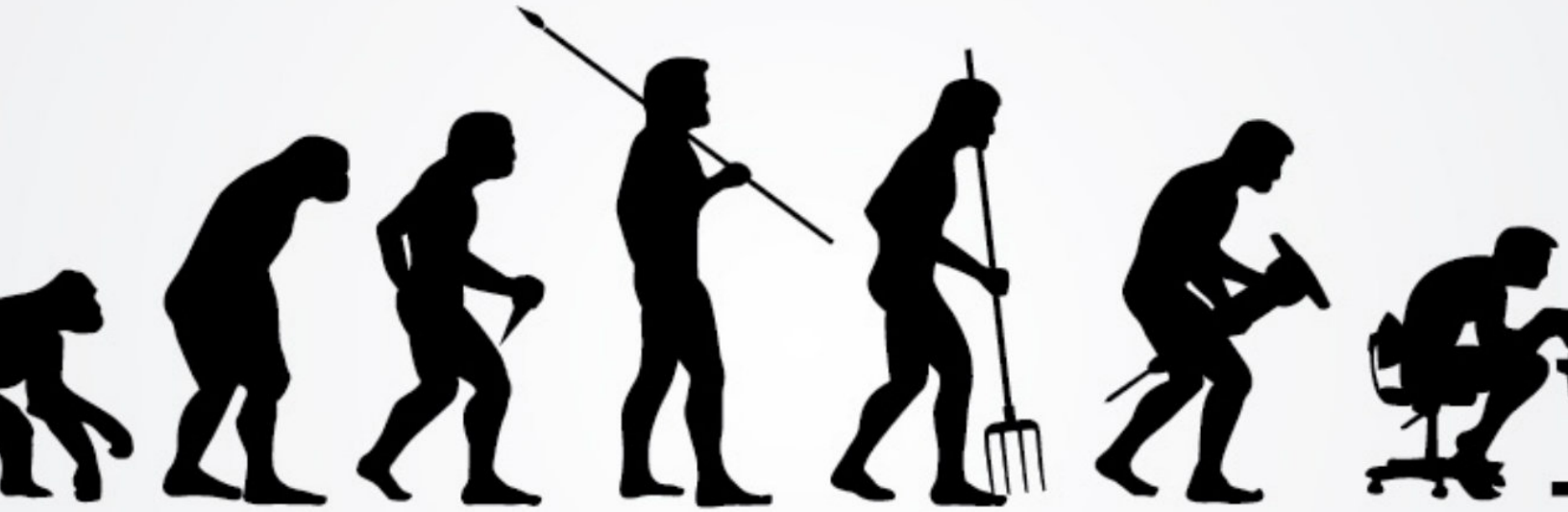
DTU Compute
Department of Applied Mathematics and Computer Science

02122 Software Technology Project

Detecting Structural Breaks in Time Series via Genetic Algorithms

Group 3: Markus B. Jensen, s183816



Contents

1 Introduction

Detecting structural break points is an essential tool in analyzing time series. Structural break points are points on a time series where the pattern of the time series measurements changes in the amplitude. A simple example is shown in figure 1a where the break point is easily detected at $t = 500$. Structural breaks are where the pattern of a time series changes.

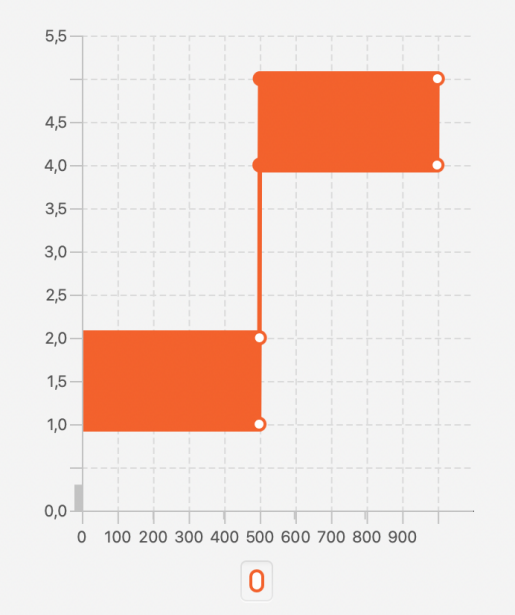
In the example above, the break point is easy to detect. As usual, the real world is far more messy. In figure 1b, which shows the number of patients hospitalized in Denmark due to Covid19, the break points are less obvious. This is where detecting break points is important: Being able to recognize when the pattern changes, so that a pandemic does not run amok or looking for fluctuations in the stock market.

