Open in app ↗ Sign up Sign In

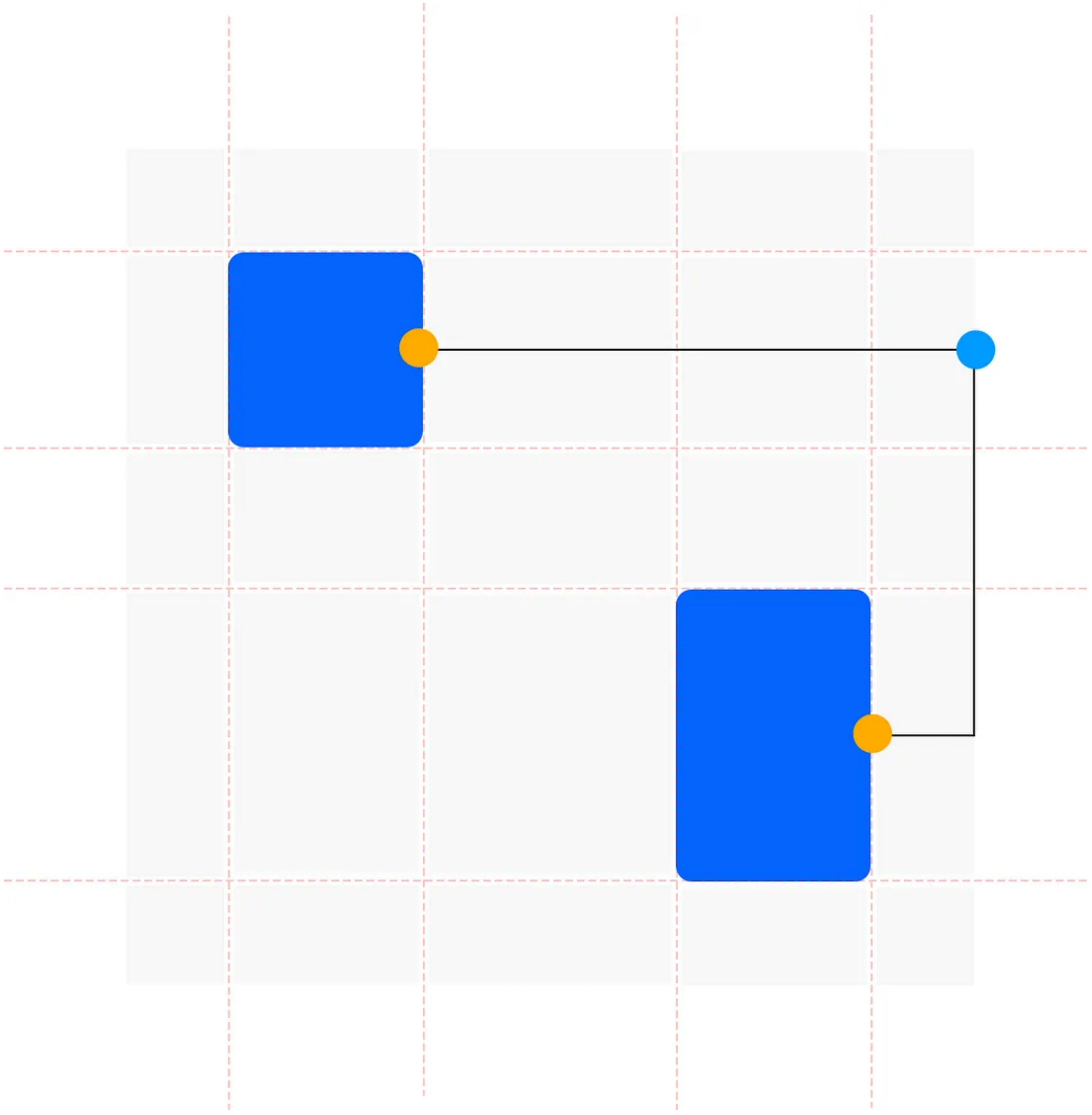
Published in The Startup

Jose M Menendez Follow
Dec 9, 2020 · 7 min read · Listen

Save

Routing Orthogonal Diagram Connectors in JavaScript

230 1



Quick links

- Live Example
- Github source

Background

This article is about how I got this problem solved in a robust way. Personally, this is how I got this out of my system.

> *TL;DR: I know there's libraries for this. It was just for fun.*

Orthogonal connectors in diagrams. They are my favorite when designing any diagram, and I should specify that I also love expressing things and thoughts as diagrams.

No that long ago, I was working on an app that required modeling some flows using a diagram-like editor. The use case was simple and enclosed enough not to go to a complex and chunky third party solution for it.

I won't enter full-rant mode about libraries, but it was a small piece of code to bring along an entire chunk of a very specialized library for such a seemingly small task.

Everything was going all right until we hit the point of connecting the shapes. We were shooting of course for orthogonal connectors as the best way of displaying beautiful diagrams. It was one of those things you just don't think much about, how it works or how complex the implementation could be; but turned out to be a **way more sophisticated** task than anticipated.

A few years later, still feeling the ghost of that challenge haunting me from my closet, I decided to give it a try again. This is the story of how I came with the approach presented in this article.

The way the live demo displays information is specifically designed to enable a good understanding of how the algorithm works. In my opinion, the solution is production-ready.

I decided to write this article because of how hard it was to find someone that explains simply how to achieve this routing. I even gave up when trying to follow the code on the popular libraries that do this. Code about geometry is often almost unreadable if there is not a graphical reference.

You might find the work useful, it might inspire you to tackle a ghost of your past, or just find the process entertaining enough to give it a read.

How I got there

As a father of two, in the middle of a pandemic and working from home, my spare time is short. Never mind. I don't have spare time. What I did is to allocate bursts of

30 minutes per day to give this a few cycles, first to propose theories, then to implement them.

