This short article illustrates how the graph extractor recognizes a graph and extracts the mathematical information.

Goal

Given a digital image of a mathematical graph which can be expressed in the form $G = (V, E)$, our goal is to extract the graph out of the picture and produce an output that can be stored in a high level format (an adjacency matrix for example).

Tool

The program was implemented using Python with two key packages: numpy and opencv.

Input

Our inputs are digital images. Other than that there is a graph in the input picture, we also assume the following.

- The graph can be hand-drawn or machine-drawn and the image can be in several popular formats (*.png etc.). The pictures are assumed to be presented in RGB colors, even though in most of the cases there seems to be only black and white colors in the pictures.

- In the pictures we assume that the background is (mostly) white while the graph itself is (mostly) black.

- Some graphs come with labels. In our task we choose to ignore them because it is beyond our mission to recognize the text and we do not think the order of the vertices matter to the way we recognize the graph. However in practice we did add a feature in the program to allow the user to manually fix the labeling before storing the result.

Once it receives the image, the program will convert it into a gray scale version and then perform thresholding to set the pixels with a value lower than t to 1 and the ones with a value higher than t to 0. Basically we are converting the image to a binary version where 0s represent background while 1s represent the contents. It is hard to choose a t that is good for all the images, so we allow the user to manually adjust this threshold value.

Recognition

When a human attempts to recognize a graph, they most likely will first find the vertices, and then attempt to follow the edges to find how they are linking to each other. Our program does the same thing, therefore

we can divide the whole problem into two parts: vertex recognition and edge extraction.

Locating Vertices

The hardest problem in this part is that, there are way too many ways to present vertices in a graph picture. A vertex can be a circle, a triangle, or even simply just a dot; it can also be filled or hallow. Due to the variety of shapes, we are using a very straightforward method to find the vertices: we ask the user to crop a rectangle piece of the picture which contains one of the vertices, and then we use a technique called template matching to find the rest of the vertices. The template matching algorithm simply takes an input piece of image as sample, creates a window of the same size of the sample, slides the window in another image, compares the sample with each window and detects where the highest similarity occurs. This will be done till all the vertices are found.

Extracting Edges

Now that the location of each vertex is known, we can start to find how they are linking to each other. One approach is line or curve detection which is widely used in computer vision. However they do not seem to work well in our problem because if three vertices are drawn in the same line while there is only one edge among them, then the line detection will tell that there are two edges instead of one. It will then require more work to remove the false edges. Therefore we will use a more straightforward approach: we want to achieve the goal by "tracing" along the edges.

We first perform image thinning on the binary image so that the graph

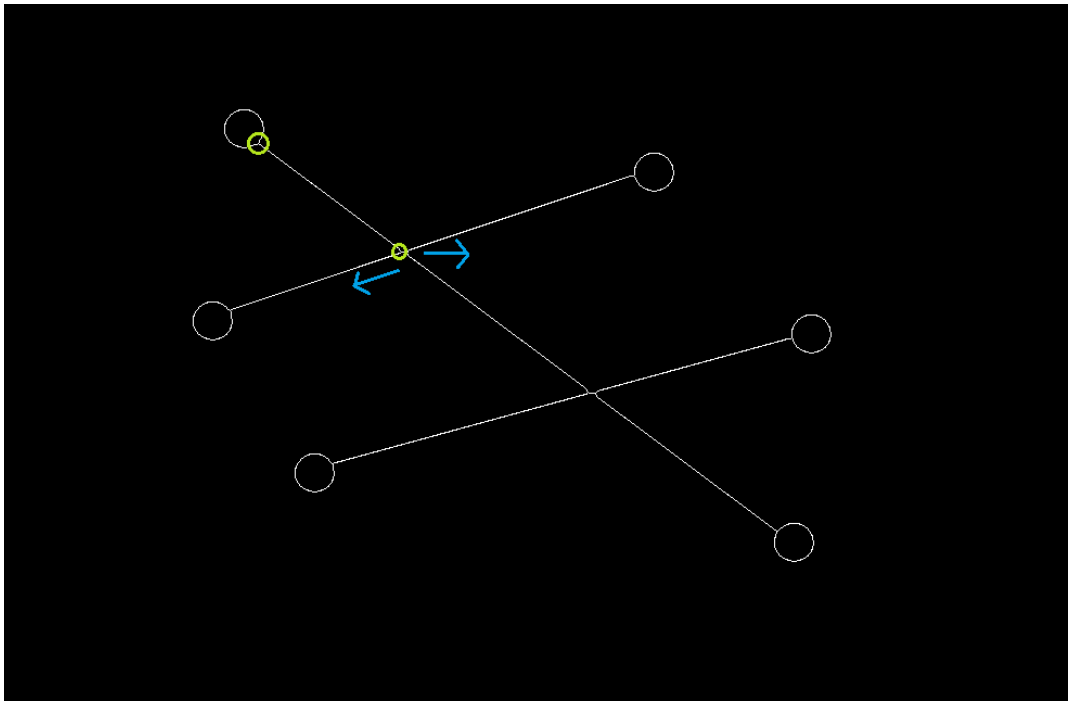
has a width of one pixel. This is done by using an adapted version of zhang-seun's image thinning algorithm. After that, because we already know the location of the vertices, we can separate all the pixels of 1 into two groups: the vertex pixel set P_V and the edge pixel set P_E. Of all the edge pixels we can easily recognize the ones that are right next to a vertex. We associate them with their closest vertex and put them into a set P_{end}. These pixels are the end points of the edges, and they will serve as the starting points of each iteration of our algorithm. Let's define $N_8(p)$ to be a set of pixels in the 8-neighborhood of a certain pixel p and $V(p_{end})$ to be the vertex associated with a pixel $p_{end} \in P_{end}$. The algorithm is the following.