Please find another example than the stock market

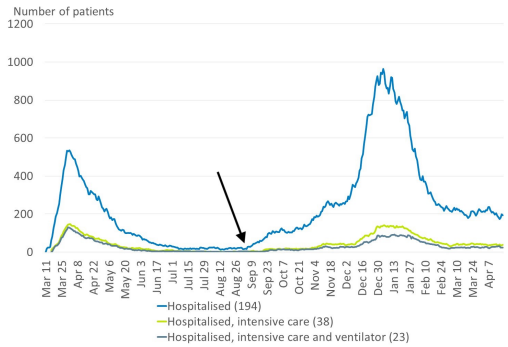
In this project, the break points are found by using a so-called genetic algorithm. This algorithm mimics natural evolution: A number of individuals (solutions that indicate the positions of break points) mutate and mate during generations where survival of the fittest is the rule. In the end, the best solution will (hopefully) have found all the break points.

This project will focus on implementing the algorithm in the programming language Java and make a simple application. The application will give the user the ability to tweak some of the algorithm's parameters and see the break points directly on the time series graph. Further improvements to the speed of the algorithm will also be implemented "under the hood".

Make the left figure more pretty!



(a) Graph with break point at $t = 500$



(b) Number of people hospitalized due to Covid19 in Denmark. The arrow indicates a break point. Source: the Danish regions

Figure 1: Two figures illustrating break points in time series

2 Goals

A prioritized list of the functional goals for the project is listed below. While adding a lot of features seems very appealing, I will in this project focus on usability, stability, and speed. This leads to a somewhat conservative list of potential features, but the quality of the final product will reflect this decision.

1. Implement the algorithm using the rectangle-method from [Doerr et al., 2017] in Java.
2. Visualize two-dimensional time series graphs together with the rectangles produced by the algorithm in a simple graphical user interface (GUI). This GUI will allow user to load a time series data file and see the output of the algorithm on the time series. The user will also be able to tweak certain parameters of the algorithm.
3. Make the algorithm more flexible by allowing the user to alter the values of algorithm-parameters in the GUI.
4. Optimize the runtime of the algorithm with range trees.

A previous goal was also to implement further fitness functions (see explanation for *fitness functions* in section 3). Since this project ended up being a one-man-project, this goal was removed. In stead, a priority is to design the project so that it is easy to implement other fitness methods. This will be discussed further in section 4.

2.1 Non-project goals

As for non-project goals, there are a few:

- Learn the Maven project structure for Java. While previous courses have dealt with Maven a little bit, it has never been fleshed out. It seems to be a structure that is widely used and thus a good system to learn.
- Working on big project. This is the first big project I am working on. A big focus here is on the project management, report writing workflow and keeping track of sources in a bibliography. Especially the report writing workflow can become crucial, as I have a tendency to postpone it to the very last minute. Here, I will make it a part of my weekly work.

My ambition level is quite high; I am a perfectionist to the core. While I will attempt to keep the perfectionism to a minimum, I like working on bigger projects and making it work well. This will probably result in me working a lot on this project, purely because it will be fun, and I like improving my less-than-optimal solutions. I am aware that this course is only 5 ECTS points and will thus keep track of my hours spent on the project as to not overdo it.

