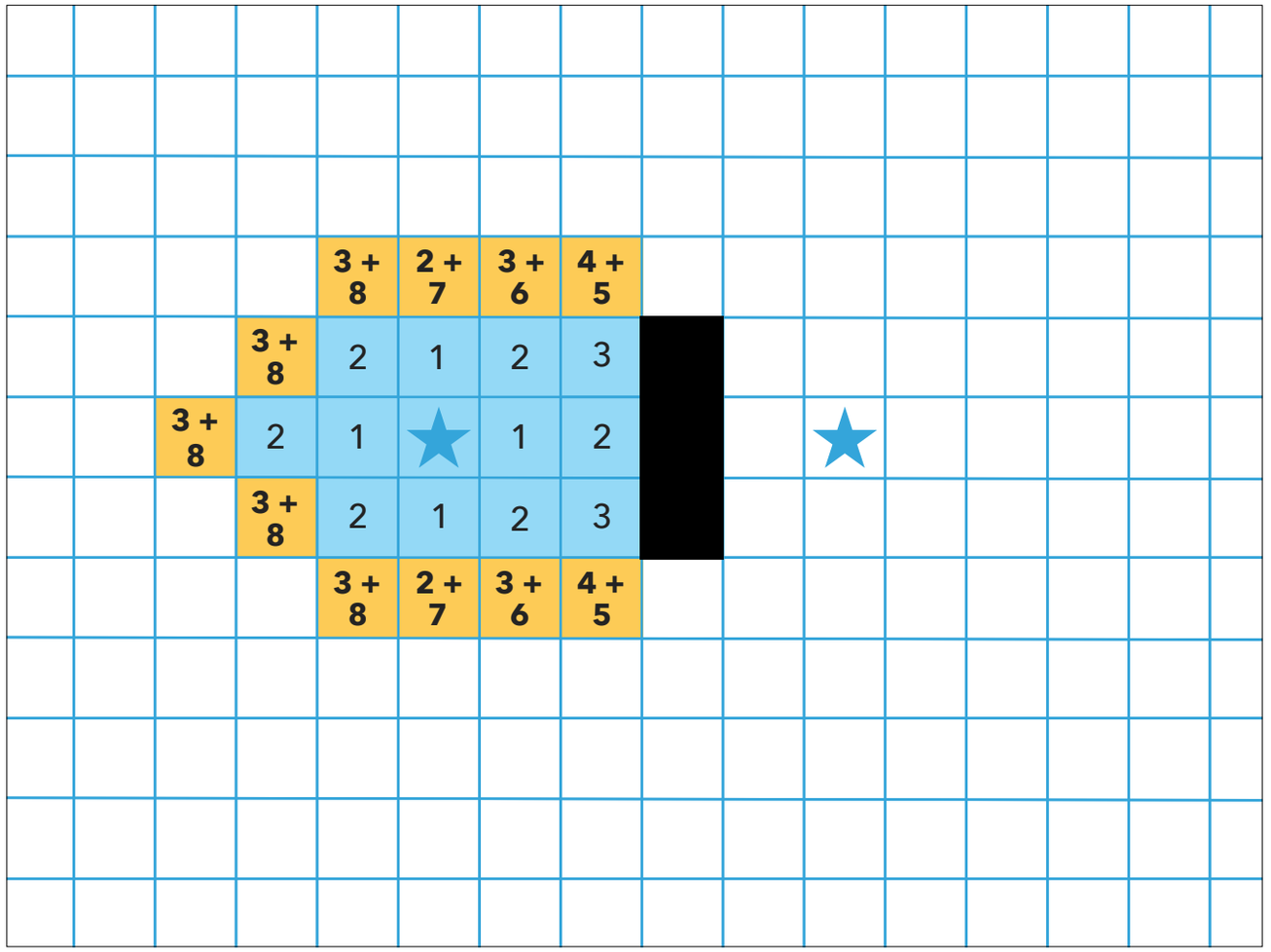

2 + 7
2 + 7
1
2 + 5
3 + 4
2 + 7
1
1
2
2 + 7
1
2 + 5
3 + 4
2 + 7

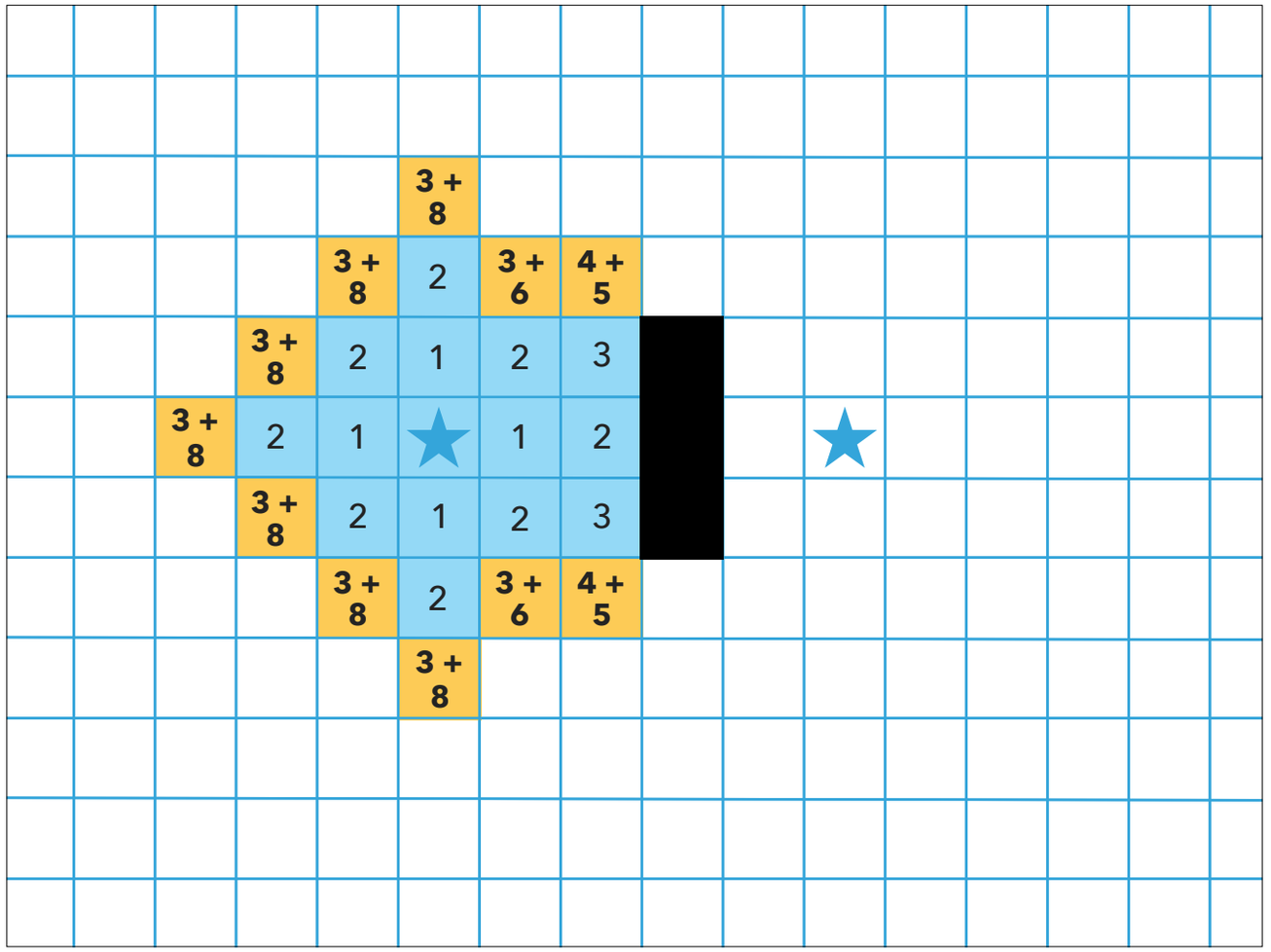

2 + 7
3 + 6
2 + 7
1
2
3 + 4
2 + 7
1
1
2
2 + 7
1
2
3 + 4
2 + 7