Laying out the groundwork

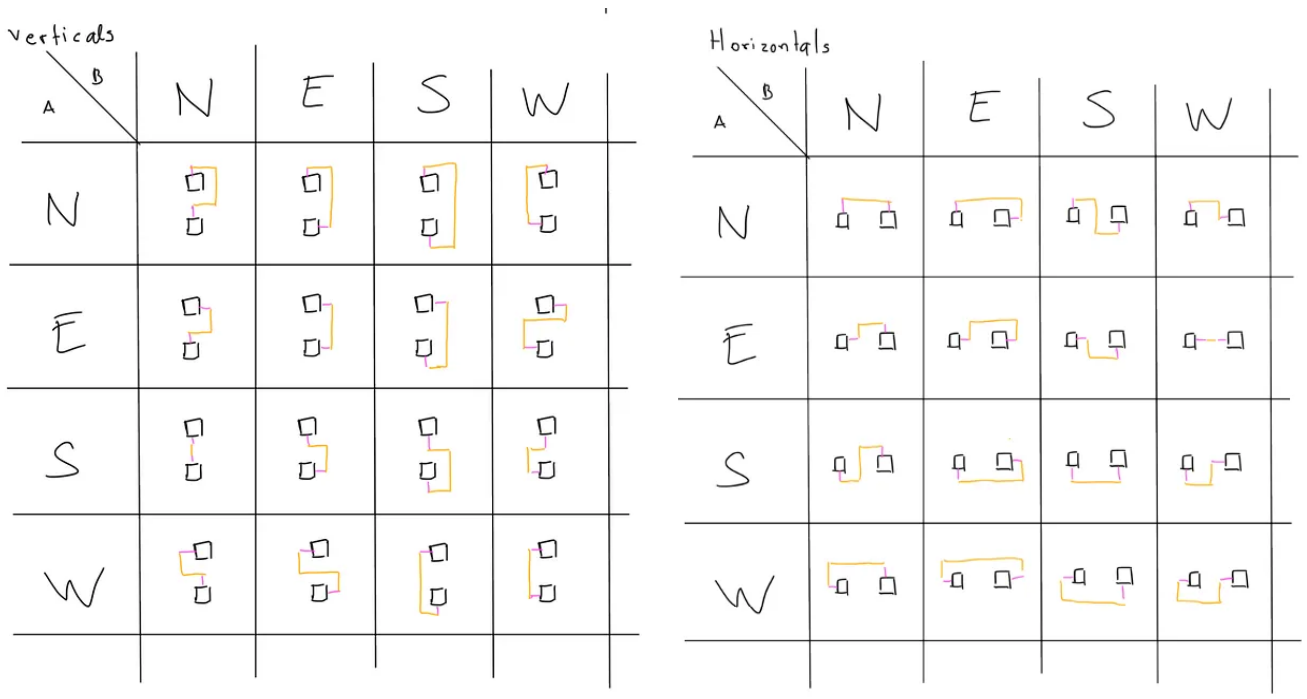
So, back to orthogonal connectors.

- Connector means they connect A to B
- Orthogonal means that lines go vertically or horizontally until they connect.
- It was a matter of automating something that comes so "natural" when doing it by hand.

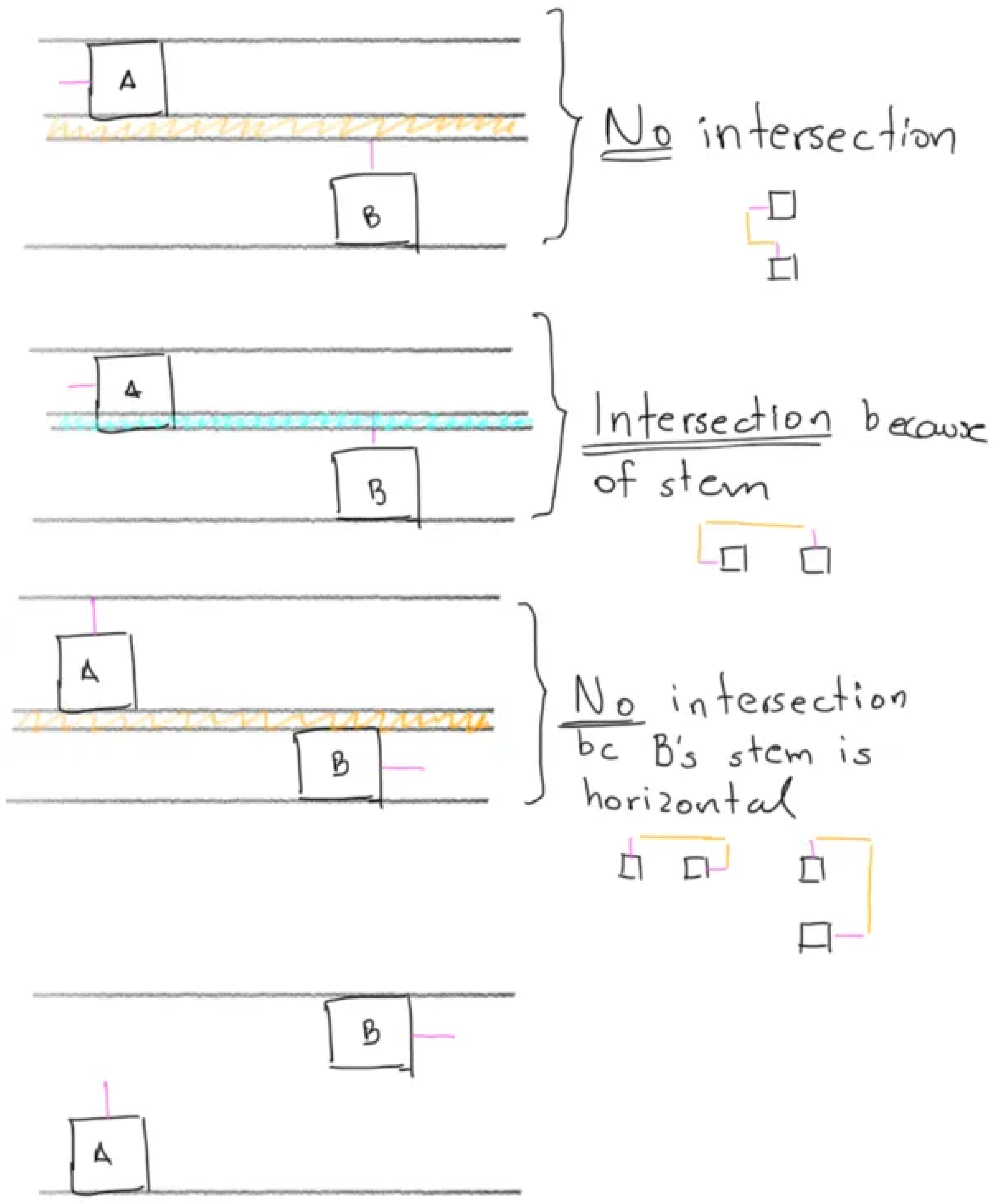
Early ideas that didn't work

Not going into much detail, but I basically tried a few "heuristics" based on notes and notes as I was absorbing and abstracting the problem, like trying to identify common patterns to solutions and so on. It was after a few tried attempts that I started thinking about a grid of possible pathways.

