Numerical Data Visualization Library

Heidi Tarkiainen, 1025372

Tietotekniikka 2021, SCI

26.4.2022

Contents

General description

The idea of the project is to be a library for visualizing numerical data. There are three different graph types implemented where to choose for the graphical visualization – a line diagram, a pie diagram and a bar chart. A user inserts a file through the interface and choses what graph to use on it. If the data on the file were read successfully the graph type chosen will appear on the interface. The user can afterward adjust the graph by changing colors, editing titles, changing names of x- or y-axis and adding a grid behind the graph. Since all three graphs are implemented on the project it means that the project is done on the demanding level.

The first one is a line diagram which is a visual comparison of how two variables—shown on the x- and y-axes—are related or vary with each other. It shows related information by drawing a continuous line between all the points on a grid. Datapoints inserted are in two-dimensional form meaning a set of x- and y-coordinates. In addition to dots and a polyline, also x- and y-axis are drawn with stamps on to give an idea of the scale of the graph. The help reading the values a grid behind the graph can be added. The grid size is editable and when changed the stamps on x- and y-axes are recounted. As mentioned before, the color of the dots and lines and names of x- and y-axis are editable after making the graph. In addition, there is feature where you can resize the whole graph. Since a graph is always autoscaled such as a graph would fill the interface as much as possible that is the maximum size of a graph. The resizing works only towards smaller sizes from that.

The second graph is a pie diagram which used to illustrate numerical proportion. A pie diagram is divided into slices depending on the quantity it represents. The sum of each value forms a full rotation around the circle. To make it easier to read each segment is colored differently. The colors can be changed by pressing a button which when pressed choses colors randomly. A user can go backwards on the colors when wanted. A title of a pie diagram can be added through the interface. If wanted a user can also add an info about the diagram which includes a color used to describe a value, a name of the value, percentage of the value and the value itself.

The third and the last graph implemented is a bar chart. A bar chart is a chart that represents categorical data with rectangular which heights depends on the proportional value that it represents. The x-axis contains specific categories that are being compared and the y-axis represents the value of a category. Such like for a line diagram an adjustable grid can be added to help with reading the graph. Also, the title and name of y-axis are editable. The columns are colored with the same color which can be chosen in the interface.

User interface

The interface is started from the UI object which is an extended JFXApp. To visualize data, user must insert a correctly formed text file from the menu bar's 'File' button. Then a file chooser is opened, and a text file can be chosen. There will appear 'File Added: *file path*' text on the interface to inform a user that a file has been added successfully. When a file has been added successfully, graph type can be chosen from 'Analyze'. For a graph to be added successfully the file must match with the graph type chosen. Whenever making a graph the interphases must be done in this specific order. Otherwise, a graph won't appear in the interface and an error will occur.

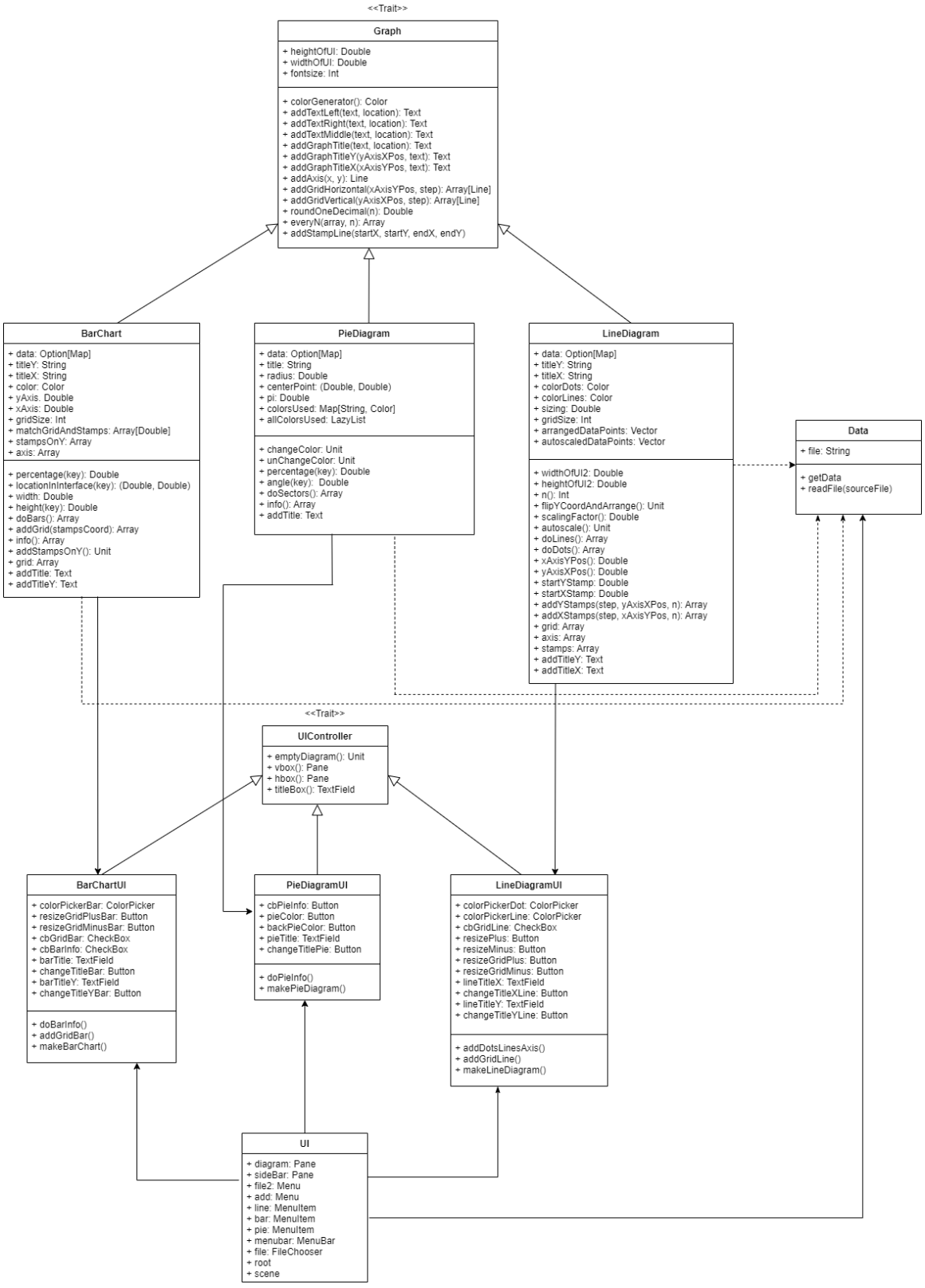
Depending on the graph type chosen, different things will appear on the side bar to adjust the graph. When the chosen graph is a line diagram the side bar will contain two color pickers for changing the color of lines and dots. For resizing the graph, the side bar contains minus and plus buttons which when pressed changes the size of a graph to certain limit. To add a grid the check box under text 'Add Grid:' must be checked. When it is checked, a grid will appear behind the graph and a minus and a plus button for resizing the grid until certain limits will appear in the side bar. For changing the names of x- and y-axis, the side bar contains two separate text fields. When 'OK' buttons next to the text fields are pressed the names inserted in the text fields will appear next to the axes.