3 + 6

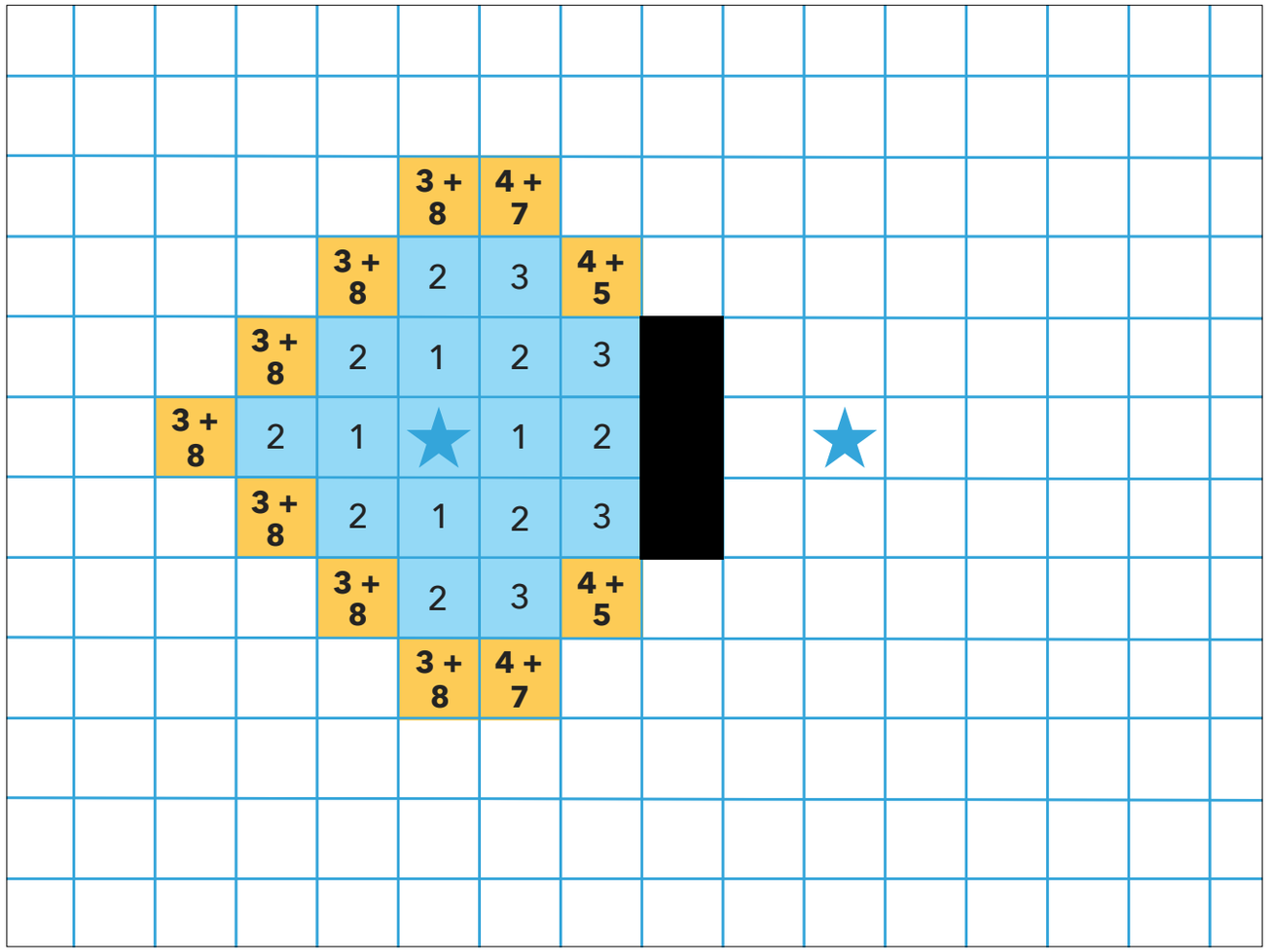


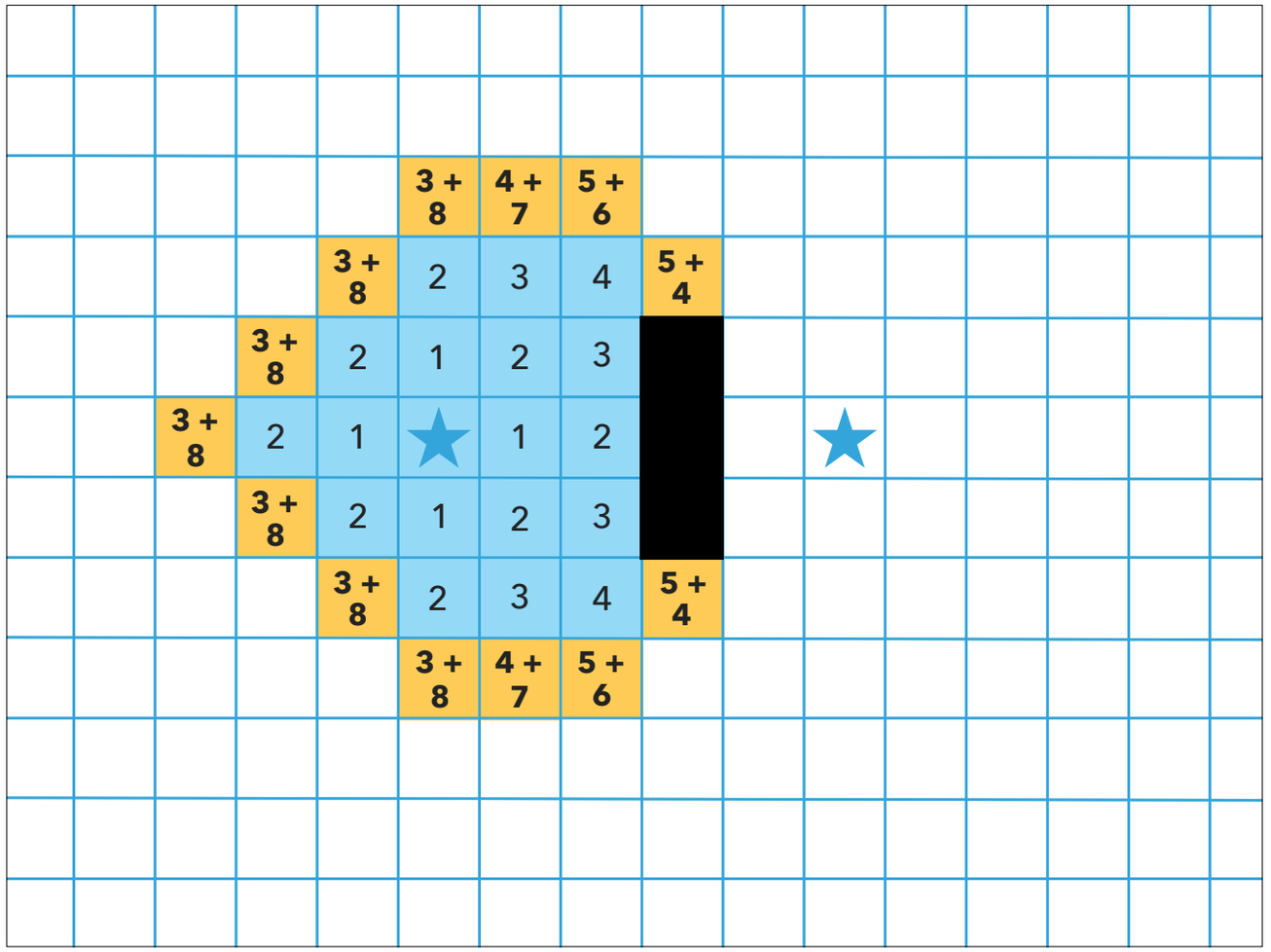
At this point, we again end up with a lot of nodes of similar costs. So we have to go through and consider all of them.