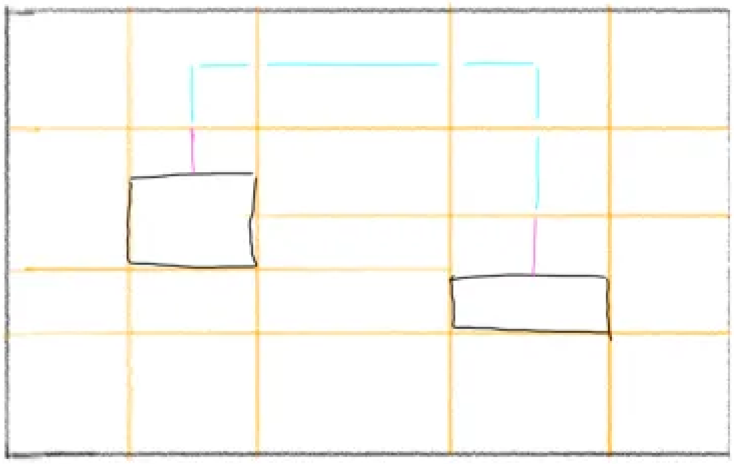
There were a few ideas that just didn't work as expected before I started seriously thinking about going the "grid way"

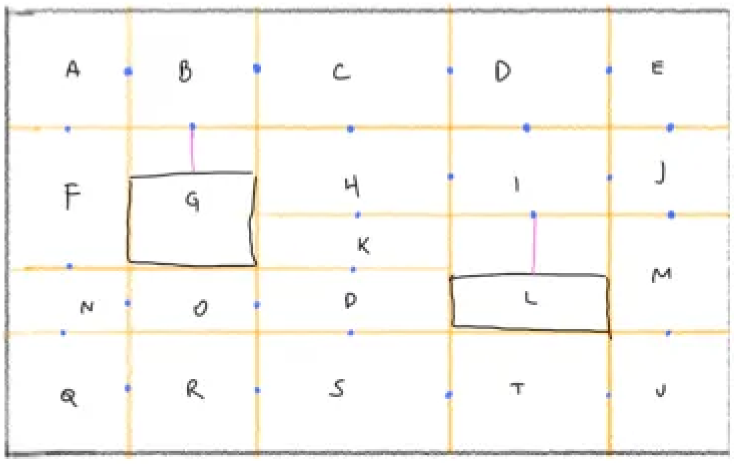


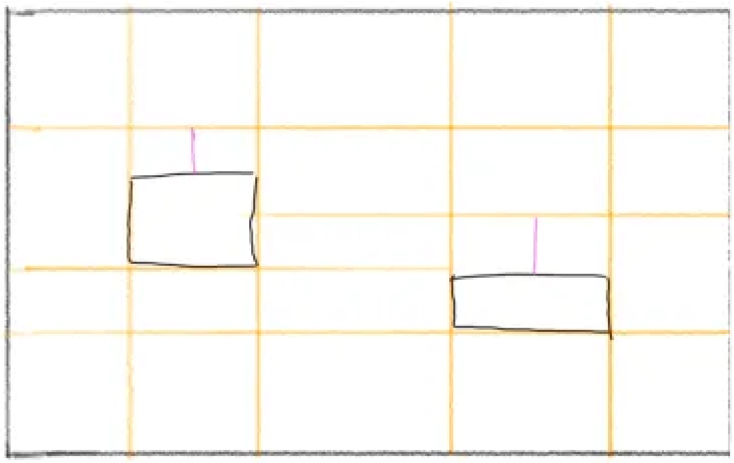
One idea was to sketch common patterns and apply them by identifying the applying situation


A
B
No intersection
A
B
Intersection because of stem
A
B
No intersection bc B's stem is horizontal
B
A

It started becoming sort of a nightmare to work a way to identify the corresponding layout







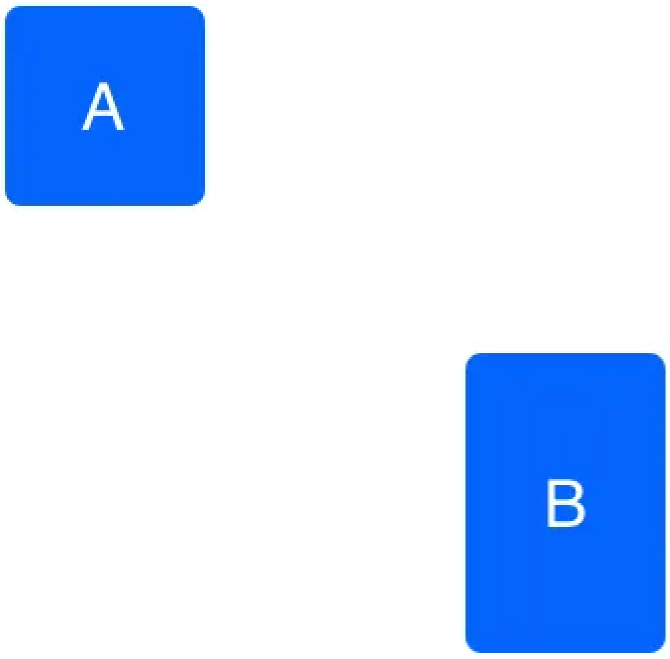
Finally, playing with the “grid” idea made much more sense as a robust solution.

The Grid Recipe

- Of course, a grid was the solution. I love grids.
- Where to begin?

