Generating Figures for Bongard Problems

By

Venkata Sai Krishna Maddineni

Masters in Computer Science

Under the Guidence of

Dr. Thomas Eskridge

Florida Institute of Technology

Contents

Figures

Equations

Introduction

Bongard Problems were first created by the soviet union's computer scientist called Mikhail Bongard. He was working on the pattern recognition which is a part of the cognitive science and was able to create 100 type of puzzles. The idea of the puzzles is that the there are six figures on the left side of the page and six figures on the right side of the page. The role of the solver is to find a rule or a pattern in which all the figure in the left side are following and the other side of the puzzle would be opposing the rule or would be a different rule. Once we were able to differentiate between the rules on the left side and the right the puzzle is solved.

There were lot of researchers tried to come up with the idea of calibrating a machine, which could solve these problems on its own. But the first person who brought these bongard problems into light was the D.R. Hofstadter, where he mentioned about them in his book called "*Goedel, Escher, Bach: an Eternal Golden Braid*". He explained the people how the system which could automate the such problem could help in the progress of technology. Then the Dr. Harry Foundalis started his dissertation and built an application called phaeco in 2006, which is an image processing technology software in which the author used to solve these problems using image processing algorithms. His software was impressive and have done a good job in solving these puzzles. This software was able to solve 15 out of 280 problems.

Since Artificial Intelligence has made a great progress in the new generation systems. Deep Learning and Neural networks are becoming increasingly popular, I have developed a program which could generate the figure to solve some of the problems in the bongard problem as to train these Neural neworks, we need to have huge amount of data, but we are lacking that and generating

figures are more essential in building a network to train and make it able to solve to feed the network. Here are some of the puzzles taken from the bongard book.

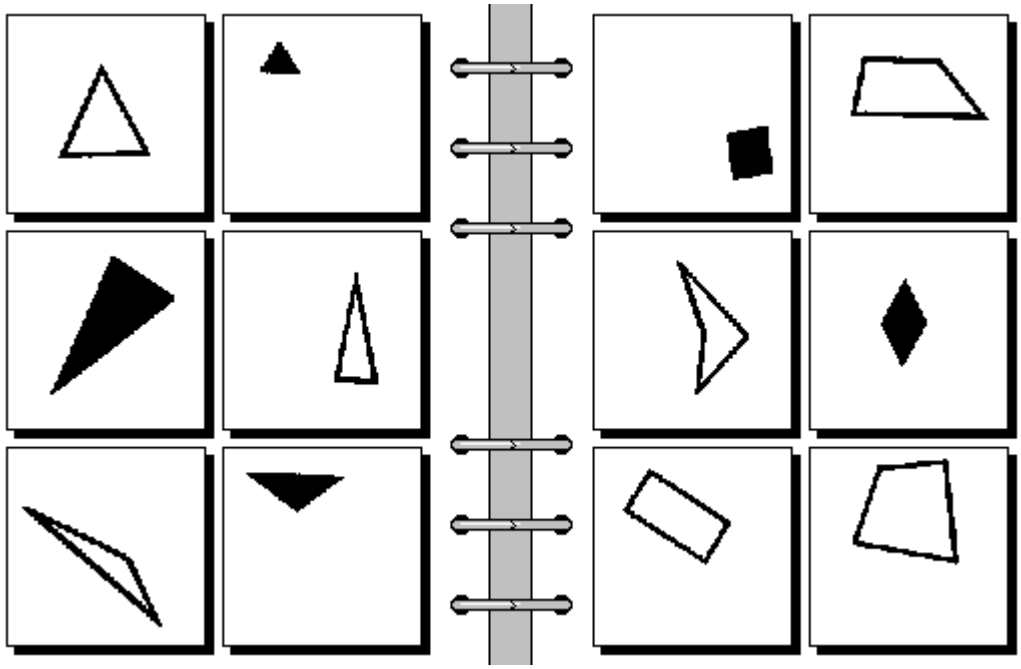

Figure I

In the above figure the figures in the left side are triangles whereas figures in the right side are quadrilaterals.