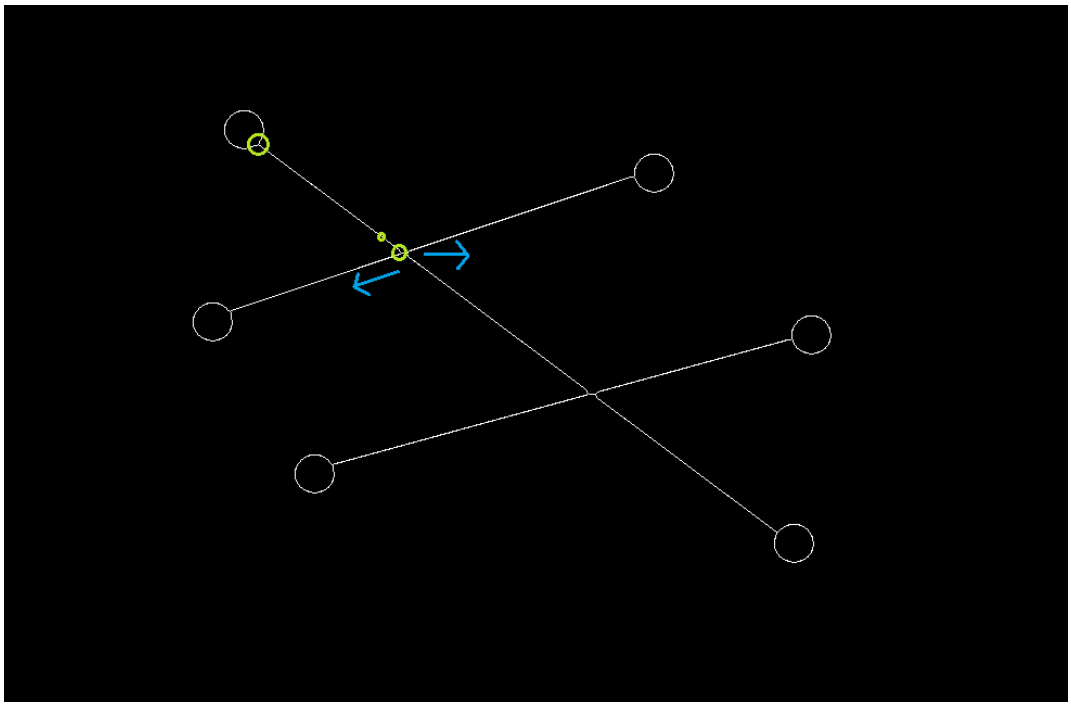
initiate $result = [\ ]$
for each $p_{end} \in P_{end}$:
    initiate $current = p_{end}$
    initiate $known = [current]$
    initiate $unknown =$ null
    for each $n \in N_8(current)$:
        if $n.value == 1$ **and** $n \notin known$ **and** $n \notin P_V$:
            $unknown =$ n
    while $unknown \neq$ null:
        $current = p_{end}$
        $known$.add($current$)
        $unknown =$ null
        for each $n \in N_8(current)$:
            if $n.value == 1$ **and** $n \notin known$ **and** $n \notin P_V$:
                $unknown =$ n

result.add((*V*(p_{end}), *V*(*current*)))
P_{end}.remove(*current*)
return *result*

Intuitively, for each outer iteration, the algorithm starts from some end point of an edge, traverses the edge by recursively choosing the unexplored pixel in the 8-neighborhood of the current pixel to be the next stone to step on, and eventually reaches a point where there is no unexplored pixel in the 8-neighborhood, which means the current point is another end point in P_{end} and hence an edge is found.