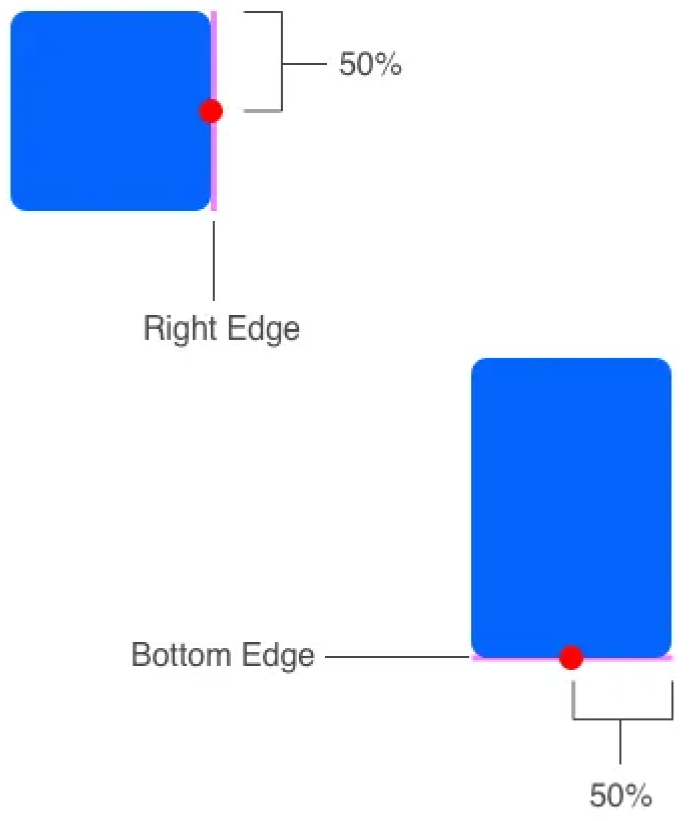
Two Shapes

We begin the journey with an *origin* and a *destination*, consisting of:



We start with two shapes to connect

- Bounds (Rectangle)
- Side of connection (Left, Right, Top, Bottom)
- The distance of connection (Relative to the side)

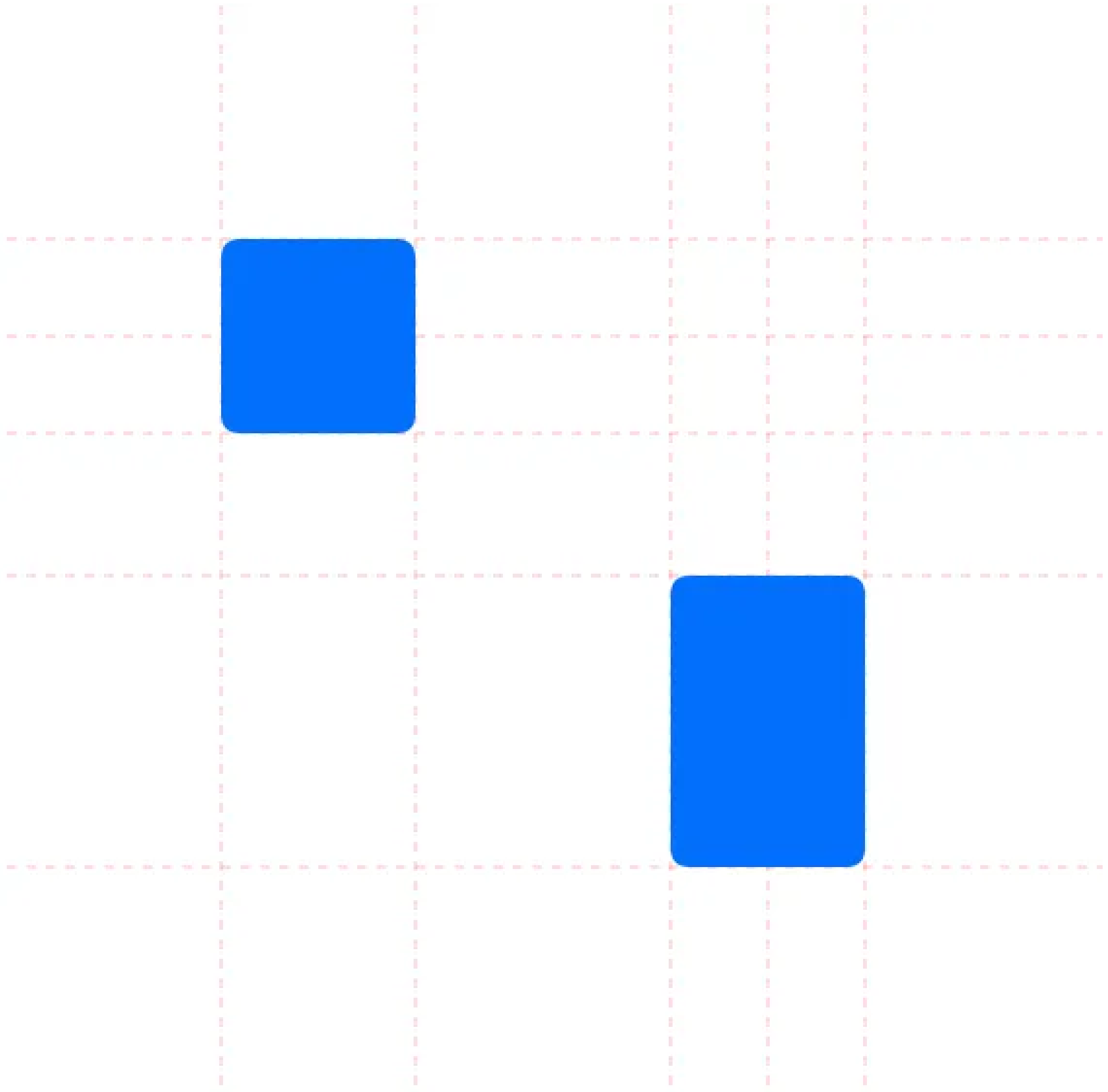


We define the points we have to connect

The illustration shows our two shapes, the first one has a spot inside, and the second one on its *Bottom* side. Let's assume that the distance for both sides is 0.5, or 50%, this is, their middle points.

Rulers Layer

First, we draw our rulers, with this identifying the coordinates of interest, which we will use later for creating the grid.

