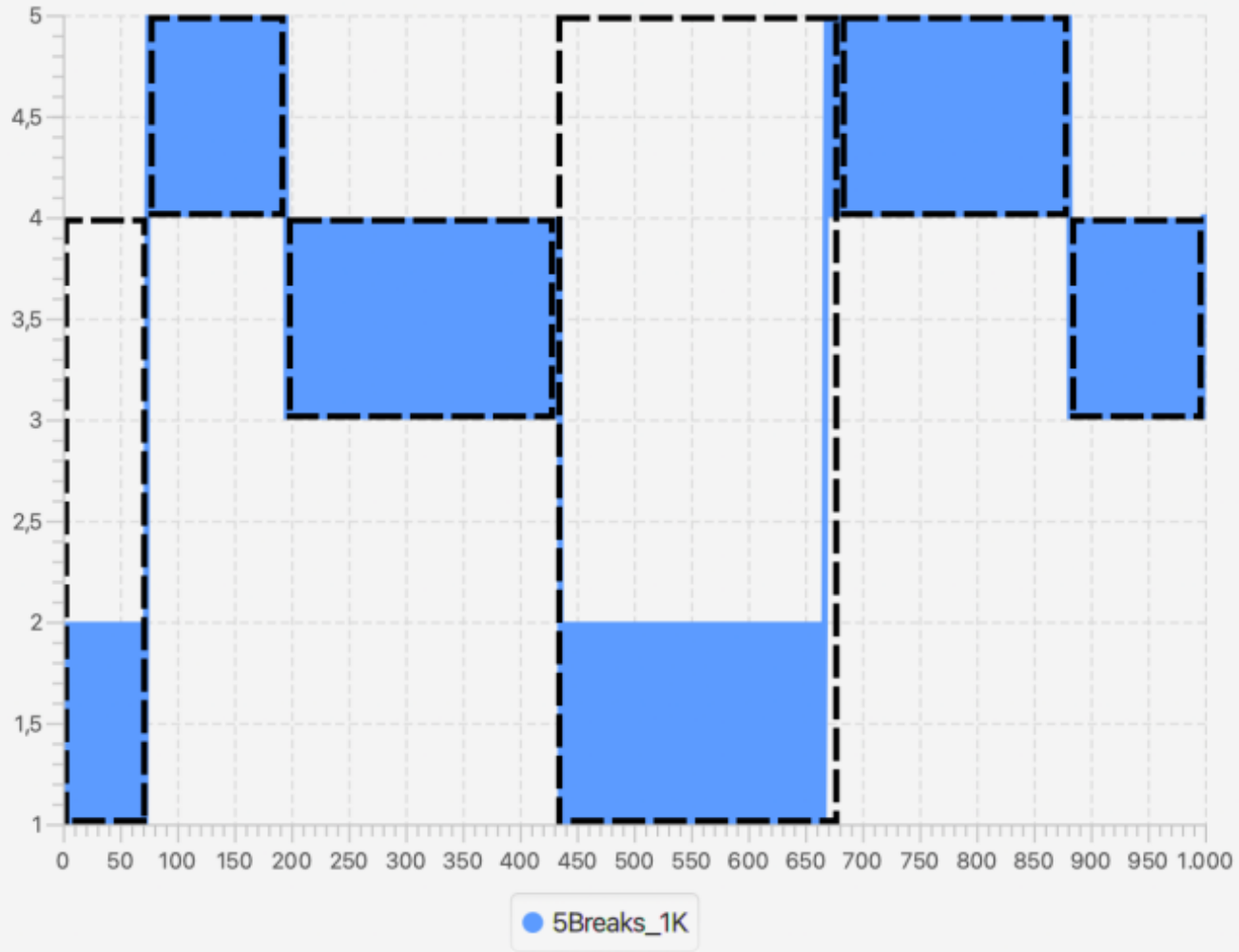
DTU Compute
Department of Applied Mathematics and Computer Science

02122 Software Technology Project

Detecting Structural Breaks in Time Series via Genetic Algorithms

Group 3: Markus B. Jensen, s183816



Contents

1 Introduction **1**

2 Goals **2**
2.1 Non-project goals . . . 2

3 Terminology **3**

4 Problem analysis and design **4**
4.1 The Simple Algorithm . . . 4
4.1.1 Simple approach reflections . . . 4
4.1.2 Break Point Algorithm Design . . . 4
4.2 Graphical User Interface, GUI . . . 5
4.2.1 Model-View-Controller . . . 7
4.2.2 Observer pattern . . . 8
4.3 Optimizing the algorithm . . . 8
4.3.1 Altering genetic algorithm procedures . . . 8
4.3.2 Range tree . . . 9

5 Implementation **11**
5.1 Genetic algorithm Implementation . . . 11
5.2 Fitness methods . . . 12
5.3 Loading the time series and range trees . . . 13
5.3.1 Non-implemented optimization for range trees . . . 13
5.4 The structural breaks algorithm . . . 15
5.5 GUI: Displaying the graph and break points . . . 15

6 Test **19**
6.1 Components . . . 19
6.2 Running time . . . 19
6.2.1 Optimization gains . . . 19
6.3 The algorithm . . . 20

7 Conclusion **21**

A Code snippets **22**

B Class diagrams **24**

C Optimized range tree query algorithm example **25**

References **27**

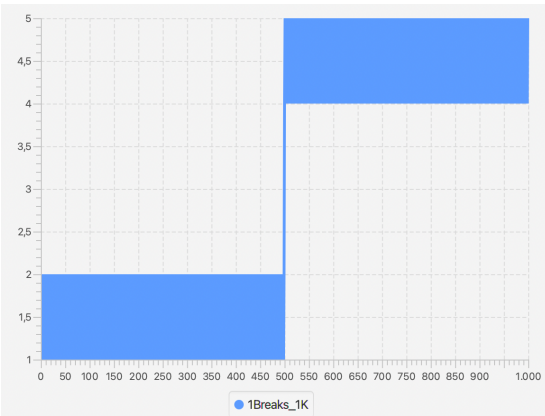
1 Introduction

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.