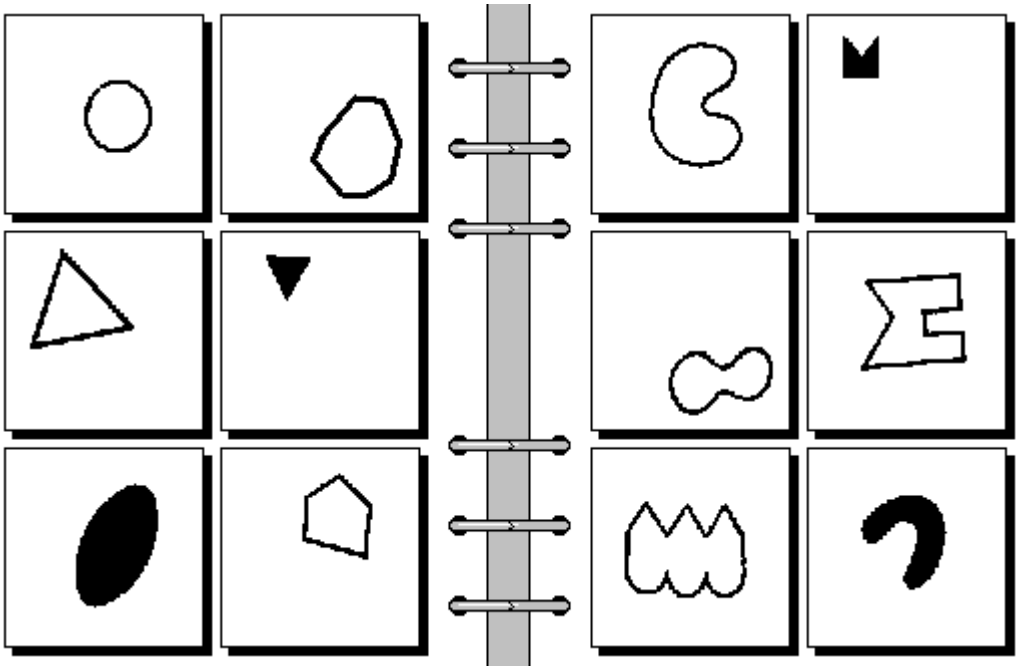

Figure II

In the above figure we can see the convex polygons on left side and concave polygons on the right side.

Implementation

The generation of the 2D Shapes are done using the java language, which has a package called awt. This is a canvas environment in which we draw 2D shapes. Before we get into the implementation, let us see the requirements first.

Requirements

- ➔ Windows, linux or Mac
- ➔ JDK 1.8 (Java Development Kit).
- ➔ JRE 1.8 (Java Runtime Environment).
- ➔ Any IDE such as Eclipse, Intellij Idea, Netbeans.
- ➔ Computer with the minimum memory of 4GB.
- ➔ Intel i5 is recommend atleast 3.0 GHz clock speed or i7 with atleast 1.6GHz clock speed.
- ➔ Batik java SVG tool kit is used for converting canvas images to svg. Please use this link to download the required file.

The code is implemented using the canvas awt by java. The following is the folder structure in the repository.

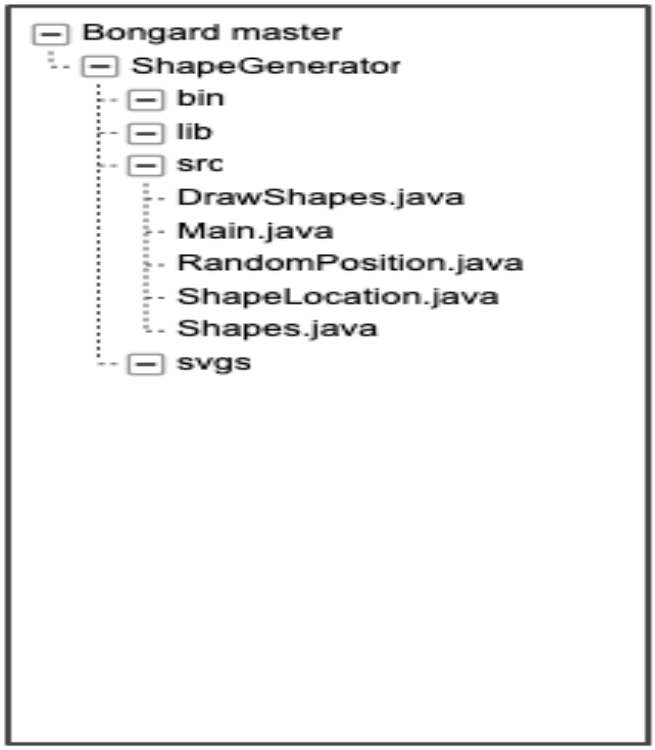

Figure III

The following is the folder structure of our generator in which the Shape Generator is a java project made using the Intellij Idea (IDE). The lib folder contains all the JAR dependency files required to add to the java project environment. To install the dependencies

1. Right click on the project folder
2. Click on open module settings
3. Select modules under Project settings
4. Choose Dependencies on Top
5. Then select + button at the bottom of the dialog
6. Finally select JAR and upload your dependencies

The below figure shows the dependencies installed.