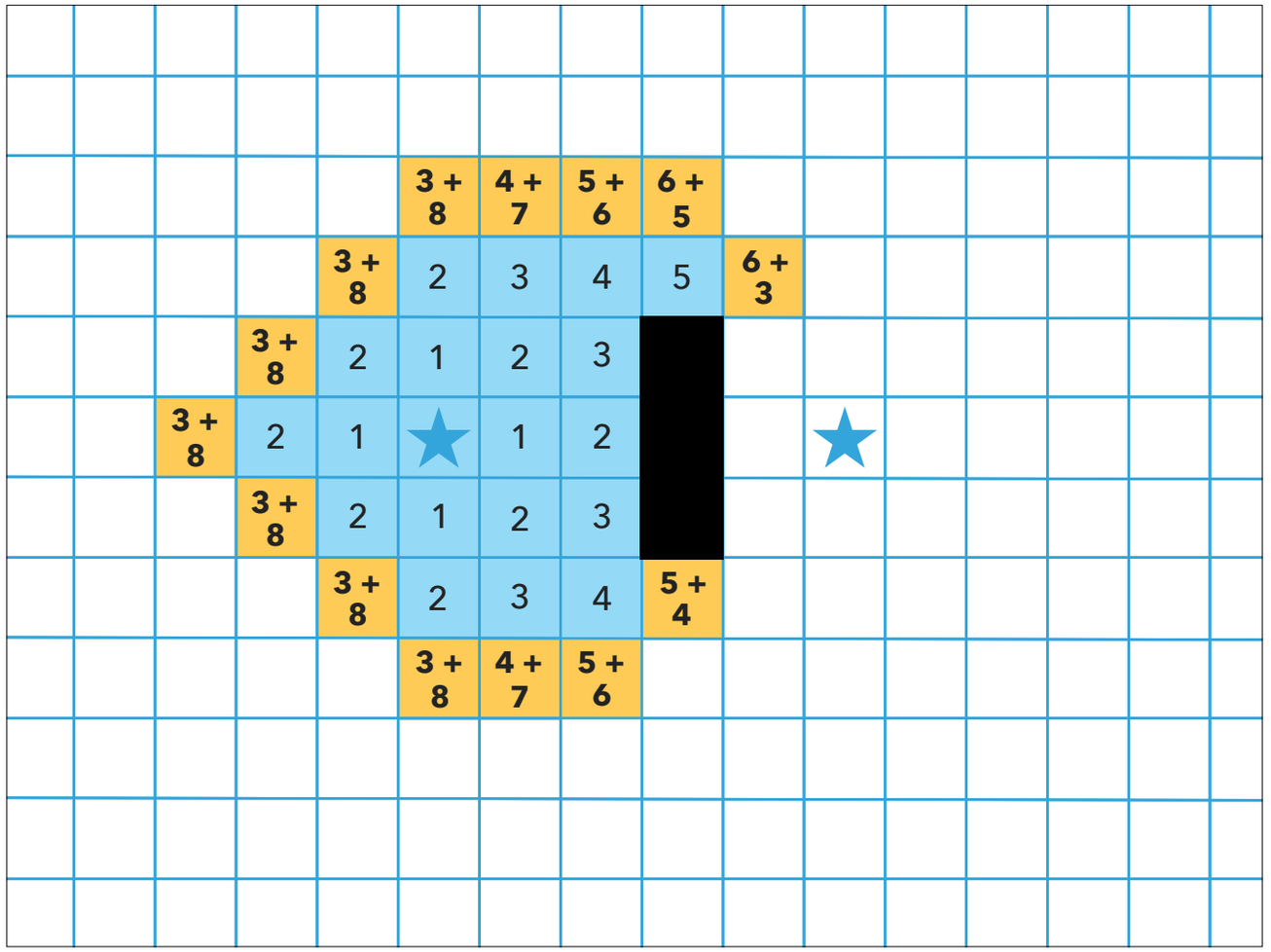

2 + 7
3 + 6
4 + 5
3 + 8
2 + 7
1
2
3
3 + 8
2
1
1
2
3 + 8
2 + 7
1
2
3
2 + 7
3 + 6
4 + 5


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

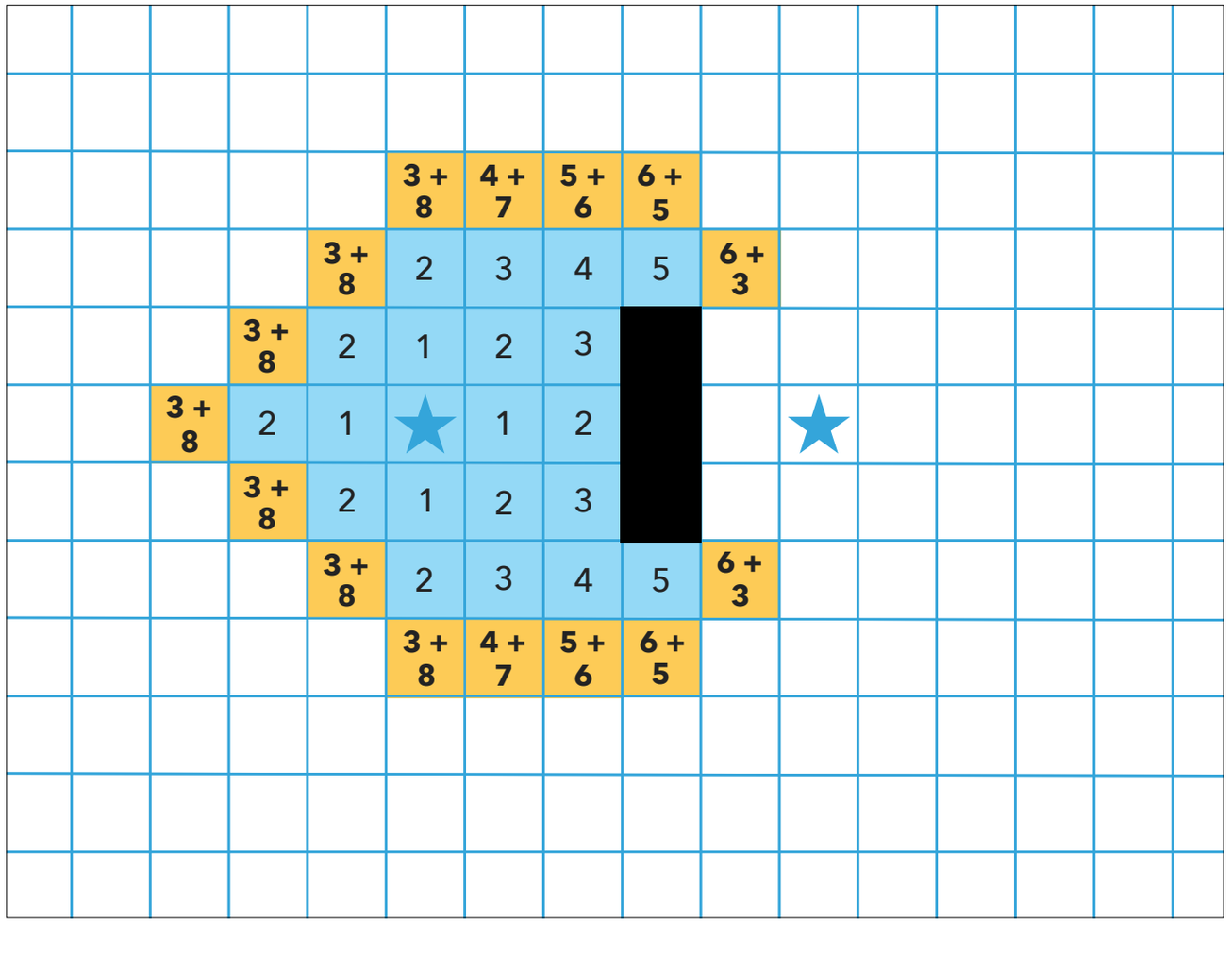

3 + 8
3 + 8
2
3 + 6
4 + 5
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3 + 6
4 + 5
3 + 8