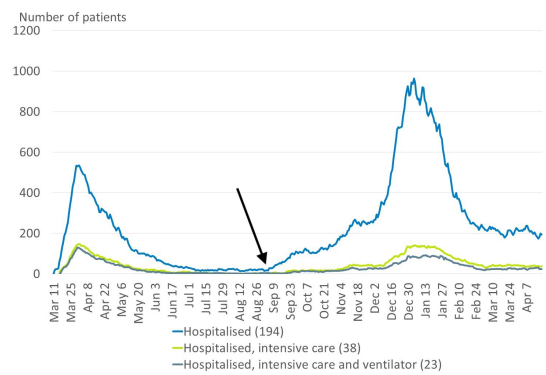
In the example above, the break point is easy to detect. As usual, the real world is far more messy. In figure 1b, which shows the number of patients hospitalized in Denmark due to Covid19, the break points are less obvious. This is where detecting break points is important: Being able to recognize when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

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".



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



(b) 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. 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.
- Write a big report in english. While I have made some assignments in english, this will be the first big project to be entirely in english.

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.

Break point Discussed and explained in section 1. Also refereed to as *structural breaks*.

Allele An allele is an entity carrying the information of break points. An allele stores the information about the location of a break point and the model information relevant for a given fitness method. (See explanation for fitness method a bit further down).

Genome A genome is an array of alleles. A *gene* indicates the index of an allele in the genome. A genome is analogous to an array of integers like so:

```
intArray[index] = integer
genome[gene] = allele
```

Individual An individual is a possible solution to the problem at hand. The individual consists of a genome and possible other information relevant for the genetic algorithm.

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.

Fitness method The fitness function is dependant on the fitness method. The method in this project is the rectangle method mentioned in [Fischer and Hilbert, 2014] and [Doerr et al., 2017].

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 Simple Algorithm

The first goal for this project is to implement the algorithm for detecting structural break points as mentioned in [Doerr et al., 2017]. While that article quickly delves into optimizing the algorithm, the first priority for this project is having a working algorithm before a fast algorithm. Thus, the simple algorithm is the first to be implemented.

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

4.1.1 Simple approach reflections

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 a few different 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: `*` meaning no break point and `!` 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.

The problem with this approach is speed: The number of breakpoints is potentially miniscule compared to the length T of the time series. Going through T elements is very inefficient if the number of elements of interest is much less. See section 4.3 for the optimized version.

4.1.2 Break Point Algorithm Design

For the design of the algorithm, the design is split up into three factions: *genetic algorithm*, *break point algorithm*, and *fitness model*. In this way, the genetic algorithm becomes its own more general tool and the break point algorithm, which uses the genetic algorithm, makes the use more specific.

Saying the genetic algorithm part is a general tool is a stretch. This generality is discussed in section 4.1.1 and the notion turns almost nonsensical with the optimizations discussed in section 4.3. Here the genetic algorithm elements are completely knowledgeable of the break points and their meaning. Thus, the genetic algorithm seizes to be general. *But* in the design, these two factions are split apart.

In the genetic algorithm part, the individual is represented as a data type which can contain the location of break points and the model information for the fitness model. The genetic algorithm part is also responsible for the procedures generating offsprings. The individual might be aware of its fitness value, in any case the fitness model itself must be able to calculate the fitness of an individual. The comparison between individuals mostly takes place in the scope of a population which can return the individual with the highest fitness and replace the individual with the lowest fitness. The population also selects the two individuals for procedures.

The break point algorithm part is in charge of running the algorithm to find structural breaks. This part is mostly in charge of manipulating all the elements from the genetic algorithm part to get an outcome that makes sense. This means choosing randomly which procedure to generate a new offspring with and keeping charge of communicating with a population: Getting the two parents for procedures and replacing the offspring with the lowest fitness in the population.

While this project only *implements* one fitness method, the design opens up for further fitness methods. The fitness method must be its own element able to calculate the fitness of an individual and display the break points of the fittest individual. Thus, the fitness is a middle-man between the genetic algorithm and the break point algorithm.

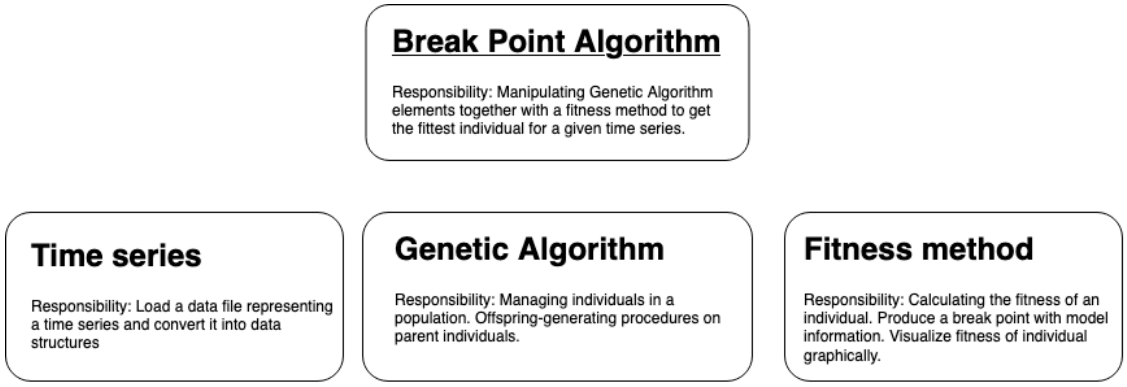


Figure 2: A simple structure for the design and interface ideas for the program (without GUI)

A fourth element, outside the strict realm of the algorithm, is *time series*. For the algorithm to work it must have something to work *on*. A time series is loaded here and the break point algorithm then uses the time series to make sense of the individuals, ie. calculating the fitness of an individual in relation to a given time series.

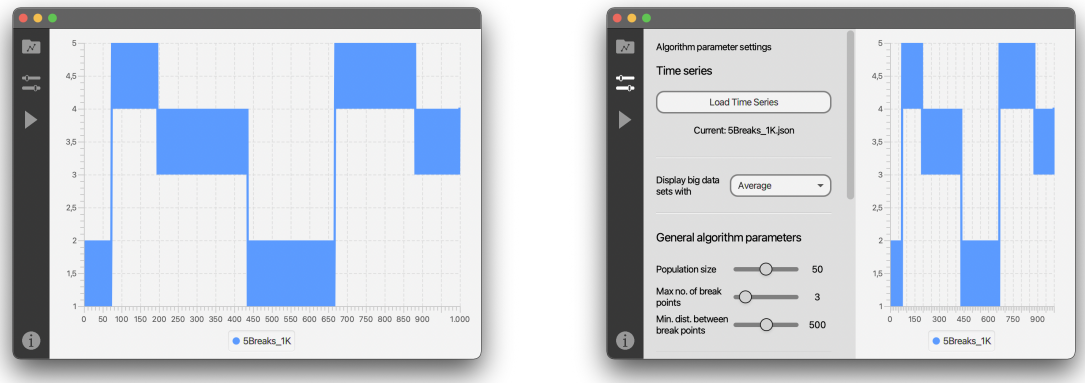
These four factions are illustrated in figure 2.