The problem of this algorithm is that it only works on the graphs where there are no two or more edges crossing each other, but clearly in most of the cases this is not true. To handle such cases one may consider changing the *unknown* in the algorithm to be a list, in this case when it reaches an intersection, it will store all possible pixels instead of choosing merely one. But we will then need to be able to correctly choose the next "unknown" pixel in the list. Auer et al. has proposed a solution which is illustrated in **Fig. 1**. The intuitive assumption of this approach is that we tend to draw the a continuous curve as smooth as possible. Nevertheless, we find it not working well when there are more than two edges intersecting at the same point and when the image thinning distorts the graph too much. **Fig. 2** shows the former case. We believe that this is because their approach only attempts to solve the problem (mainly the intersection part) locally. Thus we propose our own approach which adapts their big idea but uses more global information.





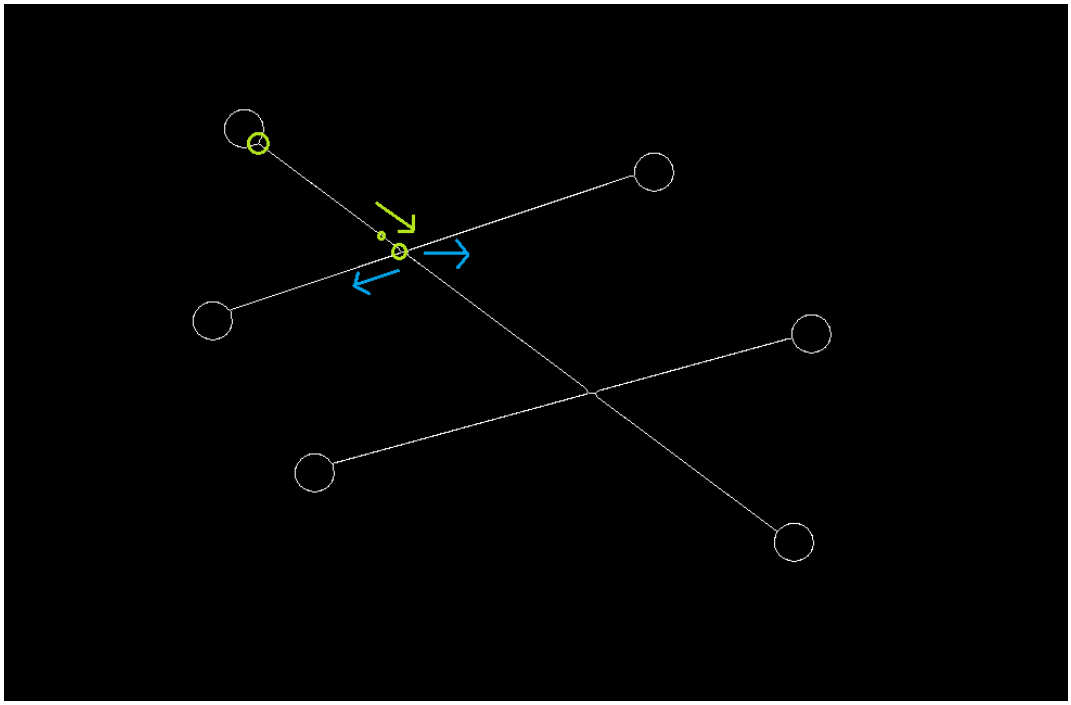


Fig. 1 The first image shows that an intersection is reached after starting from the top-right vertex and there are two potential directions(blue arrows) to go; in the second image, a point some distance back from the current intersection is selected; in the last image a vector in green is obtained using the two pixels circled in green, and this vector will be used to compared with the two blue vectors to determine which direction to go.