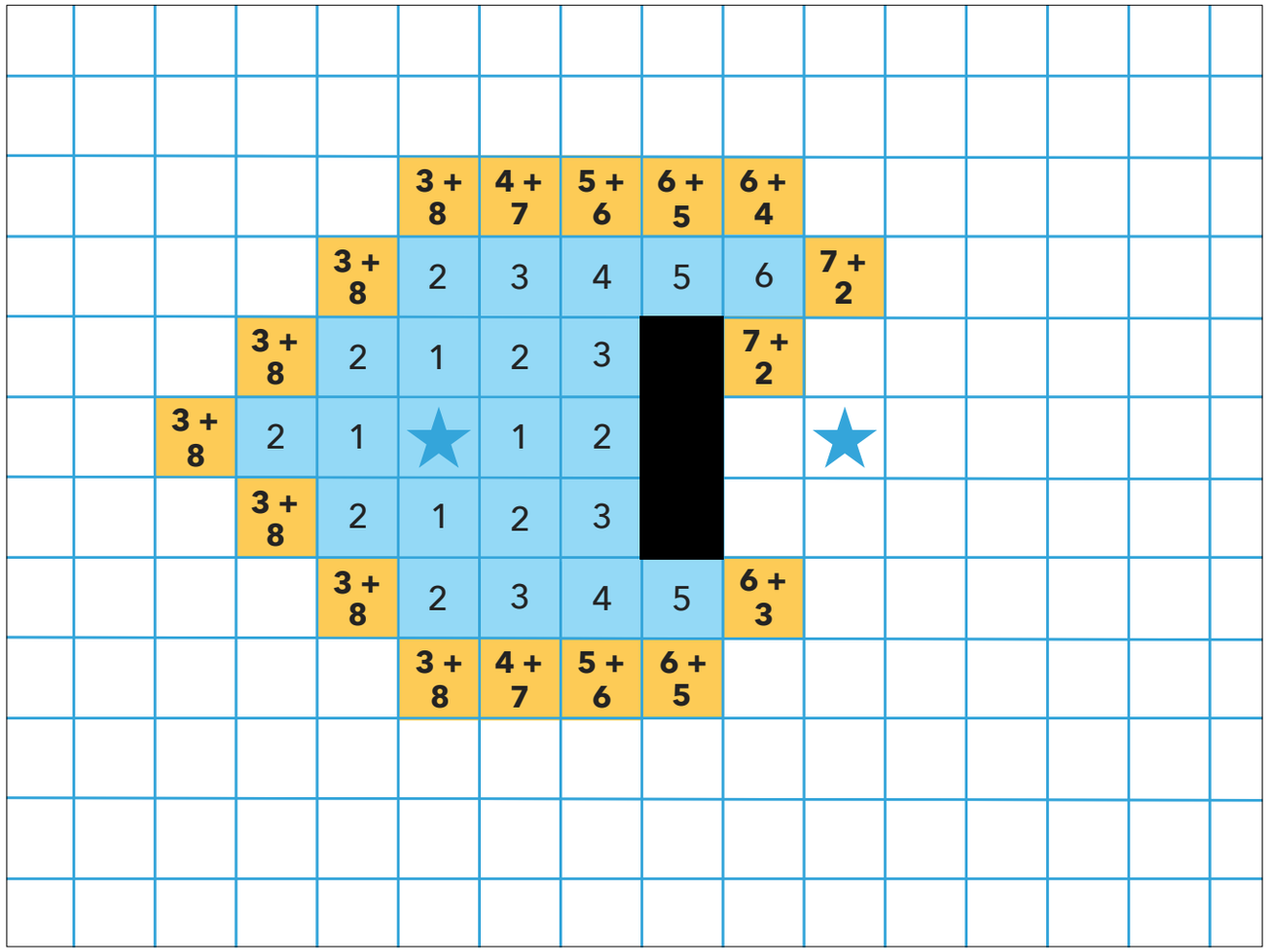

3 + 8
4 + 7
3 + 8
2
3
4 + 5
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3
4 + 5
3 + 8
4 + 7

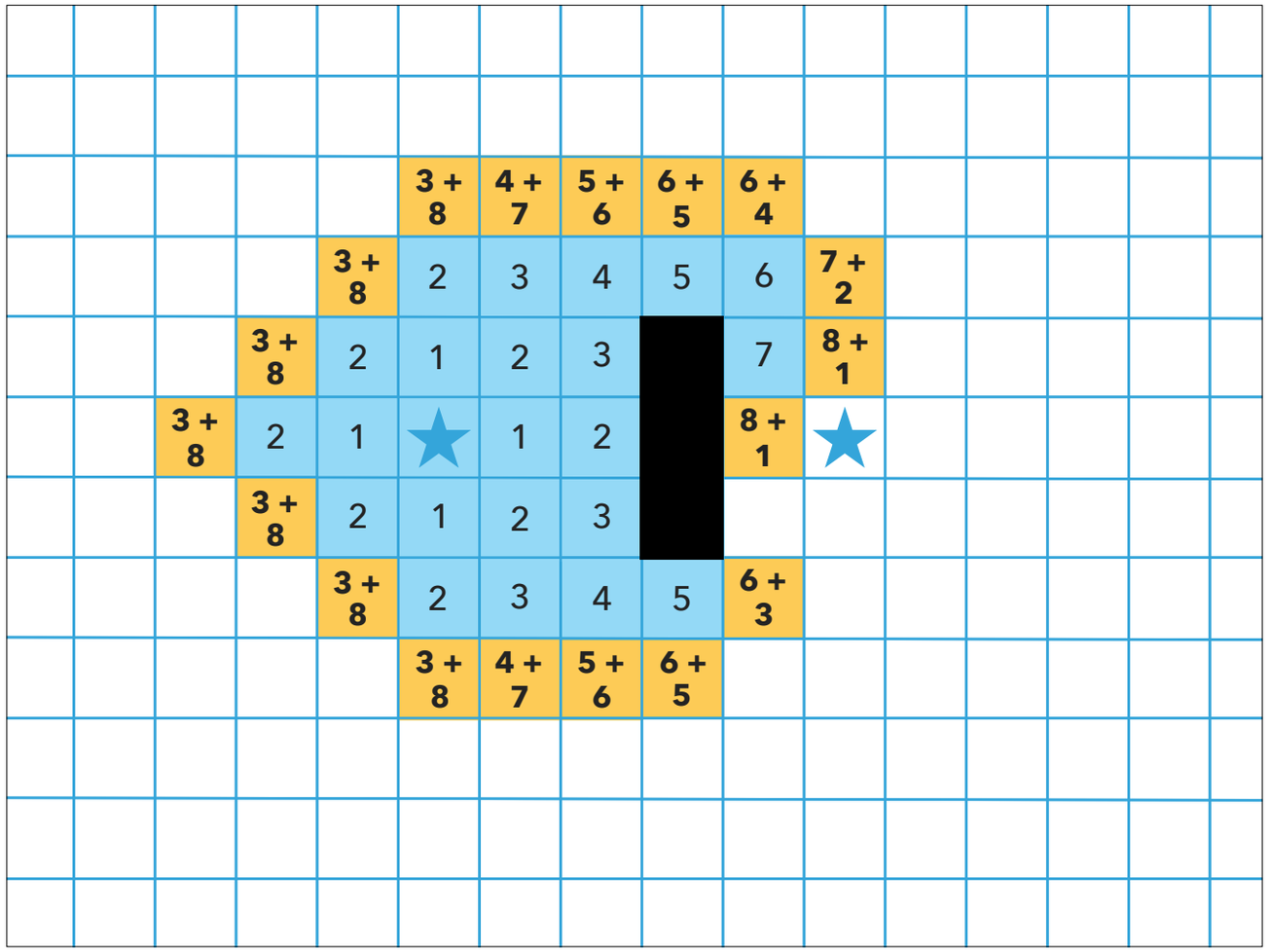

3 + 8
4 + 7
5 + 6
3 + 8
2
3
4
5 + 4
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3
4
5 + 4
3 + 8
4 + 7
5 + 6