4.2 Graphical User Interface, GUI

With the simple algorithm taken care of, the results needs to be displayed. 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 5.5, the design itself is fairly easy: Show the time series as a graph and draw some rectangles/lines to indicate break points. 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 3.



(a) GUI not showing the settings pane

(b) GUI showing the settings pane

Figure 3: 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:

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

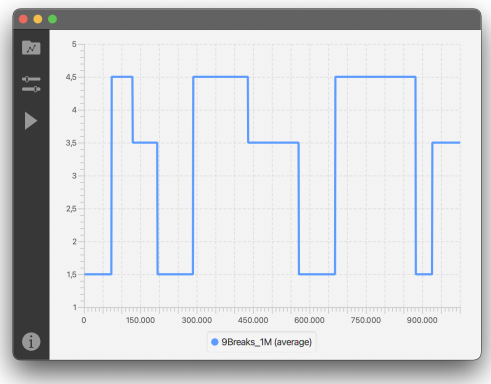
Each point in the graph thus contains multiple measurements from the time series. These multiple points can be displayed in two ways: Taking the average of the measurements (figure 4a) or displaying the minimum and maximum (figure 4b).

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.

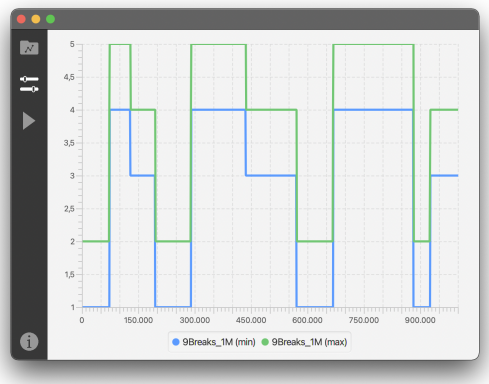
A lot of the algorithm parameters can be tuned. This is definitely not all parameters. The ones chosen go into these categories:

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



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



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

Figure 4: The graph showing a time series with $T = 10^6$ observations

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 $O(|P| + |Q|))$.

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. This is dealt with using a penalty term, discussed in section 5.2.

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 $O(|P| + |Q|)$.

Figure 5 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 (added with fifty-fifty chance) 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 $j = 3$.

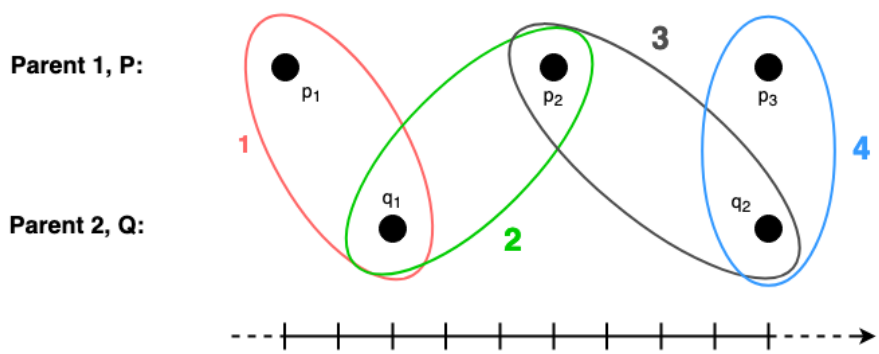


Figure 5: A diagram showing multiple cases when calculating uniform crossover.

4.3.2 Range tree

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

- An interval/range $[a, b]$ where $a \leq b$.
- A minimum min and maximum max value.

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, $min = max$. 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 $[0, \lfloor T/2 \rfloor]$ and the right child gets the index $[\lfloor T/2 \rfloor + 1, T]$. 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. Setting the values in the leaves is simple: The range for leaf nodes is singleton and thus $min = max = T[a] = T[b]$.

Constructing the tree from an array of T observations takes $O(T \log T)$ time. For large observation sets, this will slow down the algorithm a bit when initializing a time series. But the payoff is huge: Finding the minimum and maximum value inside an interval is much faster. In [Fischer and Hilbert, 2014] the algorithm for finding min and max is explained. Here it is also concluded that the running time for a query is $O(\log T)$. This is much faster (especially for bigger time series) than $O(T)$ which the simple algorithm for finding min and max takes.

See time comparisons in section 6.2.1.

5 Implementation

The application is written in Java. Given that Java is an object-oriented language, the implementation and design choices reflect this.

5.1 Genetic algorithm Implementation

The smallest component in a genetic algorithm is the allele. For the final implementation, the allele class contains two fields: `index` representing the index of a break point and `methodBreakPoint` being of an abstract object `BreakPoint`. A break point is decided by the fitness method and thus this field needs to be rather flexible. This in turn means that each fitness method needs its own. More on fitness methods in section 5.2.

The alleles is stored in the genome. The genome class has one field which is an `ArrayList` of alleles. Each index in the genome is instead called a *gene* to fit the narrative. Ie. there is an allele at a gene in the genome.

The genome makes sure the alleles are always in ascending order, sorting the break point indexes. This makes calculating fitness and displaying the break points much easier.

An individual consists of a genome, a fitness value and list of *fitness nodes*. More on the fitness nodes in section 5.2. The fitness value is a measure of how well the individual matches with a given time series. This could cause trouble if the same individual is mathed against multiple time series. This is never the case and will never be a problem.

The individual always has two breakpoints with indexes 0 and T respectively. The individual also has functionality like adding and removing break points. The individual can also tell if it has an allele with a certain break point index. Here, the method `geneOf` from the genome class is used. This method loops through all alleles in the genome and returns the gene containing the first allele matching a condition. The condition itself is specified through a `Predicate` allowing for lambda notation.

```
public boolean breakPointAtIndex(int breakPointIndex) {
    // geneOf returns > 0 if an allele a has the break point index
    return genome.geneOf(a -> a.getIndex() == breakPointIndex) != -1;
}
```

where `a` means an allele.