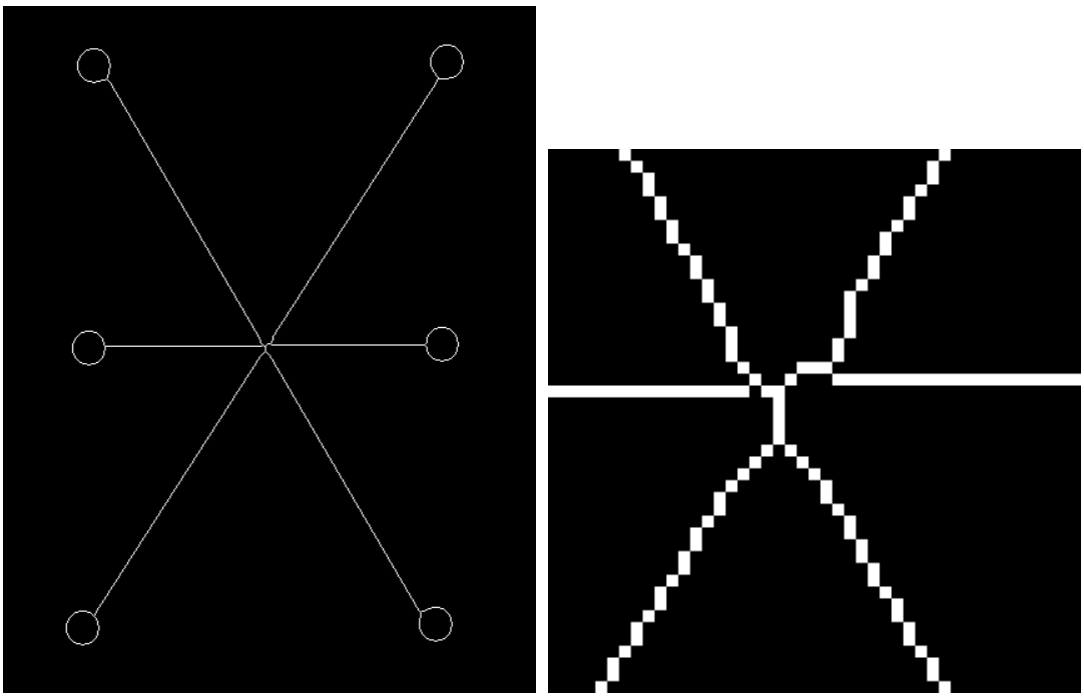


Fig. 2 The left image shows a thinned graph while the right one tells how the intersections at the middle look like.

Our algorithm is the following.

Step 1. Construct a directed weighted graph $G = (V \cup T, E, W)$ where V contains the vertices in the original graph, $T = \emptyset$, $E = \emptyset$ and $W = \emptyset$. For each pixel p in P_E, the edge pixel set in the thinned binary image, determine if it is an intersection. If so then add a node associated with this intersection to T. Then for each vertex pair $x, y \in V \cup T$, add edge (x, y) and (y, x) to E. At the end of this step, for each edge (x, y), let $\{p_0, p_1, p_2, ..., p_n\}$ be a list of edge pixels connecting x and y and let $\{v_0, v_1, v_2, ..., v_{n-1}\}$ be a list of vectors where v_i is a vector pointing from

p_i to p_{i+1}, then the weight of the edge $w(x, y)$ is defined as:

$$w(x, y) = \sum_{i=1}^{n-1} z_i v_i,$$

where z_i follows some normal distribution with mean equals to a small value.

Step 2. Create connected components on $V \cup T$ such that each connect component possesses two properties: there exists a path from any vertex to another; two vertices, x and y, inside are directly connected if $(x, y) \in E$ and they are geologically closed enough to each other in the binary thinned image. After that replace every component with a single node and update G accordingly. This step is illustrated in **Fig. 3**.

Step 3. For each node x in T, let $P_x = \{p_1, p_2, ..., p_n\}$ be a set of edges going out of x. Now compute $Q_x = \{(p_i, p_j) | p_i, p_j \in P_x\}$ using the following objective function ($\theta(w(p_i), w(p_j))$ measures the angle between $w(p_i)$ and $w(p_j)$):

$$min \sum_{(p_i, p_j) \in Q_x} (\theta(w(p_i), w(p_j)) - \pi)^2.$$

Step 4. Starting from each $u \in V$, explore G along the edges. Let x be the previous node ($x = u$ at the beginning), y the current one, if $y \in V$ then record (x, u) into the final edge set R; otherwise find z such that $((y, x), (y, z)) \in Q_y$ where Q_y was computed in the previous step, and then set z to be the next node to explore.