3 Terminology

The terminology used in this report is specified below. The terminology differs a bit from [Doerr et al., 2017]. It takes inspiration from other sources ([Thede, 2004, Point,]) and follows a more biological narrative.

Individual An individual consists of a solution string. It is one possible solution to the problem.

Genome An individual consists of a genome. A genome is an array of genes. The genome *is* the solution string.

Allele A gene's value is called an allele. In this project, the allele is thus responsible for holding the information of whether or not a certain gene is a break point. An allele is the value at a certain gene in the genome/solution string.

Population A group of individuals.

Fitness function A function that measures how well the solution from an individual fits with the data. This can be any function relevant for a particular problem.

Parent and offspring individual Since genetic algorithms mimic evolution, procedures must include parent and offspring individuals. A parent individual is a parent to the offspring individual, meaning that any procedure must take one (or two) parent individual(s) and the procedure creates an offspring.

Crossover (Procedure) Crossover-procedures takes two parent individuals and creates an offspring from the genomes of the two parents. This project will incorporate *one-point crossover* and *uniform crossover*.

One-point crossover extracts the genes in the interval 0 to a random gene $i-1$ from the first parent. Then, it appends the genome with the genes i to the last gene $n-1$ from the second parent. Thus, the offspring's genome consists of the first i genes from parent one and $n-i$ last genes from parent two.

Uniform crossover is more simple. For each gene in the offspring's genome, there is a fifty-fifty chance whether it will be the gene from parent one or the gene from parent two.

Mutation (Procedure) Just like the corona virus, the genome of an individual can mutate. It means, that for any gene, there is a random chance that the allele will change. In this case, the change will be either creating break points or removing them.

4 Problem analysis and design

The main challenges are already hinted at in section 2. All these goals pose different problems. These problems (and especially their solutions) is best analyzed when also discussing the chosen design.

4.1 The Basic, Simple Algorithm

This building blocks for the basic, simple algorithm is this: An individual X is represented as a string with the same length as the time series Y with length T, $|X| = T$, and Y itself is represented as an array of T observations. The data structures themselves are very simple.

The design of the algorithm is found in the handed out pseudo code. Thus, discussing the implementation and functionality of this code seems redundant. Especially given that all the procedures are explained in [Doerr et al., 2017].

What *is* interesting about this design is the generality of it: A genetic algorithm is a very general tool. In the most simple of cases, the individuals manipulated are simple solution strings containing 1s and 0s - or other simple characters. The individual is thus a simple entity, and the value of it only becomes meaningful in the context of a given problem.

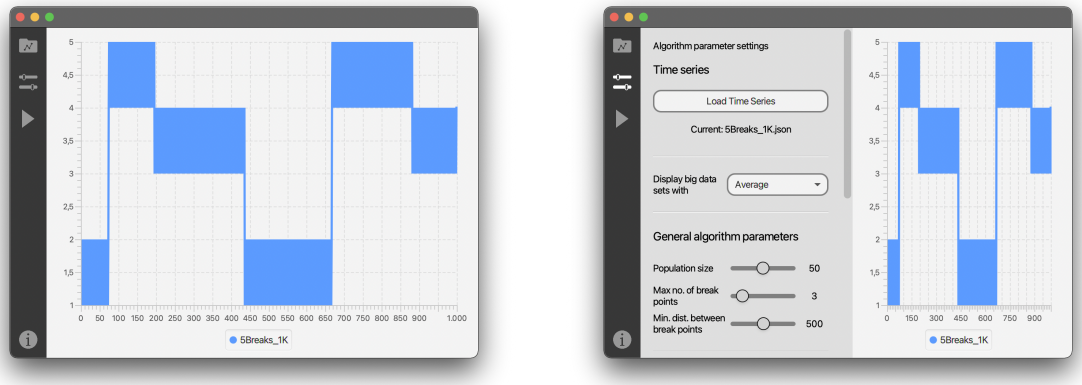
In the context of finding structural break points with the rectangle fitness method, the genome of the individuals contains two types of alleles: `0` meaning no break point and `1` meaning a break point. Thus, the genome has T genes, where each gene can contain one of two types of alleles.