All the individuals are stored in a population. A population is a simple array of individuals. It is able to retrieve the fittest individual and replacing the least fit individual with a new individual. At one point, the population class stored the individuals in a *priority queue*. Although faster for replacing the least fit individual, the code itself became more convoluted. Since the population can contain at most 100 individuals, the gain in speed did not seem that important.

The population class is also responsible for initializing itself with a number of random individuals as in the pseudo code. Here, it also assigns the fitness to the individuals.

See figure 6 for a class diagram of the elements in the genetic algorithm part/package.

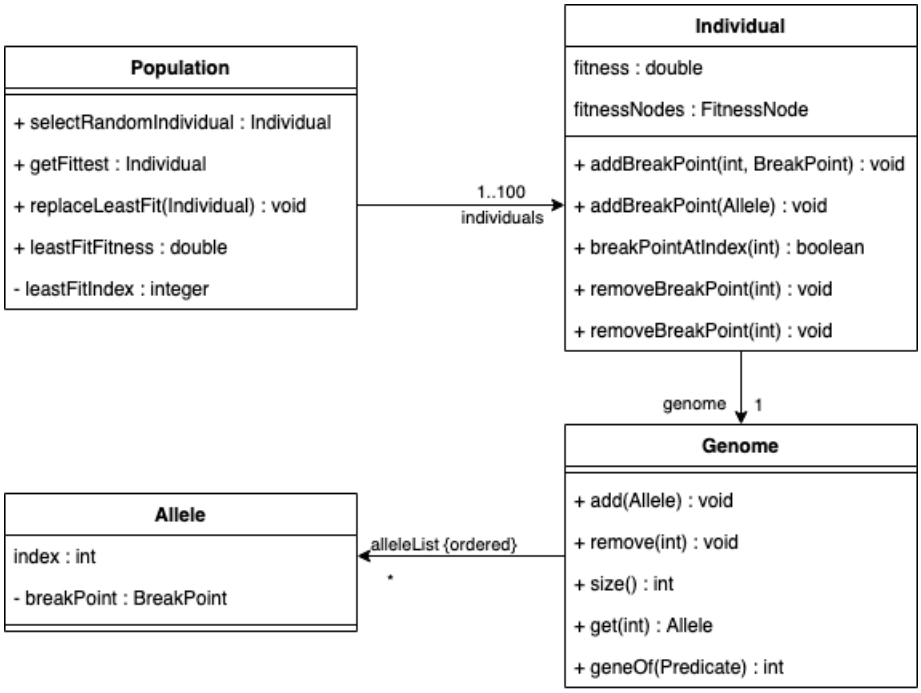


Figure 6: Class diagram for the genetic algorithm elements

5.2 Fitness methods

For this project, only the rectangle fitness method has actually been implemented. But the rectangle fitness class is based on an abstract fitness class. This means that new fitness methods can easily be implemented as sub-classes of the abstract class. All fitness classes must have the following following:

fitnessOf A method for calculating the fitness of an individual given a time series.

newBreakPoint Each fitness method has its own kind of break point containing some information. This is an object of the `BreakPoint` object and must also be unique for each model.

getModelCode The model code is simply a string for the name when choosing the fitness in the GUI.

getNodes When drawing the fitness in the GUI, each fitness method must have its own way of displaying break points. For rectangles, this is drawing rectangles on the graph, see section 5.5. A `FitnessNode` is an (abstract) object which stores some information relevant for drawing on the graph. Each fitness method must have its own kind of `FitnessNode`.

For the fitness, both penalty terms mentioned in [Doerr et al., 2017] are included. The penalty

term $p(k)$ is:

$$p(k) = \begin{cases} 1/k & \text{if } k_{max} = 0 \\ \max\left\{0; \frac{k_{max}-k+1}{k_{max}}\right\} & \text{if } k_{max} > 0 \end{cases} \tag{2}$$

where k is the current number of break points and k_{max} is the maximum number of break points, set by the user. The other penalty term is for the number of break points violating a the minimum distance between break points. Here, the following factor is used:

$$1 - \frac{k_v}{k} \tag{3}$$

where k_v is the number of break points violating the minimum distance.

In all, the fitness function $f(X)$ is:

$$f(X) = 1 - \frac{k_v}{k}(g(X) + \alpha p(k)) \tag{4}$$

where $g(X)$ is the fitness of an individual X determined by a given fitness model. In this project, only one fitness method is implemented. That is the rectangle method from [Doerr et al., 2017] and [Fischer and Hilbert, 2014].

5.3 Loading the time series and range trees

The time series data files are stored in the *JSON* format. To load these, Googles Maven package *JSON.simple* is used. That package has a great `get`-method to retrieve information under a given string name, ie. `get("observations")` or `get("time")`.

An issue with the package was encountered when the data files were large, around 500 MB. (100 MB files work fine). The JSON reader would simply not be able to go through the file and crash after a few minutes. Thus, if the file size exceeds 120 MB, an error is thrown and shown to the user, see figure 7b. 120MB is a conservative limit but it should ensure no errors.

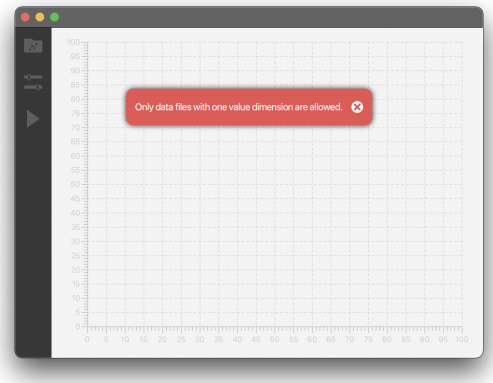
For this project, when reading a data file with more than 1 dimension, an exception is thrown and shown, see figure 7a. For all data files with 1 dimension, the files are read easily as JSON.simple handles the files very much like arrays.

When all the data is loaded, the observation values are read into a range tree. This is a custom data structure implemented in Java. The range tree is implemented as mentioned in section 4.3.2. The code for building the tree and making queries can be found in appendix A.