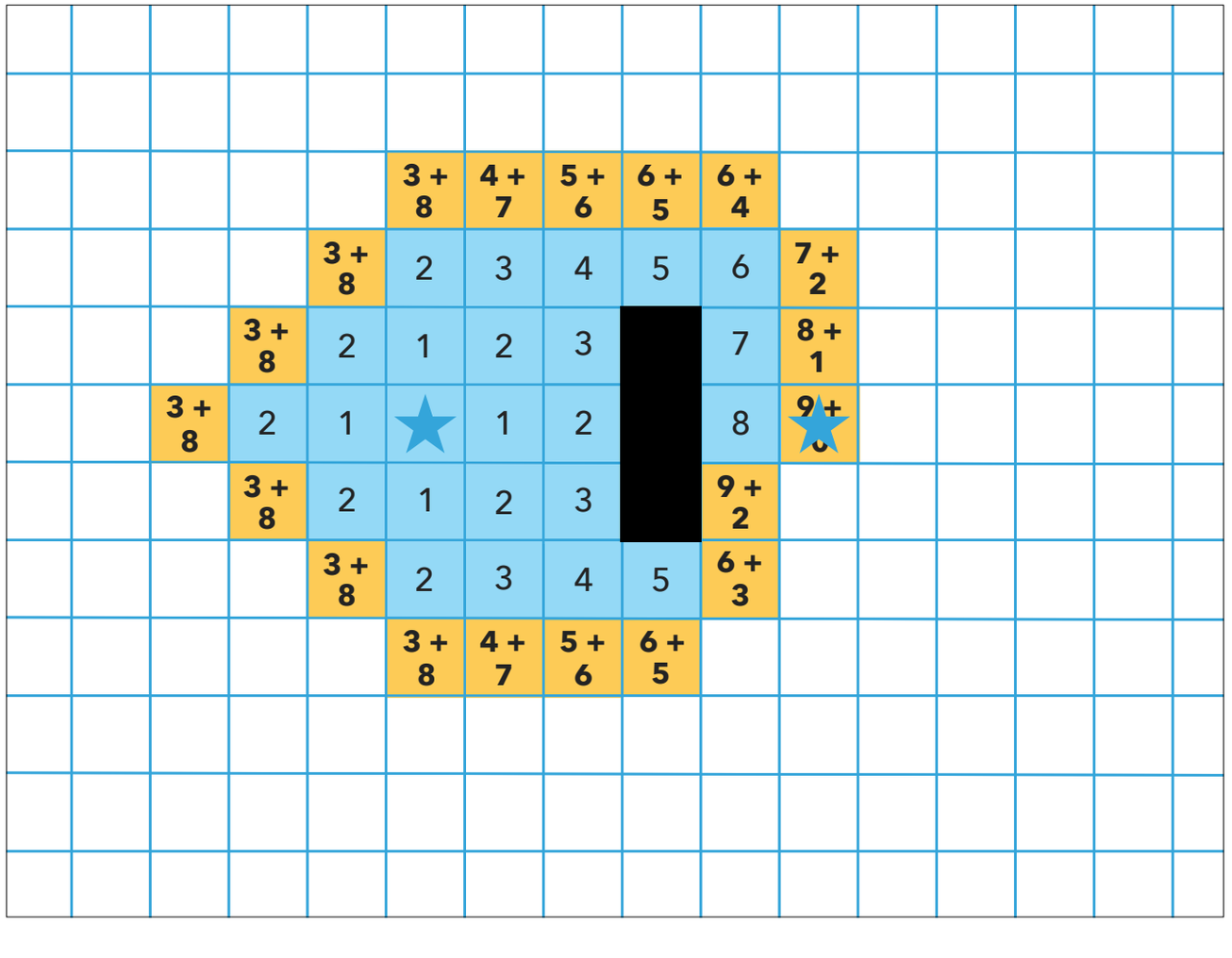


And now we're starting to branch off to the right around the wall! So let's go a few steps further.

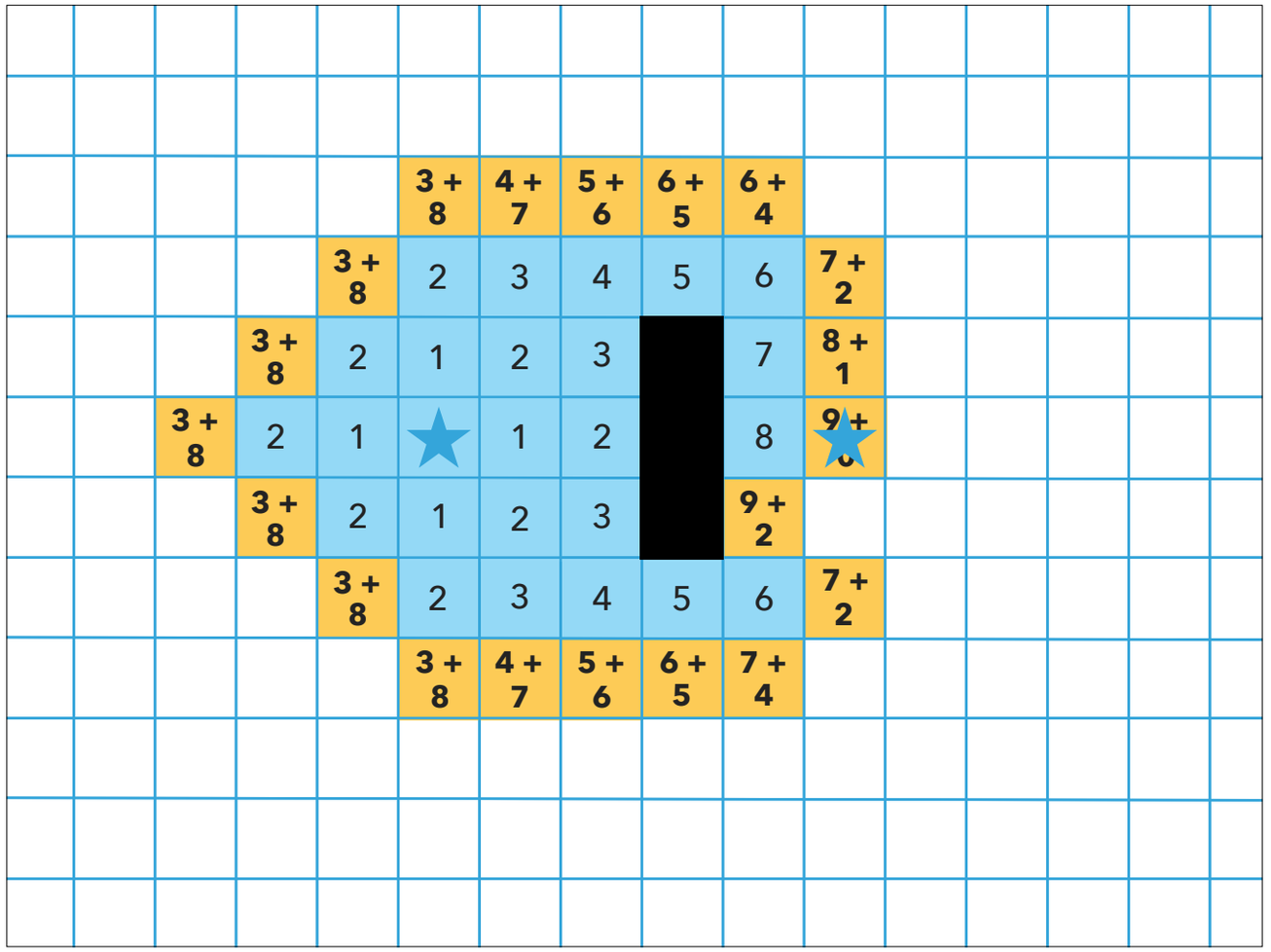

3 + 8
4 + 7
5 + 6
6 + 5
3 + 8
2
3
4
5
6 + 3
3 + 8
2
1
2
3
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3
4
5
6 + 3
3 + 8
4 + 7
5 + 6
6 + 5

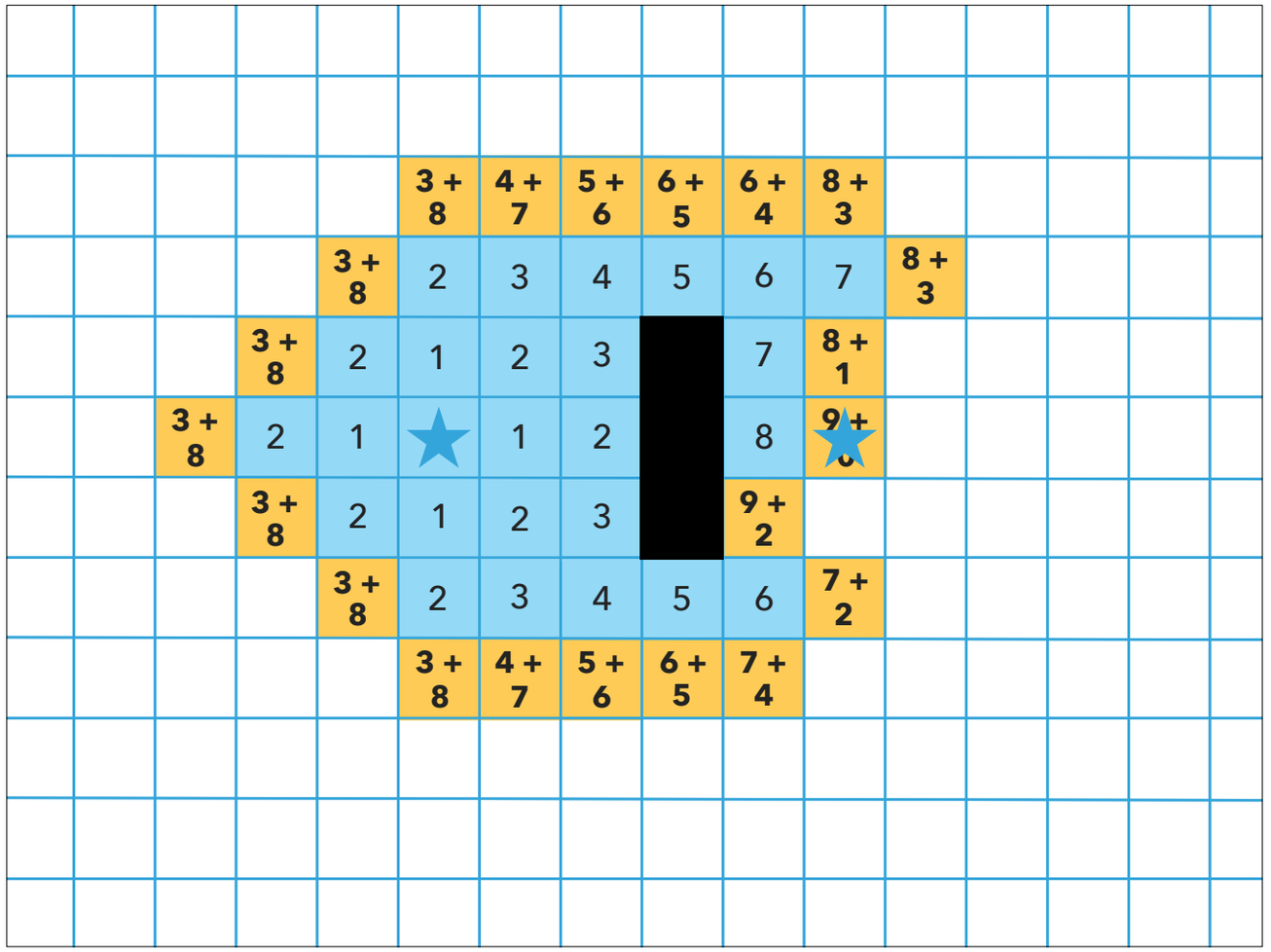

3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
3 + 8
2
3
4
5
6
7 + 2
3 + 8
2
1
2
3
7 + 2
3 + 8
2
1
1
2
3 + 8
2
1
2
3
3 + 8
2
3
4
5
6 + 3
3 + 8
4 + 7
5 + 6
6 + 5