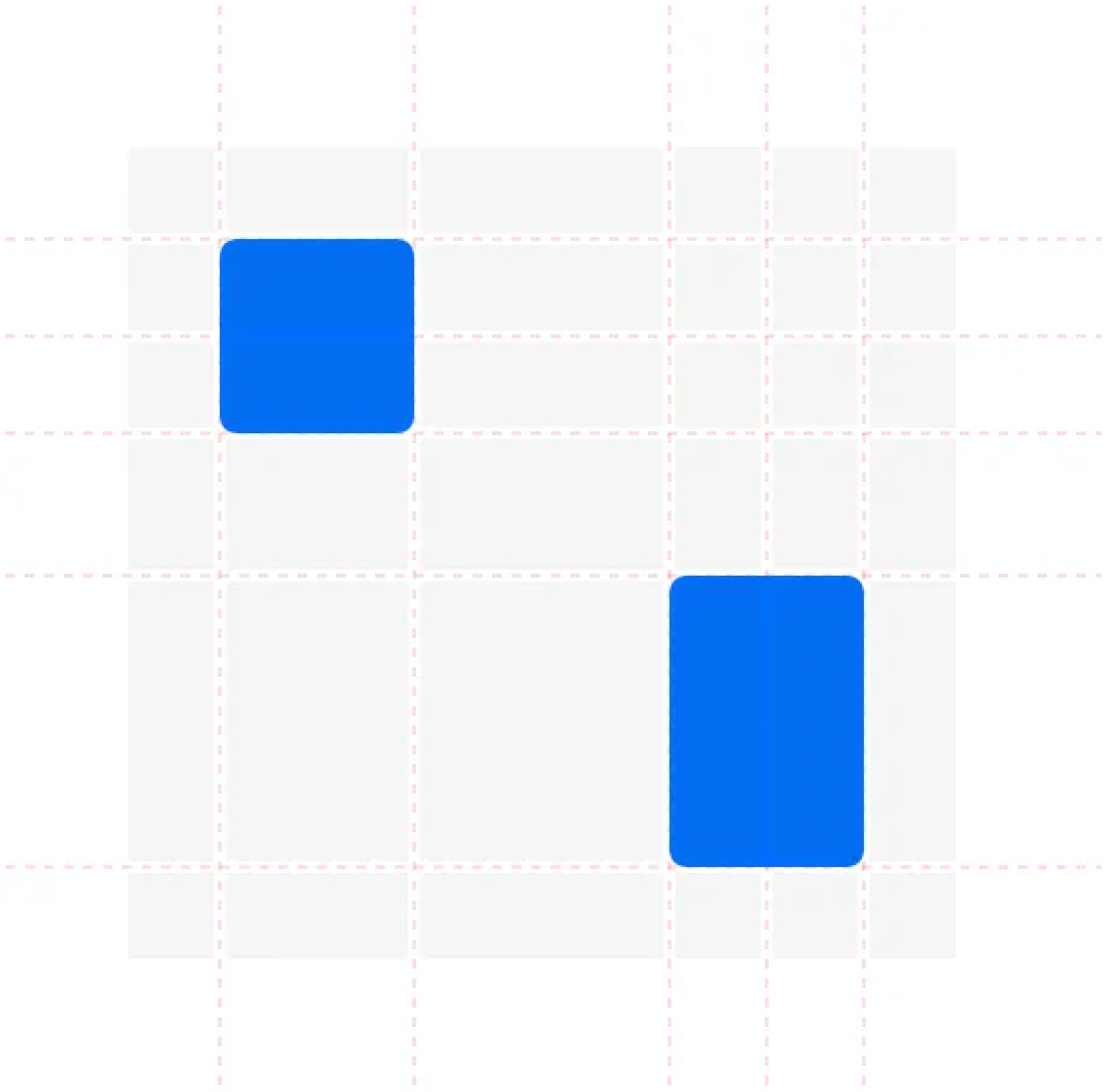


Rulers highlight the relevant coordinates to start laying out a grid

Observe that we added a ruler relative to our sides of interest, this is at 50% of both the horizontal axis for shape A and at 50% of the vertical axis for shape B, this is because our origin and destination points are *Right* and *Bottom*, respectively.

Grid Layer

Then we use these rulers to define a Grid. You can think of this grid as a bounding box for both rectangles, sliced by the rulers defined in the last step.



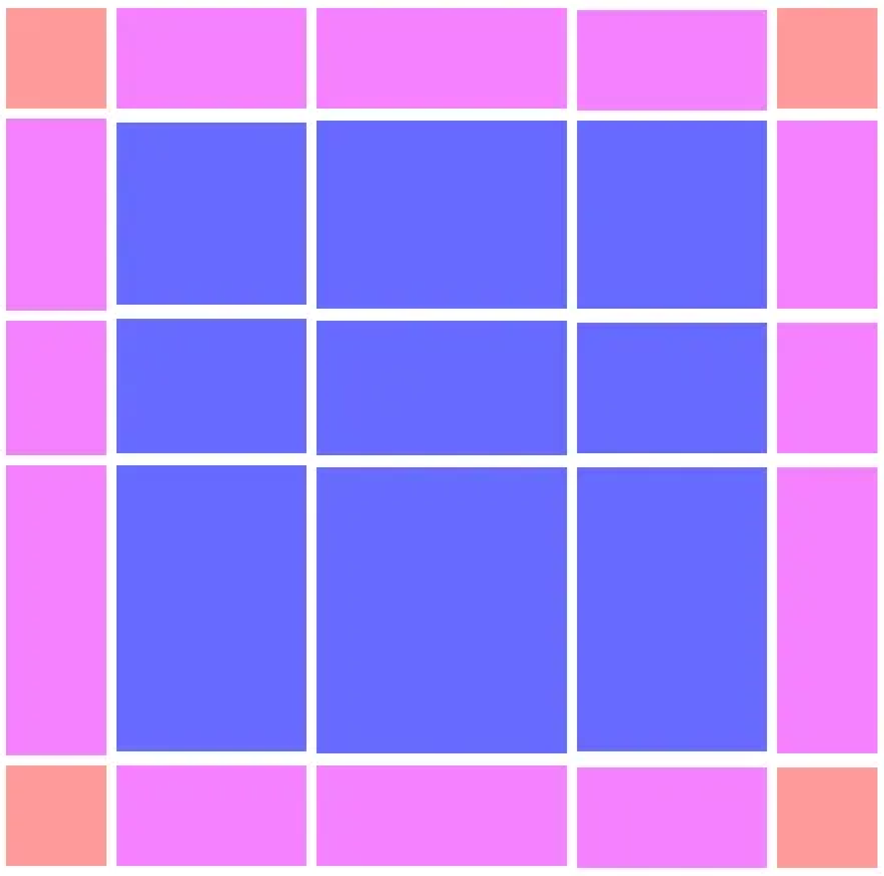
We create a grid by slicing with our rulers

Notice that the Grid on the illustration shows little “gaps” close to the rulers, this is for visualization purposes only, the algorithm rectangles actually share edges with their neighbors.

Reference Points

We use our grid now to designate reference points. These points will be the nodes of our graph to do a “shortest path” search.