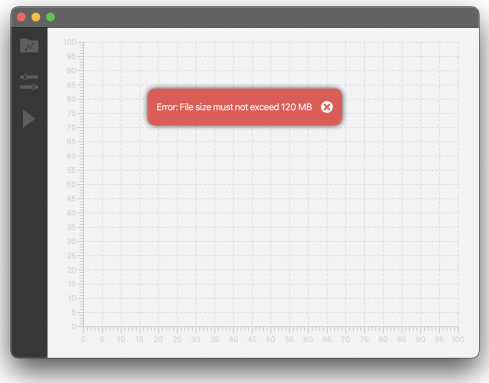
In the implementations above, a `MinMax` object is used. This is a simple object to store a minimum and a maximum value.

5.3.1 Non-implemented optimization for range trees

The run time for the range tree query algorithm can be optimized. As discussed in section 4.3.2, the run time of the algorithm is $O(\log T)$. This must also be the lower bound $\Omega(\log T)$ as the the



(a) Error popup when loading time series with multiple dimensions



(b) Error popup when loading file with size ¿ 120 MB

Figure 7: Errors to pop up when loading a time series with GUI

algorithm must always traverse the height of the tree $\log T$. While the run time of $O(\log T)$ will not be optimized, the lower bound will.

The new algorithm is based on the following principle: A node's minimum and maximum values are the combined values of the two children's values. Thus, if the query range completely matches the current node's range, the minimum and maximum value of *that* node can be returned as the result. This is in stead of going to the leaf nodes having the singleton ranges of the two interval boundaries. For the new algorithm, the mergability property will also be used.

The new query algorithm must abide by the following rules where the algorithm is called with a query range and starts at the root:

- For a given range query, there must be two paths down the tree, left and right. In the beginning, these paths can be identical.
- For a query range $[x, y]$, at a node with range $[a, b]$, the path will continue (only) left if $a \leq x \leq y \leq \lfloor (a+b)/2 \rfloor$ or (only) right if $\lfloor (a+b)/2 \rfloor + 1 \leq x \leq y \leq b$. This rules applies both the paths are identical and after they have split up. The query range is kept the same going into the child node.
- Given a query range $[x, y]$ and at a node with range $[a, b]$ where $x \in [a, \lfloor (a+b)/2 \rfloor]$ and $y \in [\lfloor (a+b)/2 \rfloor + 1, b]$ the two paths must split up. The left path goes into the left child with query range $[x, \lfloor (a+b)/2 \rfloor]$ and the right path goes into the right child with query range $[\lfloor (a+b)/2 \rfloor + 1, y]$.

  The algorithm combines the minimum and maximum values from the left path and the right path.

  (Note that it is the *query range* being updated in either path, not the range of any node).

- Given a query range $[x, y]$ at a node with range $[a, b]$. If $x = a$ and $y = b$ then the minimum and maximum values of that node is returned.

An example of the algorithm is included in appendix C.

With the new algorithm, the lower bound for the algorithm becomes $\Omega(1)$. An easy example is a range query $[x, y]$ where the root node's range is itself $[x, y]$. Since the two ranges are the same, the algorithm immediately returns the minimum and maximum values stored in the root node since this *is* the minimum and maximum in that entire interval.

The example above is of course very idealized. Usually, the two paths, left and right, will traverse the tree in a more chaotic manner. This means that the algorithm might still need to traverse the entire height of the tree, thus the run time still remains $O(\log T)$.

This algorithm is not implemented as the algorithm from section 4.3.2 works fine in the limited scope of this project, see the already accomplished speed gain in section 6.2.1. The time series are not *that* big (due to data file restrictions mentioned in section 5.3) and the fairly small amounts of break points. Implementing this was not a priority.

5.4 The structural breaks algorithm

The algorithm for finding structural breaks is found in the class `BreakPointAlgorithm`. This is the brain of the system and the *model* in model-view-controller. This class itself manipulates the other objects in a manner much like the handed out pseudo code. Thus, the implementation here is not very interesting. A class diagram of the break point algorithm together with time series and fitness models is shown in figure 13 in appendix B.

There is a small deviation from the pseudo code: A new population is initialized for each new run. This is not the case in the pseudo code where the algorithm can be run with an already initialized algorithm.

5.5 GUI: Displaying the graph and break points

The GUI itself is made in the FXML language from JavaFX with the help of the application *SceneBuilder*. Styling has been done in CSS. Given the simplicity of the application, only one scene has been designed as is ever in use. This has made designing the GUI a breeze.

Writing GUI in FXML makes the separation of *view* and *controller* very apparent. The view is in the FXML language. The controller is in Java.

The controller handles all the interactions from the user. This is methods like `openFileChooser` which opens the operating systems own file chooser to select a JSON file. With the file chooser open, only JSON files can be chosen. The controller then loads the time series sets that as the time series in the `BreakPointAlgorithm` and displays the time series in the GUI. The controller also tells the `BreakPointAlgorithm` to find the break points when the relevant button is pressed.

The controller also initializes all the slider and drop-down menus. For the drop-down menus this means adding all the elements and adding a listener to get the selection when a user selects an element. For the sliders, the initial value is set as the same as the `BreakPointAlgorithm` using the

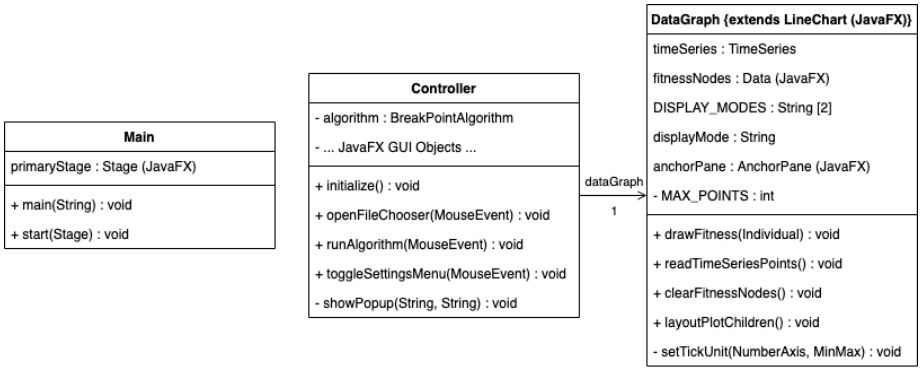
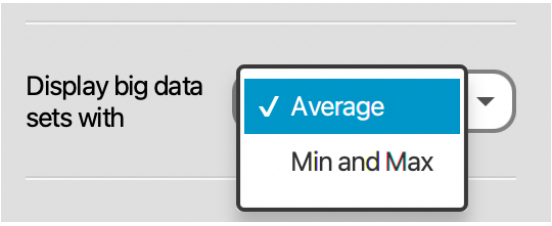
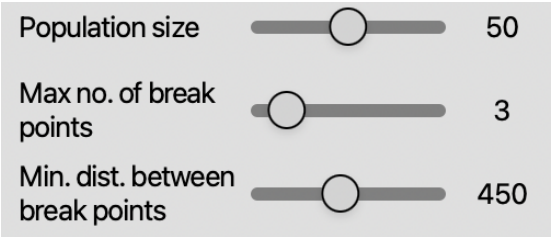