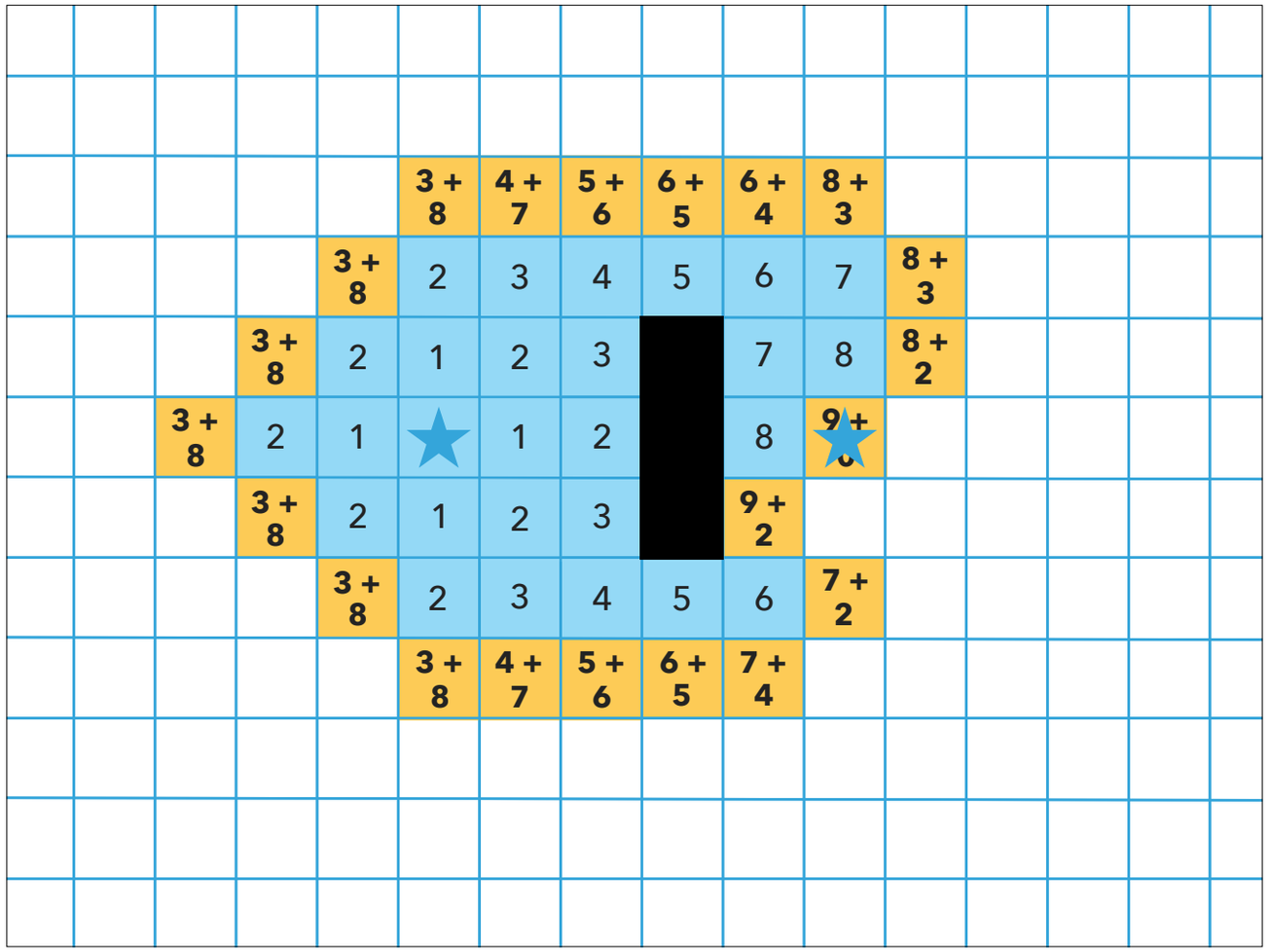

3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
3 + 8
2
3
4
5
6
7 + 2
3 + 8
2
1
2
3
7
8 + 1
3 + 8
2
1
1
2
8 + 1
3 + 8
2
1
2
3
3 + 8
2
3
4
5
6 + 3
3 + 8
4 + 7
5 + 6
6 + 5