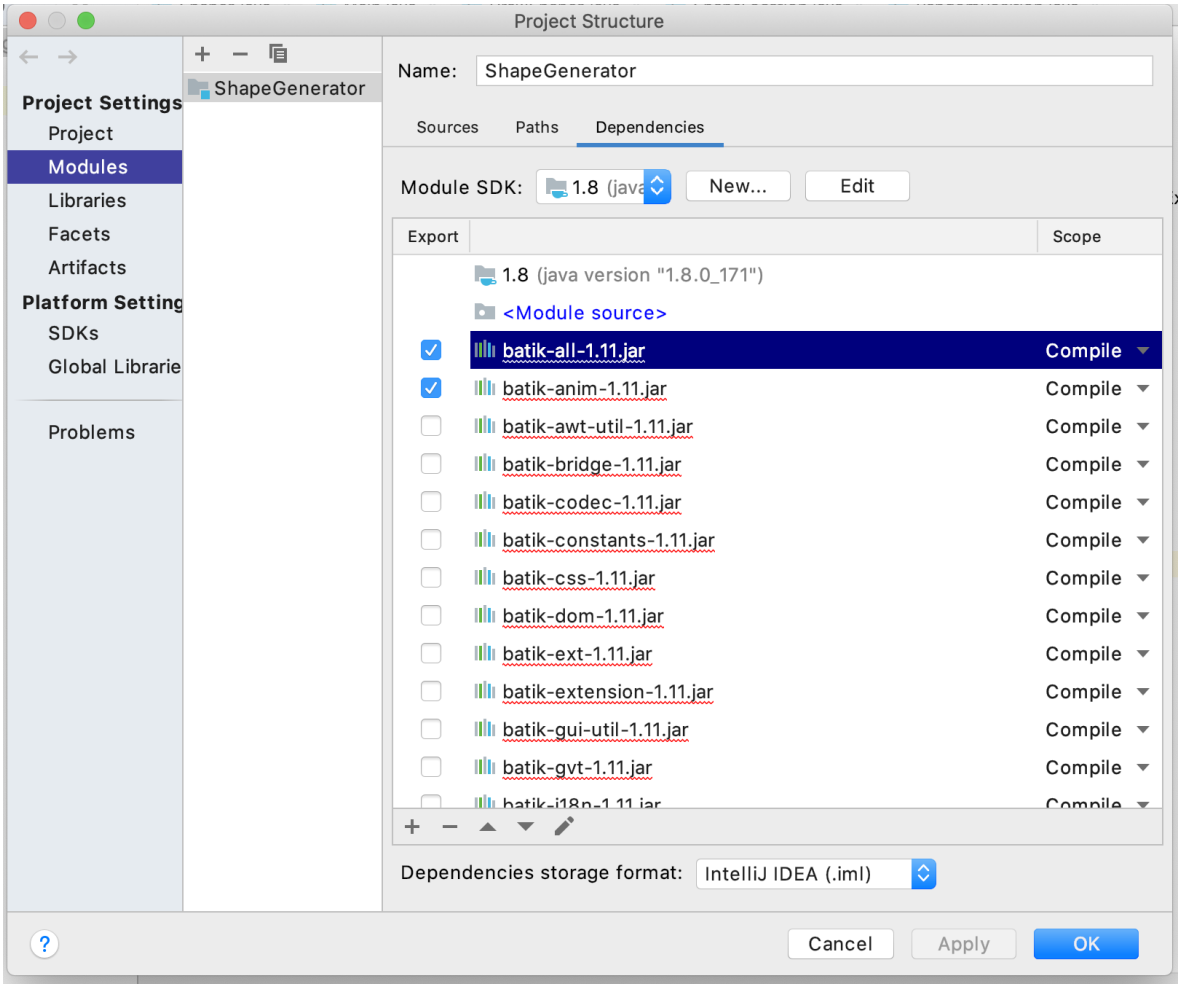

Figure IV

Now since the dependencies are added. We are ready to compile the code. The starting point of the project is the Main.java. Now, build the project and run it from Main.java.