Figure 8: Class diagram for controller and main

static class `InitValues`. Listeners are added to observe the changing values. For most, this means updating the value display in the GUI and updating the relevant value in `BreakPointAlgorithm`. For the probability values a bit more work is needed: The math from section 4.2 is implemented also.



(a) Drop-down menu example.



(b) Slider example.

Figure 9: Examples of initialized and working menus and sliders

Most of the sliders have fixed minimum and maximum value except the slider that determines the minimum distance between break points. This slider scales depending on the size of the time series.

```
int length = timeSeries.getLength();
minDistanceSlider.setMax(Math.min(length, 10000));
minDistanceSlider.setMin(Math.max(length / 200, 1));
```

This means that if the length T of the time series $T < 1000$ then the maximum value for slider is set to T. On the other hand, if T is very big, the minimum value for slider becomes $T/200$. This changing of the minimum value is to ensure non-crazy runtimes for the algorithm. If the minimum distance between breakpoints is too low, the algorithm takes forever to run as the genomes become cluttered with break points which halts the execution time. This small fix eliminated that issue.

The slider for the maximum number of break points allows the user to choose a "don't know" value, "?". When selecting this value, all the individuals are initialized with a random number of

break points between 1 and 20. In the code, this works by setting the maximum number of break points `kMax` to 0 and, in the initialization of an individual, setting the actual number of break points like so:

```
int noOfBPs = kMax > 0 ? 1 + RAND.nextInt(kMax) : 2 + RAND.nextInt(19);
```

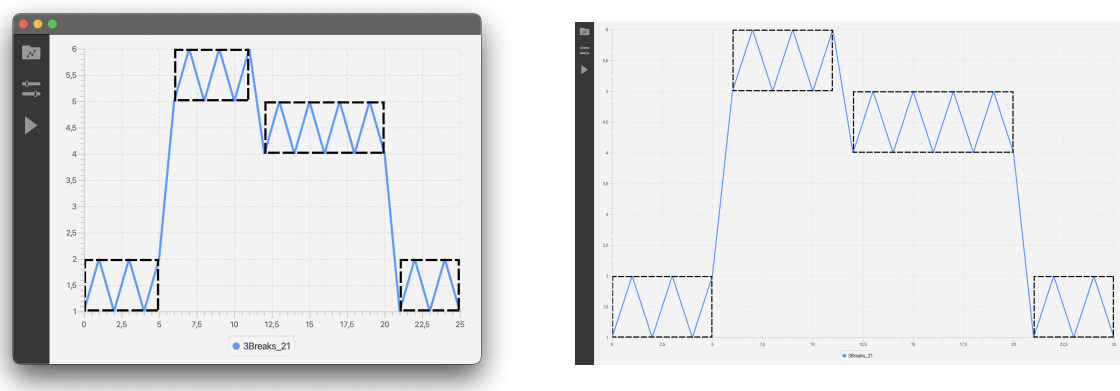
This must in turn also alter the penalty term in the fitness function. This is handled in the code like so (where `k = noOfBPs`):

```
if (kMax > 0) {
    penalty = Math.max(0., (kMax - k + 1.) / kMax);
} else {
    penalty = 1 / Math.sqrt(k);
}
```

This was found to a nice feature when the number of break points is not clear from the graph itself.

The controller only makes one *change* to the view: Replacing a placeholder `LineChart` with a custom `LineChart` object `DataGraph`. The reason to create a custom object is to draw the fitness markers on the graph. JavaFX' `LineChart` is *not* user-friendly in this regard and it has to be done through the private `layoutPlotChildren` method. Thus, a new object needed to be created.

Getting the break point markers on the graph was yet another challenge. First, the scaling from width and height on the time series to width and height on the display on the screen needed to be calculated. Then, the node has to be placed on the graph which, with the scale, was somewhat easy but took forever to get right.



(a) GUI in 480 x 640 pixels

(b) GUI in full screen, 2560 x 1600 pixels

Figure 10: GUI scaled from small to big (without rerunning algorithm)

At first, the markers was placed directly on the `LineChart` but it was changed to be placed on a `AnchorPane`. The decision to move to `AnchorPane` was motivated by one thing: The different type of nodes/figures in JavaFX has different parameters to decide the positioning. In an `AnchorPane`, the positions can be made to only x and y. Thus, the fitness nodes must have an x and y position for upper left corner, and they must also have a width and height. For the fitness node itself, these

values are in the scope of the time series. The transformation is done in `DataGraph` itself and is only graphical.

All this work payed off as the GUI is completely scalable and the fitness markers follows the scaling of the window, see figure 10

A class diagram for the controller part can be seen in figure 8.

6 Test

For this project, not many tests where performed. The tests where only for smaller components as the structural breaks algorithm by design is random and the outcome is not the same each time. The tests are only black box tests.

Another reason for the lack of tests was simply the scope of the project. It is not a very large project, and the amount of testable components is small. But a few tests are performed and can serve as a proof of concept.

6.1 Components

Tests were performed on the individual components to make sure they worked as intended. This was both the case for custom data structures and other objects.

For the range trees, a few tests were carried out to check that the queries returned the correct values. This was useful for making sure the data structure worked as intended.

Other tests were also carried out for the objects like `Individual` and `Genome` while they were being implemented. For the genome, it was important that the break point indexes are in increasing order.

6.2 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.

6.2.1 Optimization gains

A test were performed where the break points was found using two different `getMinMax` methods. The first run was with the simple method of looping through an interval and the other run used range trees.

The test was performed on a time series with 10^6 observations and the algorithm ran 30 times for each test. All the parameters of the algorithm was the same value across tests.