This general and simple approach is alluring due to its simplicity. This is especially the case within the biological narrative of the genetic algorithm: A genome of fixed length and a small set of alleles matches the narrative. But for this problem, where the number of breakpoints is potentially miniscule compared to the length of the time series, there is a more efficient approach, see section 4.3.

4.2 Graphical User Interface, GUI

For the design of the GUI, there are two key priorities: 1) Displaying the time series with break points and 2) being able to configure algorithm parameters.

While the implementation of displaying the graph and showing break points was incredibly tricky, see section 5.1, the design itself is fairly easy: Show a graph and draw some rectangles/lines. The focus here is to make the graph area take up as much of the GUI as possible. This is achieved by being able to toggle the settings pane, see figure 2.



(a) GUI not showing the settings pane

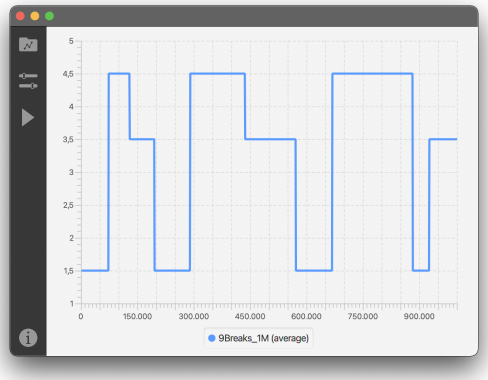
(b) GUI showing the settings pane

Figure 2: GUI with and without settings pane toggled

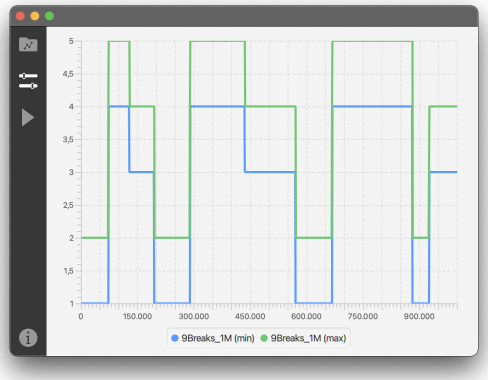
For large time series, it is unnecessary and inefficient to display all measurements. In stead, a maximum of 1000 points are displayed on the graph. The number of graph points per measurement n_p is given by:

$$n_p = \left\lceil \frac{T}{1000} \right\rceil \tag{1}$$

Each point in the graph thus contains multiple measurements from the time series. These multiple points can be displayed in two ways: Taking the average of the measurements (figure 3a) or displaying the minimum and maximum (figure 3b).



(a) Each point on the graph is the average of 1000 observations



(b) For each 1000 observations, the min and max is displayed

Figure 3: The graph showing a time series with $T = 10^6$ observations

For tuning the parameters, the design for this GUI has been sliders, buttons and menus. These ways of interacting are very bulletproof as the user cannot do anything wrong with the controls. (This is very much not the case with text fields where a lot of error-handling is needed for even simple integer fields). The downside to especially sliders is that they limit the user to a certain range. The controls are reliable but not very flexible.

A lot of the algorithm parameters can be tuned. This is definitely not all parameters. The ones chosen go into these categories:

General algorithm parameters Things like the size of the population, the maximum number of break points, and the minimum distance between the break points.