Classes Implementation

ShapeLocation.java

This is the class where the location of the x-axes of the figure and y-axes of the figure are determined from the position. In this we have the following major methods

getRandomFigure() -- String

This method generates the random figure based on the number and generates the output type as string.

findStartingXPosition(String horizontalPosition, String shape) – int

This method generates the starting point of the figure based on the horizontal position and shapes. This takes input as horizontal position, shape and returns the x-coordinates as integer.

findStartingYPosition(String verticallPosition, String shape) – int

This method generates the starting point of the figure based on the horizontal position and shapes. This takes input as vertical position, shape and returns the y-coordinates as integer.

DrawShapes.java

In this class we grab the x-coordinates, y-coordinates and draw the figures to the canvas. DrawShapes class have the following methods.

DrawShapes(Graphics2D g, String side)

This is a constructor which takes the graphics object and a side string which tells the position of the image. This uses drawFiguresAtRandomPosition() method.

DrawShapes(Graphics2D g)

This constructer only takes the graphics object and positions the figure based on selected point. This uses drawFiguresInside() method.

drawMainShape() --void

This method is used for generating the main square container to hold the figures.

drawFiguresAtRandomPosition() --void

This method is used to generate the random figures at the random positions.

drawFiguresInside(String shape, String horizontalPosition, String verticalPosition) -- void

This method selects the shape and horizontal and vertical positions based on the string and generates the figure at that position.

Shapes.java

This is the class we use to generate various shapes based on the count on the positive side figures and the negative side figures and generate svg images from the awt objects using the batik repository. The following methods are implemented

generateShapes(int leftFiguresInside, rightFiguresInside, int figureCount) – void

This method takes the count of the figures on the negative side, total number of figures to be drawn on the positive side and the figure count to add to the name of the file and generates the SVG image of the file in the folder named svgs.

Main.java

This is the starting point of the whole project where we write the count of the images of the object and generate all the images of your requirement.

Analysis

Theorotical Analysis

Let us consider the 2 sets of figures. 6 on the positive side containing 2 figures in each square and 6 on the negative side containing 3 figures in each square. Please find the example below