In figure 11a the average run time for the simple min-max algorithm was $t_s = 434688ms/30 = 15000ms$. With the range tree implementation, the average time is $t_r = 7229ms/30 = 241ms$. Thus, the algorithm using range trees is $t_s/t_r = 60$ times faster for time series with one million data points.

Note that the running time is for the *entire* break point algorithm and not just queries.

Unfortunately, these tests were not performed after going from the simple arrays mentioned in section 4.1 to the optimized arrays mentioned in section 4.3. The more theoretical analysis of the run times will have to suffice.

```
Run time: 434688ms
```

(a) Run with simple min max algorithm

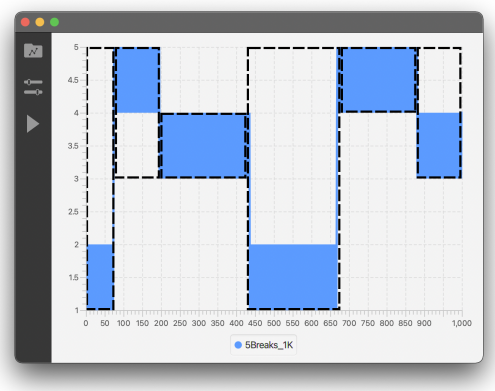
```
Run time: 7229ms
```

(b) Run with range tree min max

Figure 11: The run time for finding break points with different min-max methods. Time is running the algorithm 30 times. The time series had length $T = 10^6$ and all runs were performed with the same parameter values.

6.3 The algorithm

As mentioned above, the algorithm will not return the same result each time due to the randomness. Thus, for testing the algorithm actually works, the result was displayed either in the terminal or on the GUI, see figure 12. For displaying in the terminal, the fittest individual was printed. The GUI displays the fittest individual in accordance with the fitness model. The "tests" were black box tests with subjective evaluation.



(a) Algorithm output in GUI

```
77 198 428 678 880
```

(b) Algorithm output in terminal

Figure 12: Displaying the output of the algorithm for testing. (Results from the same run)

In theory, some automatic tests could be set up. This could be a matter of testing for fitness: Testing if the fitness of the fittest individual never declines during the algorithm; testing if the fitness of the final fittest individual is within a certain range of the optimal fitness for a given time series. These seem somewhat abstract though and are therefor not implemented during this project.

7 Conclusion

Of the three goals mentioned in section 2, all three have been reached. The algorithm works and it is fast due to range trees, 60 times faster in fact (for larger time series). A GUI is implemented which can show the time series with fitness markers/break points markers. Furthermore, the user can tweak a multitude of parameters from the GUI. The bonus goal, to make it is easy to implement further fitness methods is also achieved through the use of abstract classes. All goals are met.

On the non-project-goals side I have also succeeded. I have fought hard and long with Maven, but at some point, it finally wanted to cooperate. The last goal, writing the report in english succeeded I think. Obviously it was a challenge but it was nice to try out. As for the other goal, about working on a big project, is perhaps the least concrete and thus difficult to measure. But, I have learned a lot about working on big projects and made many mistakes doing the months. And I guess that counts as making it.

One of the key words for the application was stability. While the GUI is indeed stable and works great, it would have been preferable to ensure stability of the program. The problem lies within error handling. This has not been a focus during the implementation fase. It *is* implemented but almost all exceptions are shown to user via popups in the GUI. It could have been handled more elegantly.

If I could start again, I would have waited longer before coding. I rather quickly jumped into the coding fase without having much overview of the design. This lead to hours of aimless typing and waisted time. I should have stayed in the design fase for longer.

Another thing I would improve in my workflow is writing the report while coding. I kept telling myself I would do it, but I "just had to finish this last thing" until it was finished and I found another thing to improve. In the future, writing the report must be higher on my daily to-do list.

During the project, I have learned a lot about my own perfectionism and how to limit it. I have tracked my time, so I have not spent more than 9 hours on the project each week (or day during the last three weeks). During the last fase, I have wanted to tweak everything *one last time* and make it perfect. But I have been good to stop those thoughts. The perfectionism is still there but it is managed. And I am proud of the product.

A Code snippets

Listing 1: Code for generating range tree recursively

```
private void generateTree(Vertex vertex, int leftIndex, int rightIndex) {

    vertex.leftKey = leftIndex;
    vertex.rightKey = rightIndex;

    if (!vertex.isLeaf()) {

        int separatorIndex = (leftIndex + rightIndex) / 2;

        Vertex leftChild = new Vertex();
        vertex.left = leftChild;
        generateTree(leftChild, leftIndex, separatorIndex);

        Vertex rightChild = new Vertex();;
        vertex.right = rightChild;
        generateTree(rightChild, separatorIndex + 1, rightIndex);

        // Min and max is calculated from the children
        vertex.minMax = MinMax.combine(leftChild.minMax, rightChild.minMax);

    } else {

        // The vertex is a leaf, so leftIndex = rightIndex
        double value = values[leftIndex];
        vertex.minMax = new MinMax(value, value);

    }
}
```

Listing 2: Java code for retrieving min and max in a query range [leftKey, rightKey]

```java
public MinMax getMinMax(int leftKey, int rightKey) throws Exception {
    return getMinAndMaxHelper(leftKey, rightKey, root);
}

private MinMax getMinAndMaxHelper(int a, int b, Vertex x) throws Exception {

    if (x.isLeaf())
        return x.minMax;

    // The two paths are following each other, ie. this node is on both paths.
    if (x.left.inInterval(a) && x.left.inInterval(b)) {
        return getMinAndMaxHelper(a, b, x.left);
    } else if (x.right.inInterval(a) && x.right.inInterval(b)) {
        return getMinAndMaxHelper(a, b, x.right);
    }

    // The two paths are splitting up AT THIS NODE
    if (x.left.inInterval(a) && x.right.inInterval(b)) {
        return MinMax.combine(getMinAndMaxHelper(a, b, x.left),
                              getMinAndMaxHelper(a, b, x.right));
    }

    // The two paths have already split apart. This means that the current
    // node is only on the path to ONE of the two endpoints, either the left
    // path or the right path.
    if (x.left.inInterval(a)) {
        return MinMax.combine(getMinAndMaxHelper(a, b, x.left), x.right.minMax);
    } else if (x.right.inInterval(a)) {
        return getMinAndMaxHelper(a, b, x.right);
    }

    if (x.right.inInterval(b)) {
        return MinMax.combine(getMinAndMaxHelper(a, b, x.right), x.left.minMax);
    } else if (x.left.inInterval(b)) {
        return getMinAndMaxHelper(a, b, x.left);
    }

    // If this is reached then the interval is not valid
    throw new Exception("The␣interval␣is␣not␣valid");
}
```