Once the algorithm terminates, the set R will contain all of the edges in the original graph.

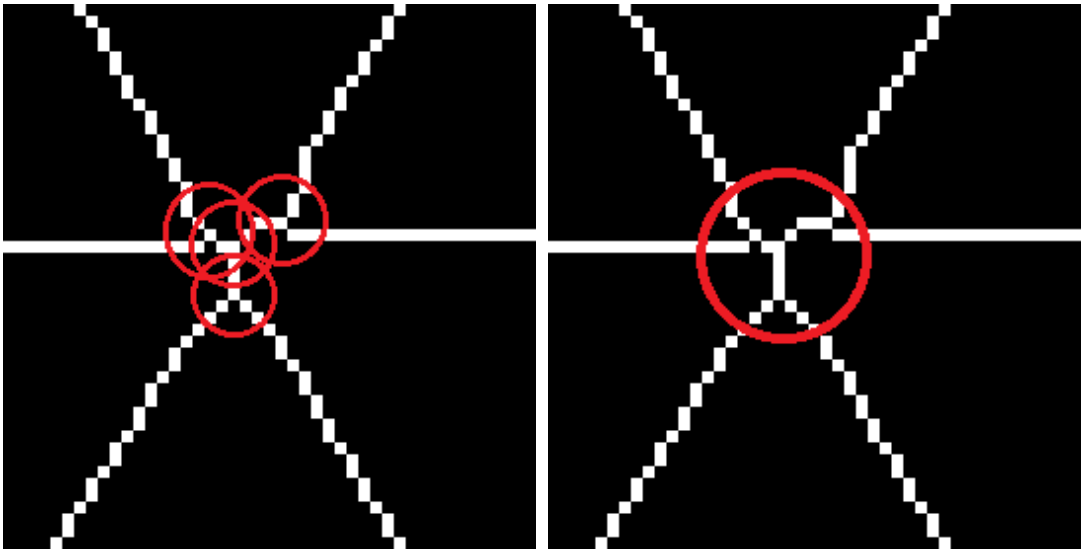


Fig. 3 In the left image, there are 4 intersections that can be recognized locally and treated as nodes. Because they are connected and very closed to each other, they can be grouped as one single node that represent a single intersection as shown in the right image.

The intuitions behind each step are illustrated below.

Step 1. The binary thinned image is hard to process due to the nature of pixels. Therefore in this step the main goal is to convert the "ugly" structures to some high-level ones. In particular, note that the intersections are all gone in G. The reason why we assign a weight for each edge with a vector sum is because we still want to make use of the vector approach like in Auer's paper in later steps. That the vector sum is a weighted sum is due to the fact that an edge may be drawn as a curve

instead of a straightline and attaching some normal distributed weight can take all the pixels in the edge into account while forcing the ones that are closer to the node to have more influence.

Step 2. The image thinning algorithm breaks the original intersections into multiple ones, so we want to restore the one-intersection structures. By assuming that after the thinning, the broken intersections will be closed to each other, we simply group them up by checking the connectivity and the distances.

Step 3. Now each node in T represents an intersection in the original graph. We assume that an edge, whether it is a line or a curve, is supposed to be drawn as smooth as possible, which implies that it shouldn't take a big turn after passing through an intersection. Therefore we can try to pair up the outgoing edges for each node in T such that the weights, which are in fact vectors describing how the edges are drawn, for each pair of edges form an angle as closed to π as possible.

Step 4. The intuition at this step is simple. We now have all the information we need, so we can restore the original graph by exploring the G we constructed. This can be done by starting from a vertex in V and following the edges in E to reach another vertex in V. If at some point we reach a node in T, then using the pair set obtained in **Step 3**, we know which direction to go next.

The algorithm turns out to be working quite well when recognizing edges. However it is not user-friendly because it takes too many user intersection

when locating vertices. We could not find a good heuristic method to recognize the vertices because there are too many variations. Therefore the next goal is to adapt machine learning techniques to help on this task.

References

- Auer C., Bachmaier C., Brandenburg F.J., Gleiner A., Reislhuber J. (2013) Optical Graph Recognition. In: Didimo W., Patrignani M. (eds) Graph Drawing. GD 2012. Lecture Notes in Computer Science, vol 7704. Springer, Berlin, Heidelberg