On this grid, we identify three types of slices:

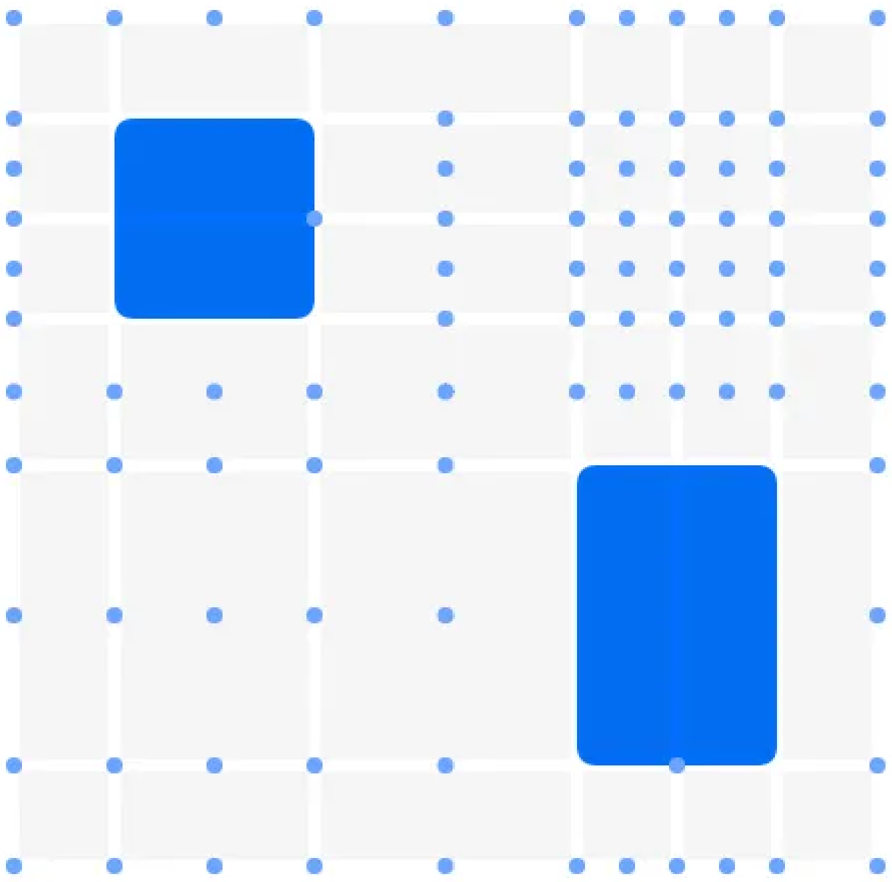


- Corner slice (Pink)
- Edge slice (Magenta)
- Internal slice (Blue)

The recipe for adding reference points to the slices is:

- All slices: Add points for each corner of the rectangle.
- Internal slice: Add points for the middle points of the edges, as well as the center of the rectangle.
- Edge slice: Add a point for the middle, external-facing edge.
- Don't add a point if it coincides with a shaped edge.

After those steps, **we add a point** to where the origin and destination points are. These should be the only points that coincide with shape edges.



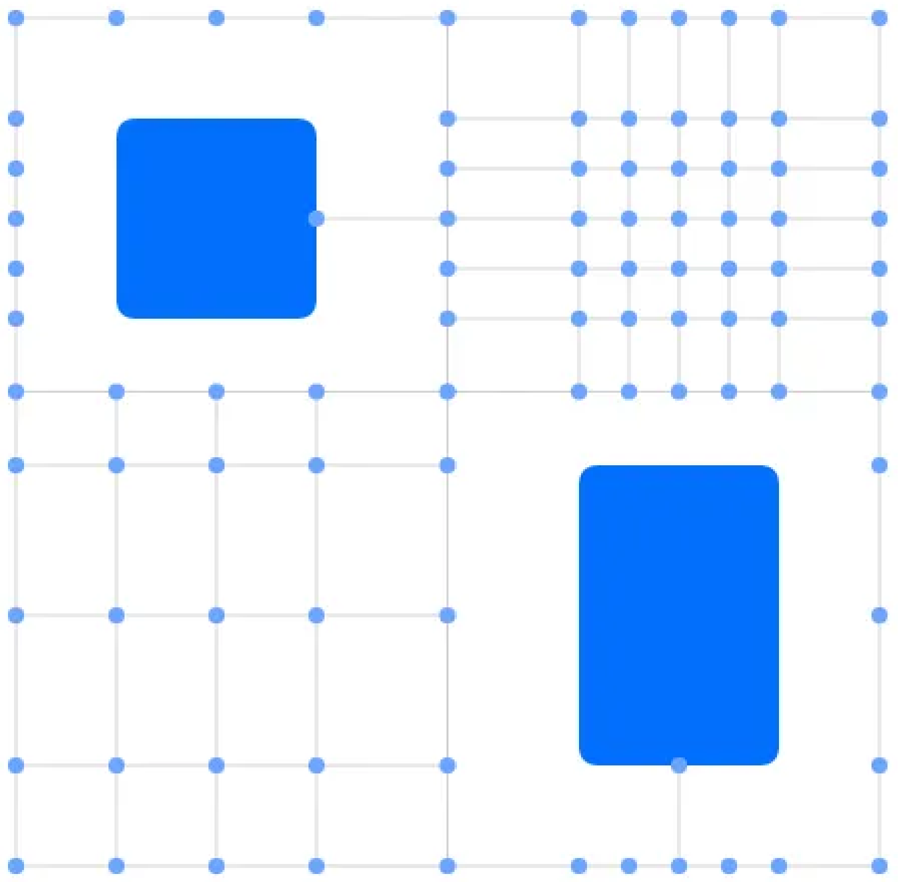
We place points of interest over our slices

Route Grid

Now, our final step to form the Route Grid. This grid is composed of all the possible orthogonal connections between the reference points we did in the previous step.