B Class diagrams

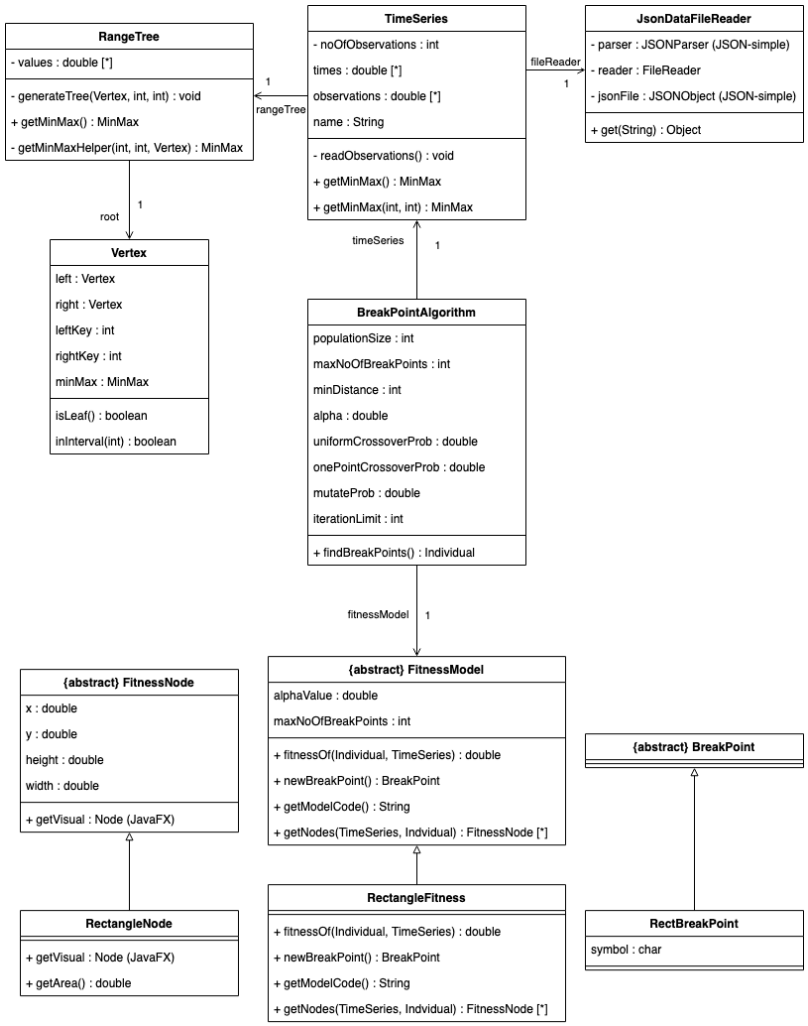


Figure 13: Class diagram of the break point algorithm with time series and fitness

C Optimized range tree query algorithm example

A few scenarios can occur. These scenarios will be illustrated by calling the query range $[1, 5]$ on the tree in figure 14.

- In the beginning, the two paths can follow each other. This is the case since the left child has the range $[0, 7]$ and $0 < 1 < 5 < 7$. The algorithm goes into the left child node and the query range remains $[1, 5]$.
- At the left child node, the two paths must split since $1 \in [0, 3]$ and $5 \in [4, 7]$.

  During the split up, two things will happen: The algorithm will go into both children respectively *and* the query range going into the either child will be *updated*.

  Going into the left child, the query range is updated to $[1, 3]$. Notice that the lower bound remains the same but the upper bound is updated. This is because, for all children of the node with range $[0, 3]$, the upper bound cannot exceed 3.

  Going into the right child, the query range is updated to $[4, 5]$. Here, the upper bound stays the same, but the lower bound is updated to 4. Again, this is because, in this side of the tree, no children can have a lower bound < 4.

  At the end, the algorithm combines the query result of the left and the right path.
- Now the left path (with query range $[1, 3]$ is at the node with range $[0, 3]$. The lower bound 1 is the left child but the upper bound 3 is in the right child. Now, the algorithm once again splits up. But notice that, going into the right child, the query range is updated to $[2, 3]$. This is exactly the range of that node, so the result from that node is returned. For the left path, this will always be case for when the path is "splitting up": The query range going into the right child will be equal to the range of the right child and the algorithm stops here and returns that nodes minimum and maximum.

  The left path continues into the node with range $[0, 1]$ with range query $[1, 1]$. This makes the path go into the right child where it returns the minimum and maximum value from that node given it is a leaf.
- From the split, the right path (with query range $4, 5)$ goes into the node with range $[4, 7]$. Given that $5 \in [4, 5]$ the path goes to the left child with query range remaining as $[4, 5]$. Notice that the left child also has the range $[4, 5]$ and the path then ends here and returns the result.

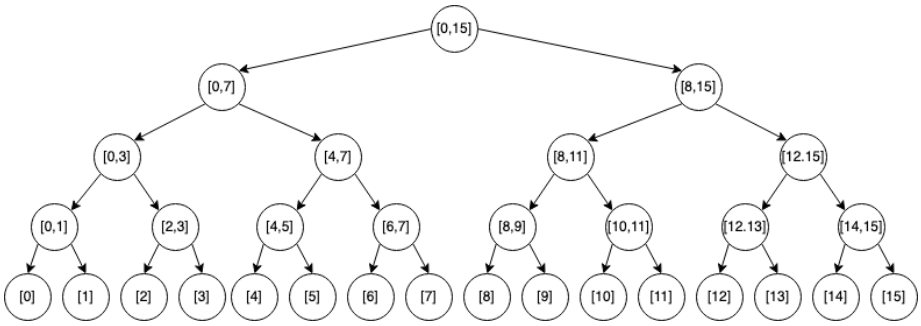


Figure 14: Example of a range tree with ranges.

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.

[Fischer and Hilbert, 2014] Fischer, P. and Hilbert, A. (2014). Fast detection of structural breaks. *COMPSTAT*.

[Point,] Point, T. Genetic algorithms.

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