Probability values Setting the probability for each of the genetic algorithm procedures: mutation p_{mu}, one point crossover p_{opc}, and uniform crossover p_{uc}. Note that $p_{mu} + p_{opc} + p_{uc} = 1$. To solve this, the slider are connected: First, p_{mu} is set, and the two other sliders are set to

$$p_{opc} = p_{uc} = \frac{1 - p_{mu}}{2}$$

Then p_{opc} is adjusted. When adjusting, $p_{opc} \in [0, 1 - p_{mu}]$ and $p_{uc} = 1 - p_{mu} - p_{opc}$. The slider for uniform crossover can never itself it moved, only the two other sliders can move it. In summation, first change the mutation slider, then change the one point crossover slider to dial in all three probabilities.

Fitness The user can change the fitness method (although only one exists now) and tune the α-parameter for the penalty term in the fitness function.

Besides this, the GUI can load time series data files using the system's own file viewer and the algorithm can of course be run with the algorithm.

4.2.1 Model-View-Controller

The design of the GUI is made with the Model-View-Controller pattern in mind. The model is of course the algorithm itself. There is one entity in charge of running the algorithm and keeping charge of the parameters. The view is simply the look of the GUI. The placement of everything and the styling itself. Lastly, the controller takes all inputs from the user and delivers the relevant information to the model.

In this case, the model is a Java class `BreakPointAlgorithm` which is the brain of the system and the class in charge of running the algorithm itself with all the parameters. The view is done using FXML files and styling with CSS. Lastly, the controller is another class which communicates directly with the model: Listeners are placed on all the controls which sends updated parameter values to the model.

4.2.2 Observer pattern

The observer pattern is only included at a small scale: The controls in the controller have listeners that updates the model when the value is changed. Besides that observer pattern is not implemented here as the application is very small. A more simple approach is simply to allow the controller and the business logic to exchange information. The business logic can, in this application, never change if no changes are performed in the GUI. This makes observer pattern rather useless.

4.3 Optimizing the algorithm

For the optimization in this project, two key areas have been optimized:

1. The genomes no longer contain genes with non-break-point-alleles. The genome is in stead a list-structure G. At each gene g is an allele a_g, $G(g) = a_g = (b_g, B_g)$. The value b_g is itself an index representing the index of the break point, $b_g \in [0, T]$. The term B_g is a more abstract value containing information relevant for the fitness model.

2. The time series observations are stored in a *range tree/interval tree*. While setting up the range tree will itself take longer than setting up a simple array, getting minimum and maximum values in index intervals is much faster.

Making these changes means altering the genetic algorithm procedures and implementing the range tree data type.

4.3.1 Altering genetic algorithm procedures

One point crossover The most simple to adapt is the `one point crossover procedure`: A random index $i \in [1, T-1]$ is chosen as the splitting point. From the first parent P, the offspring O gets all genes p with alleles $a_p = (b_p, B_p)$ where $b_p < i$. From the second parent Q, O gets all genes q with alleles $a_q = (b_q.B_q)$ where $b_q \geq i$. The time complexity is $O(Size(P) + Size(Q))$.

Mutation The mutation procedure is also simple, or at least *made* simple. A single index i is randomly selected in the interval $[1, T-1]$. With a probability p_m a break point is placed at i. Else, with a probability $1 - p_m$, the point will become at non-break-point. This means that if a break point already exists with index i, the break point is removed. If no break point

exists at i, then no action is performed. The time complexity for this mutation is $O(T^{-1})$ which is a lot better than going through T elements as in the handed out pseudo code.

For the mutation procedure it might be possible that there exists multiple break points at one index. This is allowed and in stead affects the fitness score. If a break point is removed at index with multiple break points, only one of the break points is removed.

Explain the fitness thing somewhere else and link to that from here

Uniform crossover The final procedure, uniform crossover, is a bit more complex in the new context. The uniform crossover takes two parents P and Q. In a single for loop, the two genomes are both traversed by maintaining two iteration parameters. i traversing P and j traversing Q. The two parameters are initialized to $i = j = 1$. The break point indexes at the genes p_i and q_j are compared: The allele storing the lowest break point index is added to the offspring with a fifty-fifty chance. The iteration parameter for that genome is incremented. If the two break point indexes are the same, a coin toss decides which allele is added to the offspring. Both counters are incremented afterward. This is repeated until both genomes are traversed. The time complexity for this is $O(Size(P) + Size(Q))$.