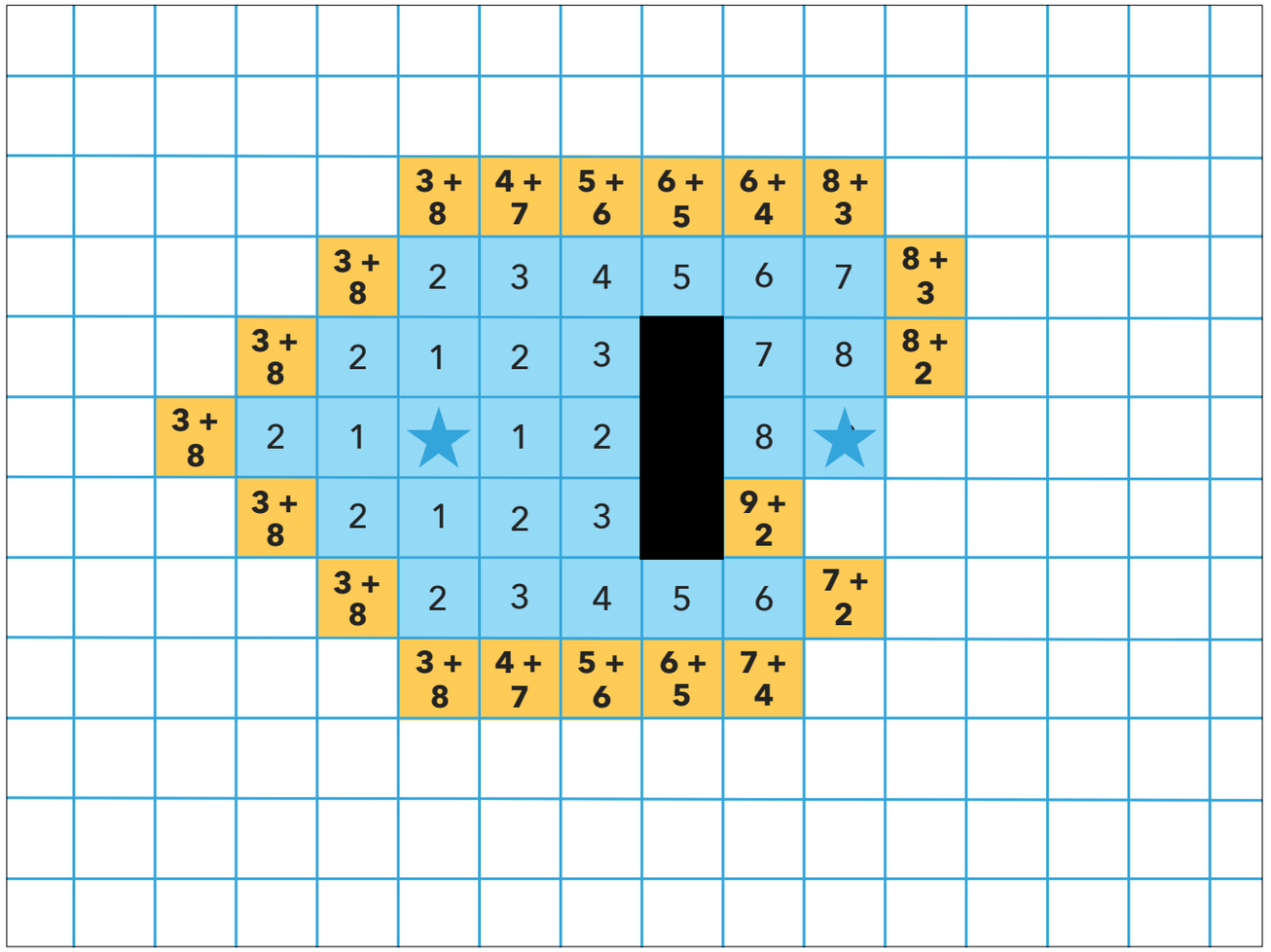


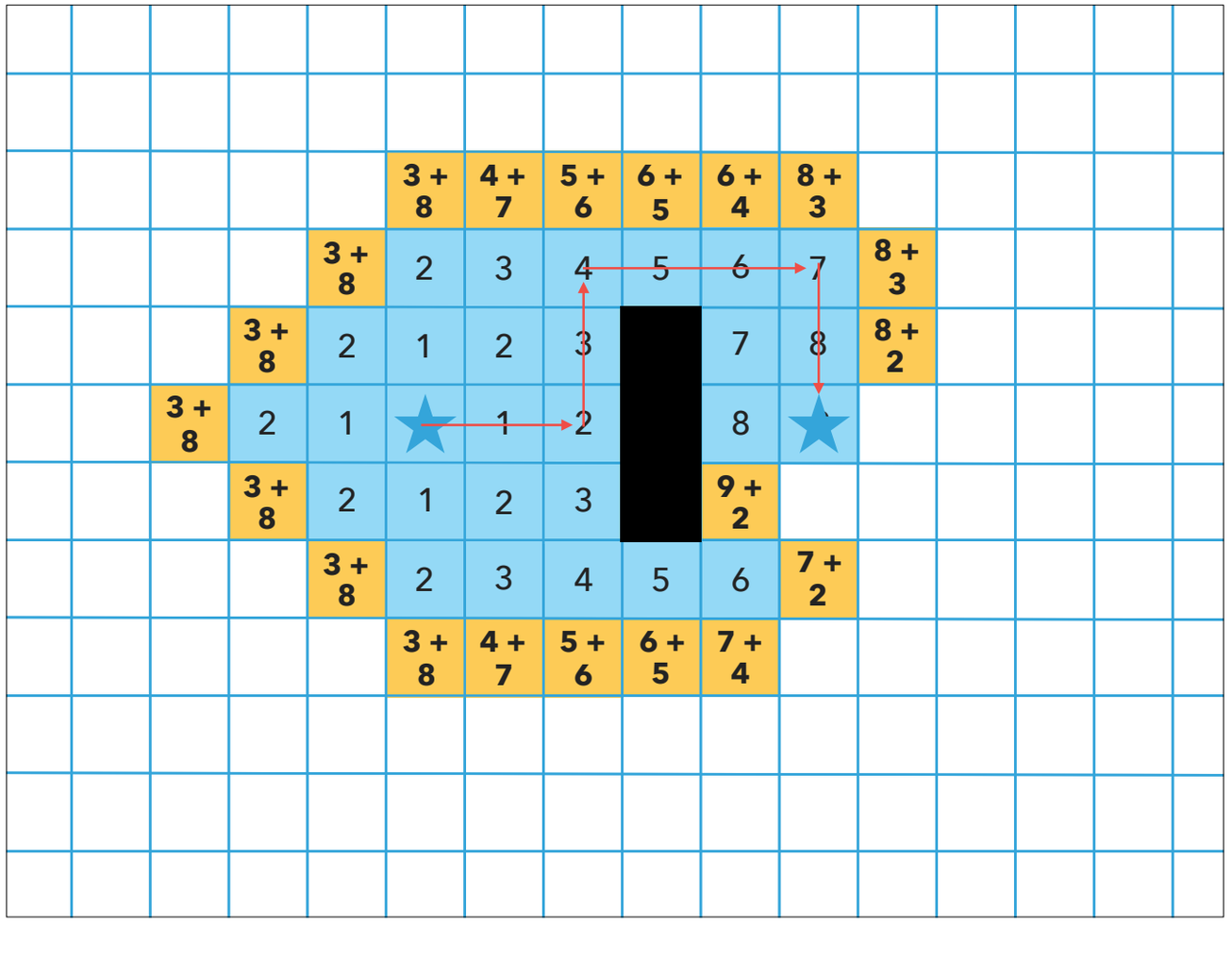
Again! We aren't done yet!


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

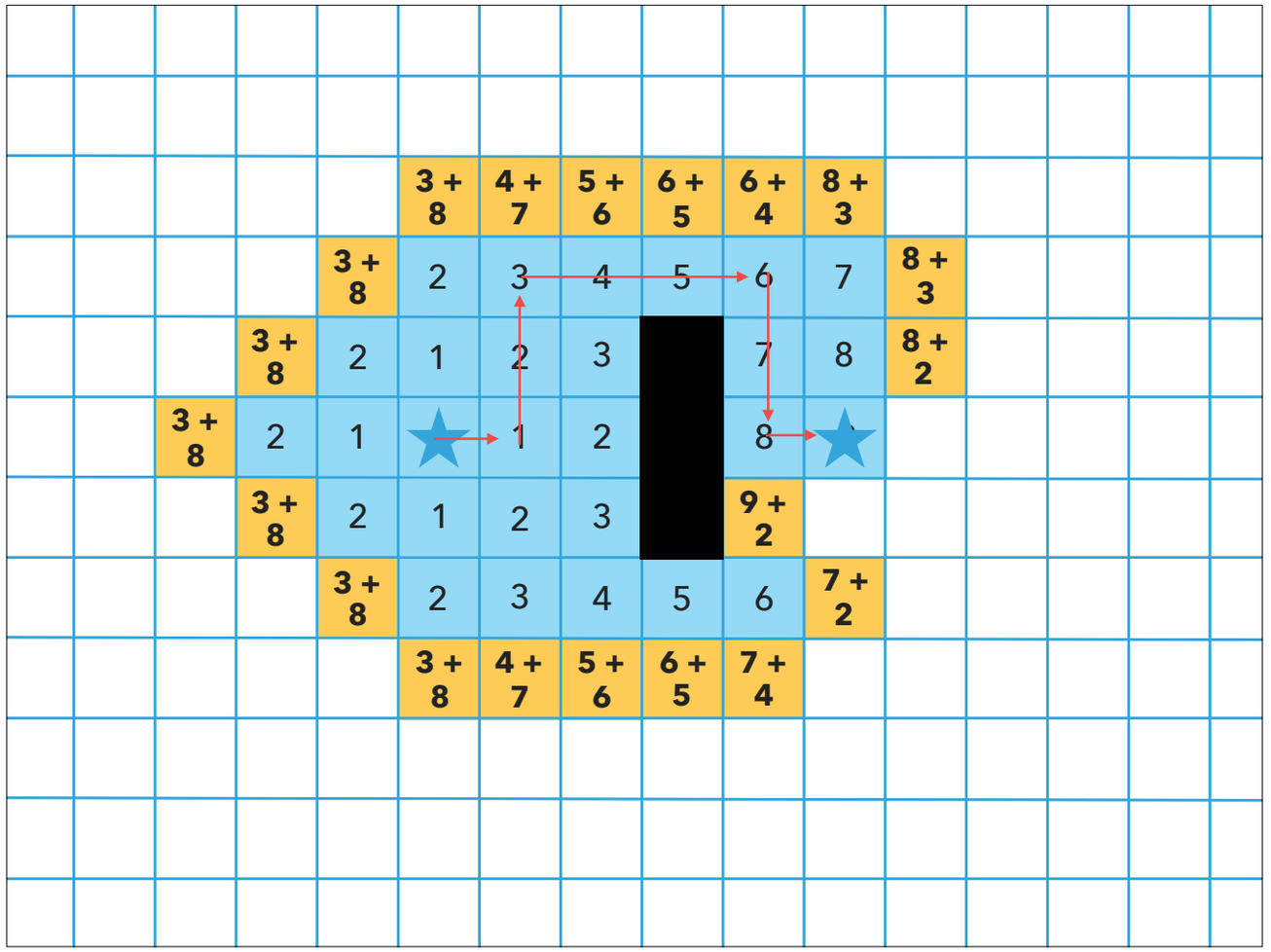

3 + 8
4 + 7
5 + 6
6 + 5
6 + 4
8 + 3
3 + 8
2
3
4
5
6
7
8 + 3
3 + 8
2
1
2
3
7
8 + 1
3 + 8
2
1
1
2
8
3 + 8
2
1
2
3
9 + 2
3 + 8
2
3
4
5
6
7 + 2
3 + 8
4 + 7
5 + 6
6 + 5
7 + 4