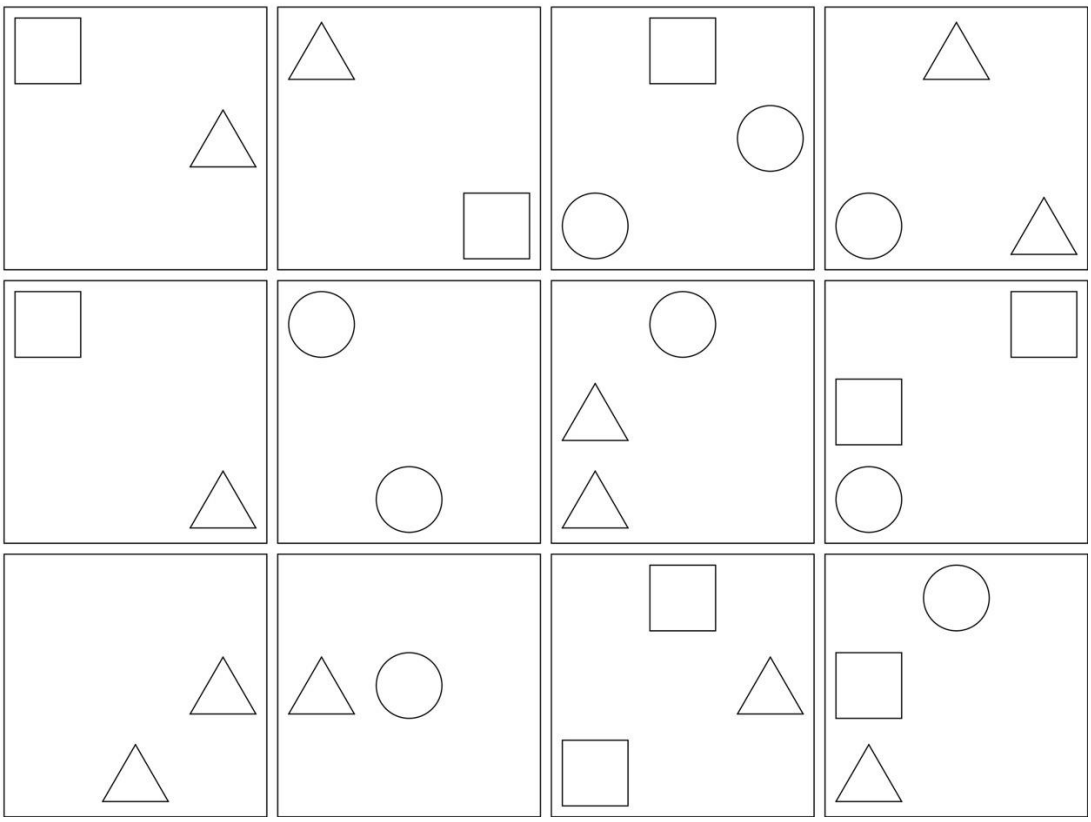

Figure V

In the following example we have six figures on the left side which are first 2 rows and 3 columns. We have 2 images on each of the following square on the positive side whereas we have 3 images on each square on the negative side. This makes the negative side opposes the rule of the positive side making it the bongard problem.

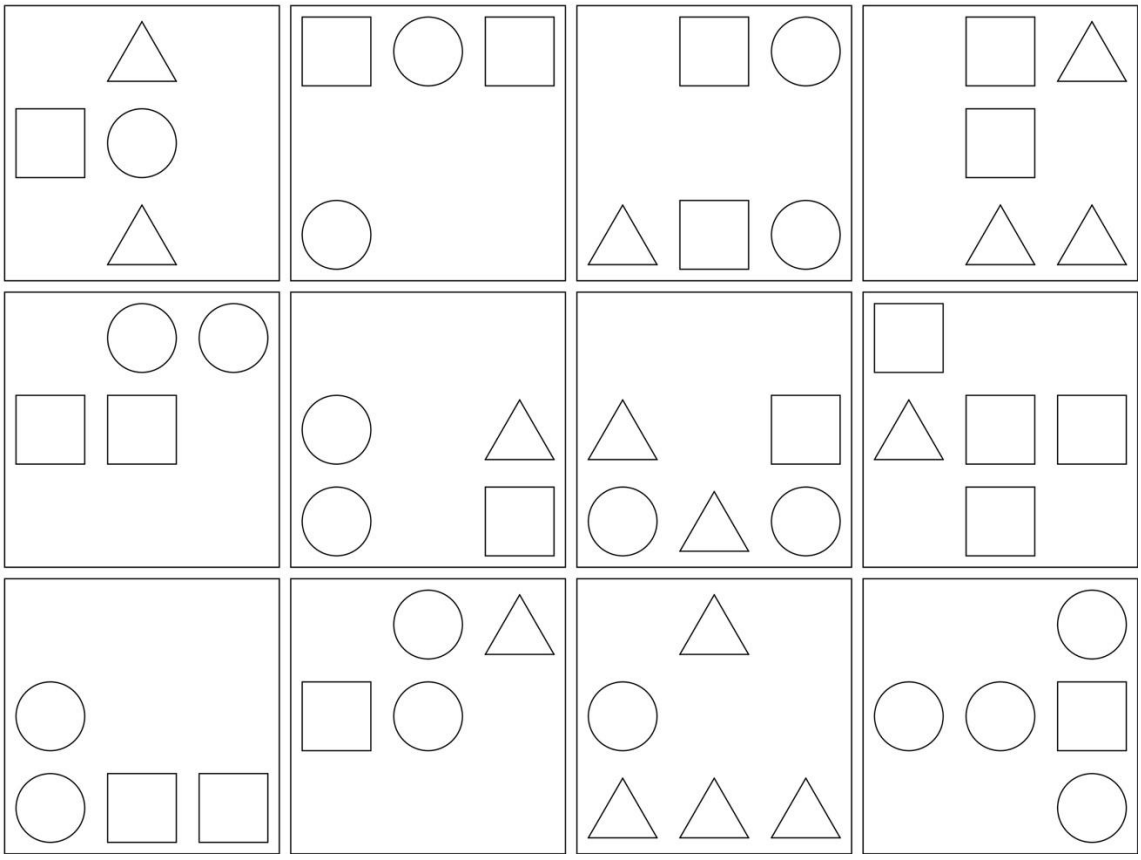

Figure VI

To generate such images we are considering the 3 figures in the program which are a triangle, square and a circle out of 9 combination of positions left, right, horizontal center, top, bottom, vertical center. We calculate it be ${}^{9}C_{3}$ which would be equal to 84 for the positive side, for the negative side we have ${}^{9}C_{2}$ Which would be equal to 36. These many combinations of positions can be selected. Now for the figures for the positive side we have $3^3 = 27$ and for the negative side we