Figure 4 shows an example. The two parents P and Q have three and two break points in the interval respectively. The diagram contains four scenarios:

1. For $i = j = 1$, the break point index $p_1 < q_1$. Thus, the allele storing p_1 is added to the offspring with a fifty-fifty chance. Now i is incremented so $i = 2$
2. $q_1 < p_2$. Allele storing q_1 is selected and j is incremented $j = 2$.
3. $p_2 < q_2$: Allele storing p_2 is selected and i is incremented to $i = 3$.
4. $p_3 = q_2$: A random allele of the two is selected and both i and j are incremented to $i = 4$ and $j = 3$.

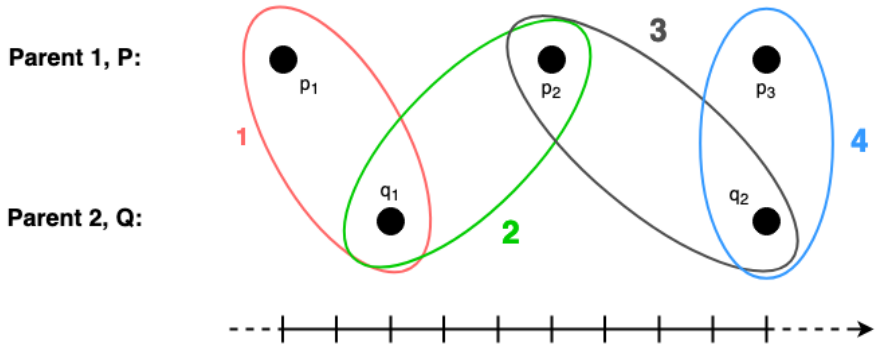


Figure 4: A diagram showing multiple cases when calculating uniform crossover.

4.4 Range tree

A range tree is a binary tree. Each node contains information about the following:

- An interval $[a, b]$ where $a \leq b$.

- A minimum min and maximum max value.

For a given node, this means that for $a \leq b \leq T$, the interval $[y_a, y_b]$ has the minimum value min and maximum value max.

For each node where $a \neq b$, the left child has the interval $[a, \lfloor (a+b)/2 \rfloor]$ and the right child has the interval $[\lceil (a+b)/2 \rceil, b]$. The maximum and minimum of the node is the combined maximum and minimum of the two children. For leaves, the range is a singleton range $[c, c]$ and the minimum and maximum values are also the same. The minimum and maximum values for each node is thus calculated recursively starting from the leaves and moving up.

Construction the range tree starts at the root. The root has the interval $[0, T-1]$. The maximum and minimum values are the combination of the maximum and minimum of the two children which is calculated recursively. The left child get the index $[0, T/2]$ and the right child gets the index $[T/2+1, T]$. This continues until the leaves where the minimum and maximum is set, resulting in all the previous nodes now being updated with their minimum and maximum.

Write about getting the min and max in a range

5 Implementation

5.1 Displaying the graph and break points

6 Test

6.1 Running time

For testing running times, a simple class `runTimeTimer` was made. It has two methods `start()` and `stop()`. `start()` must be called before the code chunk to be examined. `stop()` must be called just after. The running time will appear in milliseconds in the console.

7 Project Management

8 Conclusion

References

[Doerr et al., 2017] Doerr, B., Fischer, P., Hilbert, A., and Witt, C. (2017). Detecting structural breaks in time series via genetic algorithms. *Soft Computing*, 21(16):4707–4720.

[Point,] Point, T. Genetic algorithms.

[Thede, 2004] Thede, S. (2004). An introduction to genetic algorithms. *Journal of Computing Sciences in Colleges*, 20.