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

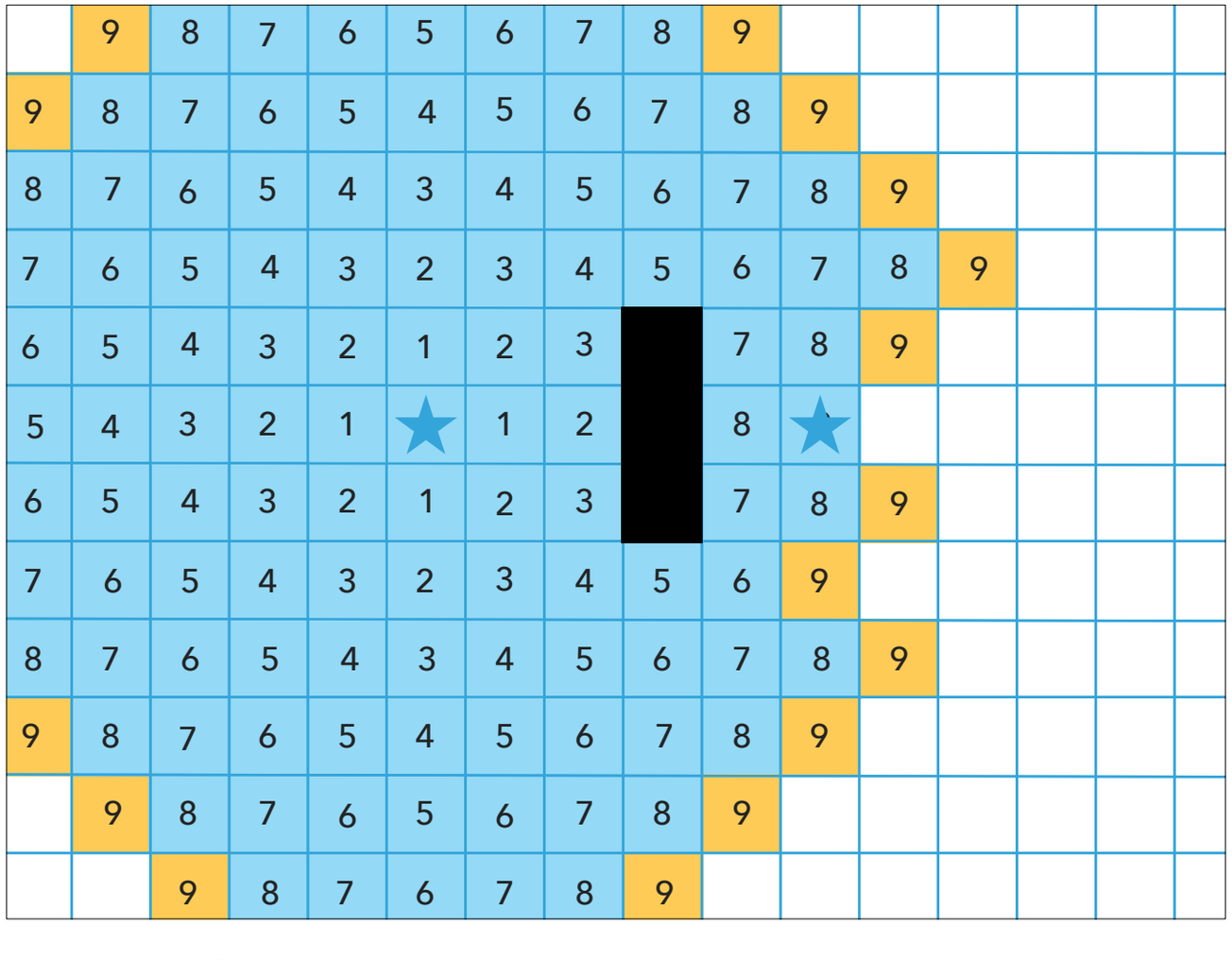


And we found a path!



Multiple paths, actually, all with the same cost.

Now something I want to point out is that A* is not perfect – there was still some wasted search that we didn't have when we could actually see into the future. But even if A* isn't perfect...



It's more perfect than Dijkstra. This is the search space Dijkstra would have explored before finding a path around the wall.

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

So I'm going to quickly throw up the pseudocode for A* here on the screen. And you can see, it still follows the same basic todo list structure that I showed you earlier.

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

In fact, look at what happens when you compare it to the pseudocode for Dijkstra we saw earlier. Does anyone see any similarities between these?

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**

Exactly! The only difference is the heuristic. So my next question is... is 0 a valid 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**

It is! And if you do that, then Dijkstra and A* are actually the same algorithm. This is why you have to be careful when picking your heuristic for A*. And I'll talk a little bit about how some heuristics are chosen in the real world in just a minute, but the thing to keep in mind is:

YOU WANT YOUR HEURISTIC TO BE AS LARGE AS POSSIBLE

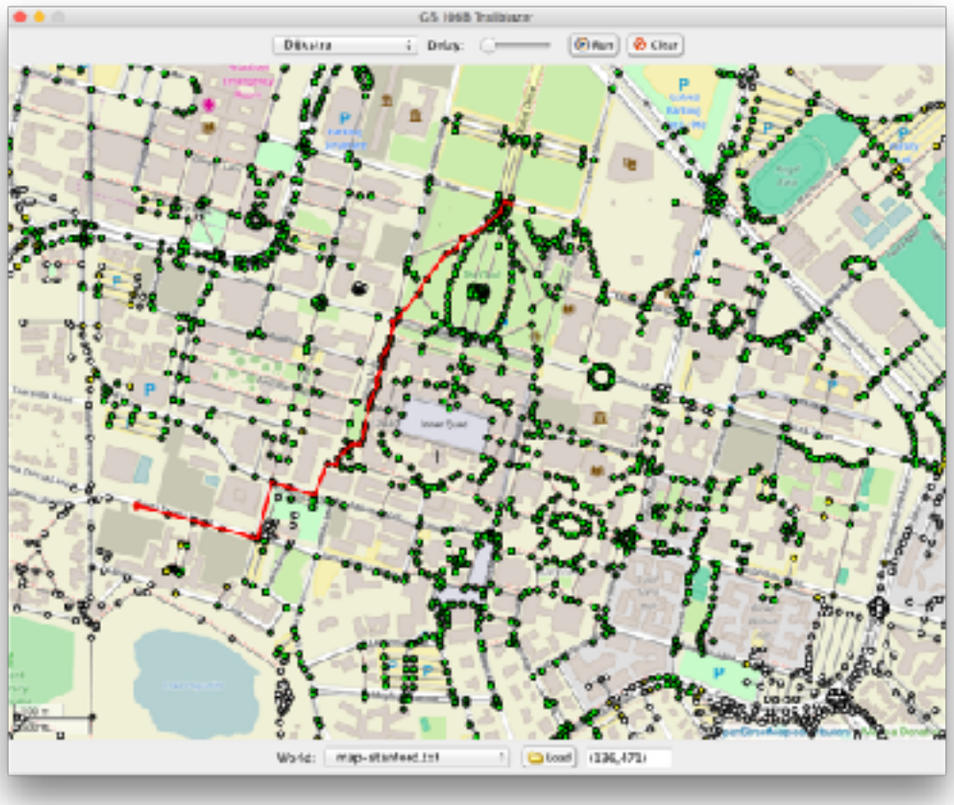
BUT YOU NEVER WANT IT TO BE LARGER THAN THE ACTUAL COST.

You want your heuristic to be as large as possible but you never want it to be larger than the actual cost.

The closer to 0 it is, the more often your algorithm will choose paths based off of their cost up to the current node. So you want the value of your heuristic to be large, but you want to always make sure its still admissible.

GOOGLE MAPS

So let's talk about google maps.



Your next assignment, Trailblazer (out today), is actually implementing some of the functionality of Google Maps.

It turns out Google Maps actually works the same way under the hood as the algorithms we've been talking about all week. Intersections and curves are marked using nodes, and paths are calculated using a variation of A*. You can assume each road has a "cost" dictated by how difficult it is drive on the road or by traffic patterns.

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

- How many nodes are in the Google Maps graph?

You might be surprised that Google calculates the directions every single time that you ask for directions. And they do a little bit of precomputation to speed it up. But it turns out it isn't actually feasible to precompute every single direction. Because Google Maps has a lot of nodes.

WHY DOESN'T GOOGLE MAPS PRECOMPUTE DIRECTIONS?

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

Around 75 million nodes.

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?

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

(Roughly) N

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?

And calculating all of those would take a long time.

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