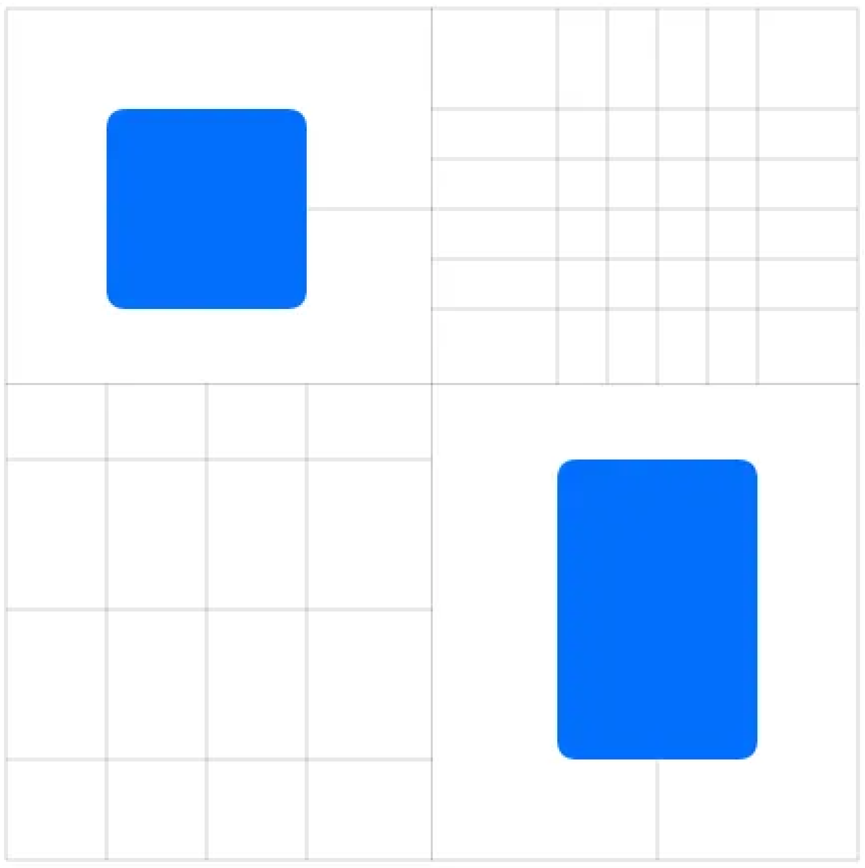
We create these connections based on the following rule:

- If the point has a neighbor horizontally or vertically, without skipping places, add a connection.
- When adding a connection, make it both ways, so the algorithm can find the best path.



We make orthogonal connections between the points

The route-grid without the points looks a little closer to our potential solution:



The Route-Grid has all the possible paths contained

This beautiful grid is formed by **all the possible** orthogonal connections to reach from our A to B.

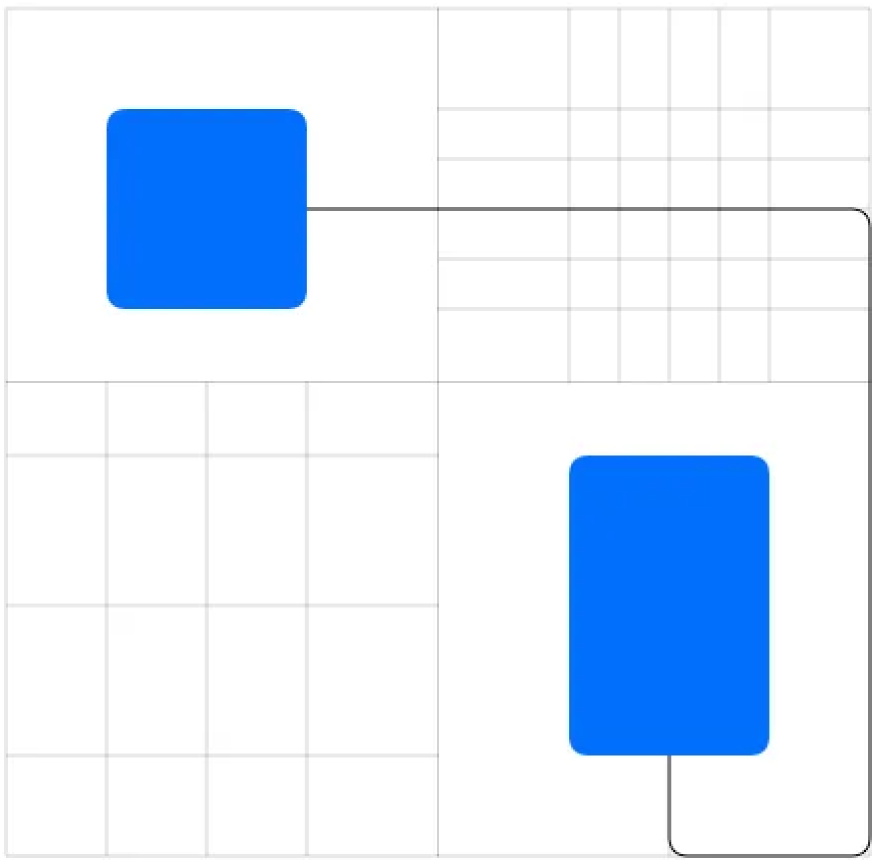
Find the Path!

Once we have our graph nodes well-formed, it’s all a matter of discovering the shortest path between our origin and destination points.

Now, feel free to suggest something more sophisticated, but I just used the Dijkstra Algorithm. It performed quite well on any layout preserving a fluid consistent 60fps animation on my computer so it was good enough for me.

The one modification I had to make to the algorithm is:

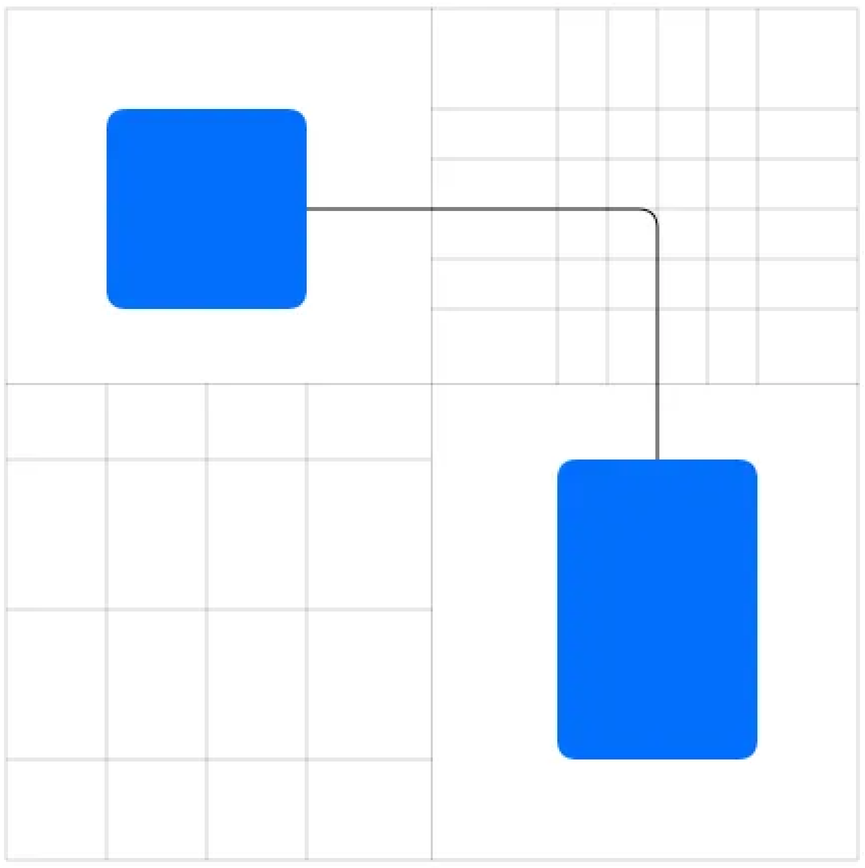
When calculating the cost of navigation between nodes, we **punish the change of direction**, i.e. going from vertical to horizontal, so it would break the fewer times possibles making some funky “staircase” effect on the resulting path.



We have a route!

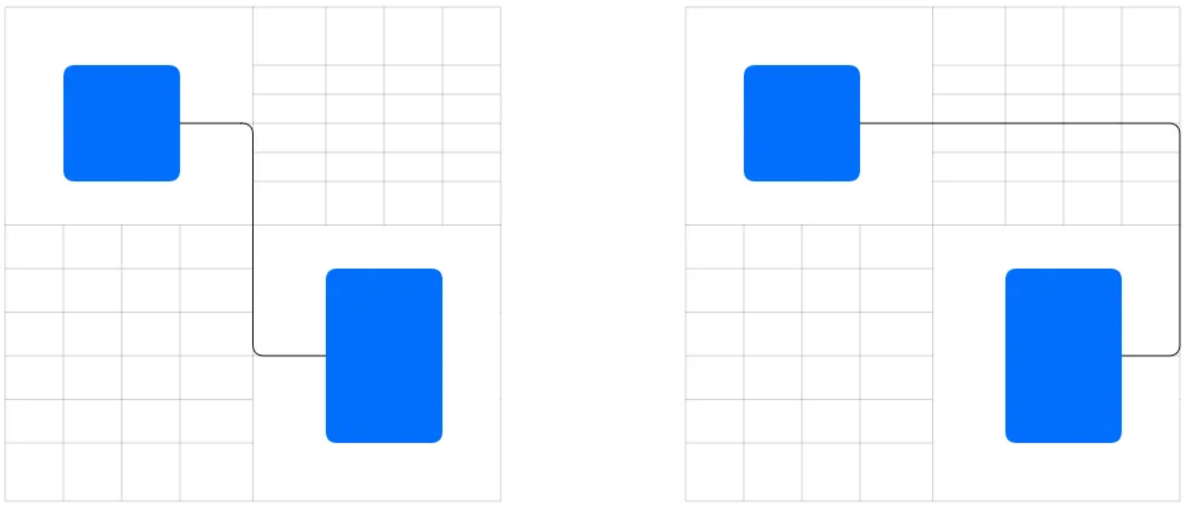
It was truly great to see the first paths come to life after so many years of having this problem in my ghost closet.

From there, because of the *rulers*-system used, the grid when connecting to the north part of the B shape looks exactly the same:



The same grid is formed when connecting to the north of the B shape

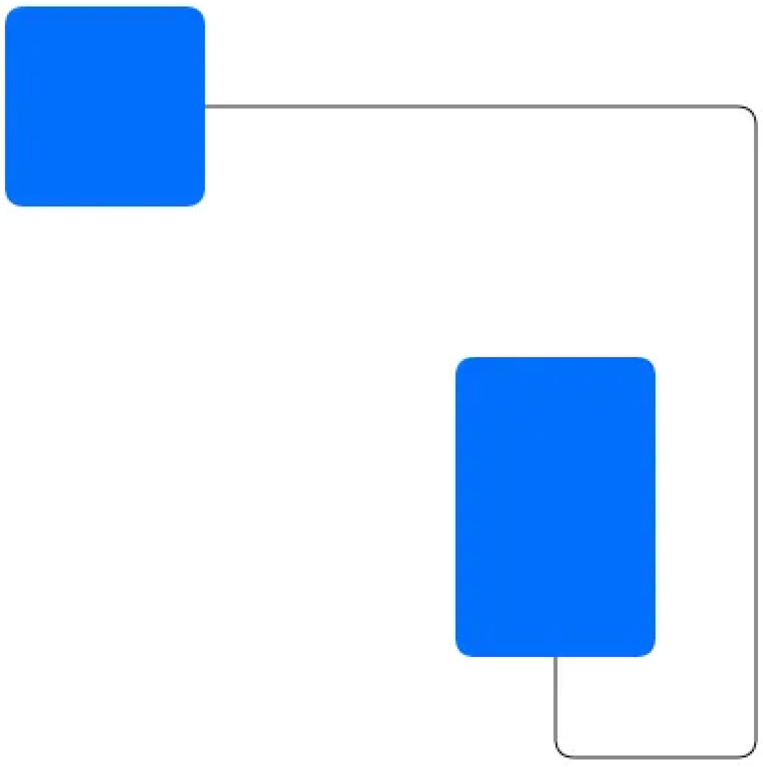
When the edge on the B shape is a horizontal one, the grid changes slightly, since there is a ruler traced across the horizontal axis of the B shape instead of the vertical one in the previous examples:



Slightly different grid for connecting to the horizontal sides.

🎉 End Result

Removing away all traces of the algorithm calculations, we get our pure connector:



Final connector without any metadata drawn.

About the Rounded Corners

I used this other Gist I have for drawing the rounded corners on the live demo.

The Code

- The Github Gist delivers the mentioned code as a node.js module. You can copy/paste the HTML example to see it at work.
- The gist provides the API for using it.
- The class *OrthogonalConnector* also exposes *byproduct* data, so you can analyze the different layer components of the algorithm.
- This byproduct data is what's used to render them on the live editor.

Hope you find this useful, please do submit feedback about it. ✌️

Diagrams Orthogonal Java Script