have $3^2 = 9$ figures can be added at those positions. Now we have a total of 84x27 = on the positive side 2268. On the negative side we have 36x9 = 324 in total.

Likewise if we consider the images of 4 on the negative side and 5 on the positive side we have ${}^9C_4$ on left and ${}^9C_5$ on the right. The figure permutations would be 3^4 on left and 3^5 on right. The total number of figure combinations possible would be $3^4 \times {}^9C_4 = 10206$ on the left and $3^5 \times {}^9C_5 = 30618$.

So if we try to generate 1000 samples of each combinations of these images we could generate 324/6 = 54 of these 2 figure square samples without any duplicates on any image but for the 3 figure squares we can generate up to 2268/6 = 378 3 figure square samples. Bur for 4 figure squares we have 10206/6 = 1701 images generated where as for the five figure squares we have 30618/6 = 5103 samples generated without any of the duplicates. So we can calculate the total number of samples without any duplicates generated would be by the below formula:

Equation 1

$$\text{Total non-duplicate figures (d)} = {}^9C_n \times 3^n$$

Where n is number of figures placed in a square.

Practical Analysis

For the generation of the images in the Figure V, 1000 samples are generated that means we get 1000 x 6 =6000 samples of each 4 figure squares and 5 figure squares are generated. Totally by calculating them manually, there was around a total of 12 duplicates for both the figures in the first time. Second time after checking there were no duplicates generated in any of the following. For the respective 3, 6, 7,8 and 9, we never had any duplicates. But for the 10^{th} time there were around 14 duplicates. In most of the cases we have a total of around single digits of images repeating. Since we have higher combination of around 10000 particles getting 6000 repeated is like one thousandth probability of chance for the 4 figure squares and for 5 figures squares the chance is around one thousand chance of repeating.

Trial	Repetitions
1	245
2	198
3	204
4	188

5	192
6	216
7	201
8	237
9	192
10	181

By this we can say that these repetitions happen if we have less combination than the give. But for a single sample we don't have any repetitions so it would be easy to train.

For 2 and 3 figure squares if we try to generate 500 images we have

Trial	Repetitions
1	12
2	3
3	0
4	2
5	6
6	0
7	0
8	0
9	0
10	14

The following formula was used to calculate the probability:

Equation 2

$$P(\text{non-duplicate chance for nth image with k duplicates}) = 1/(d-(n-1+k))$$

The total time it took generate one set of figures is around 5 sec (approx.).

Conclusion and Future work

From the analysis we can infer that there is a very less probability of images repeating with the images at the middle point of the total combinations like (n/2) as per the equation 2 the more the value of d is the greater the chance of repeating. So this generation would be helpful for the implementation of the deep learning for solving the bongard problem in a simplified classification

atleast. I hope this code would be useful for the generation of the shapes for the bongard solver in the future.

References

Khargorgiev, Sergii. (2018). Solving Bongard Problems With Deep Learning.

https://k10v.github.io/2018/02/25/Solving-Bongard-problems-with-deep-learning/

Bongard, Mikhail M. (1970). Pattern Recognition. New York: Spartan Books.

Hofstadter, D. R. (1979). Gödel, Escher, Bach: an Eternal Golden Braid. New York: Basic Books.

Foundalis, H. (2006). Phaeaco: A Cognitive Architecture Inspired by Bongard's Problems. Doctoral dissertation, Indiana University, Center for Research on Concepts and Cognition (CRCC), Bloomington, Indiana.

Mutalik, Pradeep. (2017). Solution: 'Bongard Problems and Scientific Discovery'

https://www.quantamagazine.org/puzzle-solution-bongard-problems-20170628/

László Kozma. (2011). Bongard problems
http://lkozma.net/blog/bongard-problems/