# Edge detection using resonating neural networks

## Lines

* Find all the directions possible for each pixel to build a line. This is done by comparing predefined ‘frames’ of pixels with the input. A match is converted into it’s frame. In short, we try to find if we can follow a pixel along any of the following paths: | - / or \.
* Each pixel is compared in sequence. This is done to preserve threads. This also makes certain that we find all the items of 1 line in the correct sequence.
* To find the first frame, we start in the upper left corner and move from left to right – down. This will determine the direction of all lines found in the image. The first frame is found by using all the definitions that contain a pixel in the left upper corner.
* If a frame match is found, from that moment on, the frame determines which frames to check next. This allows us to greatly cut down on the number of items to check. We can do this since each frame can only be followed by a subset of all the frames.
* Each sequence of pixels that contains the same ‘frame’ is put into a ‘line’ cluster.

## Direction

* Each frame has 1 or more possible incoming directions and always produces a direction when a match is found. This direction is calculated based on the direction of the pixels in the current frame and that of the previous. So if a frame for ‘|’ is found, and the direction is ‘North’, the new direction will remain ‘North’. If it was ‘South’, it also remains ‘South’. If a corner was found in the image, like ‘⎦’ and the prev direction was ‘South’, the new one would be ‘West’ (and visa versa).
* As the previous frame determines which frames to check next, it also determines the current direction.

## Shapes

* Whenever a line changes direction, a new cluster is started and the previous one is collected into a ‘shape’.
* If no more adjacent pixels are found, the shape is closed and collected in the list of all shapes in the image.
* When a pixel has multiple possible directions (as in a ‘T’ point for instance, if you come from the east, you can continue east or go south), both paths are evaluated at the same time (using a split).
* The new shape will become a ‘sub shape’: a part of the original shape. This is done by adding the new shape as a child of the original shape.
* When the first pixel of a new shape is found, which can go in 2 directions (usually for corners, not so much for strait lines, but those can also be used), both directions are also traversed at the same time. For this situation we also create a ‘sub shape’ which is added to the first. But if both shapes meet again, they are fused into 1, by adding all the children of one into the other, either at the end or the beginning, depending who gets too who first. A lock is required to make this thread save.

## About splits

Splits are used in various places: when a pixel has multiple possible directions to travel, but also for going to the next position that needs to be checked. As stated before, when a frame match is found, it’s the frame that determines the direction and which frames to check next. Most frames can have multiple directions as a next position. For instance, a horizontal line going ‘East (→) can be followed by a↘, → or ↗. For the last one, we need to change the line position as well. This requires a split. The problem here is that we don’t know yet if the split will yield any valid results. So handling the ‘line’ and ‘shape’ clusters becomes delicate: how do we prevent from duplicate or invalid data? When the variables are set to duplicate on each split, we will have a lot of invalid data and duplicate results since some lines will be present 2 or more times, p resented by duplicates. On the other hand, if we collect incorrect, the results of 1 split could be added to those of another, tangling up the image.

For lines, the solution lies in the direction of the pixel. If the direction of the next pixel can stay the same, this split keeps the references to the original line. If it is not possible to keep the direction, the line is cleared (previous data belongs to the other split) but a reference is stored as ‘attach to’, so we know where any possible new shape should start in the original.

Shapes are a bit more complex. It is possible that there is no valid next path, 1 split contains valid paths or more splits have a result. We solve this by a first come first serve basis: before splitting to ‘test the next pixel’, we backup the ‘shape’ cluster. The split that can consume this cluster the first, can keep it, the other split(s) will have to ‘attach’ to the root. So actually before the split, 2 backups need to be made of the same value: the root and the to-be-consumed value.

The shape is also backed up into a ‘root shape’ so we can clear it.

Wrong -> real solution is to use 3by3 frames for the inner part, so that all directions can be found without performing a split.