For a pie diagram there is two buttons for changing the color of each segment. When 'Change' button is pressed the segments will be colored by randomly chosen colors. If wanted, user can go back to previous colors by pressing 'Back' button next to 'Change' button. Below that under a 'Show info' text there is a check box for adding more detailed information of the colors and proportional values of each value. For adding a title there is a text field such as in line diagram's side bar. When 'OK' button is pressed next to the text field the title will appear at the top center of the interface.

For a bar chart at top of the side bar is a color picker for choosing the color of the columns. Below that under a 'Show Values:' text there is a check box for showing columns value which appears above the bar it belongs to. For adding a grid there is a checkbox such like implemented for the line diagram so that when checked, minus and plus buttons will appear in the side bar for resizing the grid. For adding a title and changing name of y axis there are text fields such like in line and bar diagrams.

Program structure

Graphic Class Diagram



The project structure is split in two separate main sub-parts – to a part that has all the algorithms to construct a diagram and to a part that handles events in the interface such as a graph appearing in the interface. These sub-parts are split in 'Interface' package and 'Graph' package. This solution for the class structure helped to keep the sub-parts apart as much as possible which made division of assignment clearer.

Since each graph has their unique algorithms to construct a graph, there is own objects 'LineDiagram', 'BarChart' and 'PieDiagram' implemented. Each of these objects extends a trait class called 'Graph' which has variables and methods which are needed in every one of these objects such as measures of the pane that the graph will appear in the interface and methods for adding a title or axes and much more. This is to minimize repetition. For 'LineDiagram' object the main methods are autoscale() which autoscales data points so that the line diagram will appear as big as possible in the interface and doLines() and doDots() which makes an array of the diagram's components. For 'BarChart' object main methods are height(key), width and percentage(key) which do the calculations of the size of each bar depending on the proportional value of a value. Main methods for 'PieDiagram' are changeColor() which chooses randomly color for each sector, angle(key) which calculates central angle of sector depending on the proportional value of a value and doSectors() which adds the components of a pie diagram to an array.

The objects 'LineDiagram', 'BarChart' and 'PieDiagram' get the data for constructing the graph from object 'Data' which makes these object dependents on 'Data'. This is marked on dashed line on the UML diagram. The object 'Data' is for reading a file added in the interface. This makes 'Data' dependent on the 'UI' object. Most of the problem handling is handled in 'Data'. Since way of reading a file is depends on what graph type is the file intended for, if the graph type in the file and the graph type chosen in the interface don't match an exception is thrown. The exception is thrown in the graph's object which is chosen in the interface. Also, if the data in the file is wrongly formatted a 'Data corrupted' exception is thrown which happens in the 'Data' object. Other exception that can been thrown are 'File not found.' and 'Error occurred.' which are thrown in 'Data'. When a file is read, and the data added to the right object the algorithms needed to construct a graph can be called.

The sub-part that handles events happening in the interface are all in the 'Interface' package. This package contains own objects for each graph type – 'BarChartUI', 'LineDiagramUI' and 'PieDiagramUI'. These objects extend trait class 'UIController' which has methods that all the objects need to make the graph appear in the interface. Basically objects 'BarChartUI', 'LineDiagramUI' and 'PieDiagramUI' have methods that make the graph appear in the interface and the sidebar components appear on the side bar. Main method in these objects is the make*graphType* which adds the graph to the pane's children. Since the objects in 'Graph' package constructs the graph and objects in package 'Graph' adds the graph in the interface, this

dependency is marked on the UML diagram with a line between objects in ‘Graph’ and ‘Interface’ having an arrow towards the object in ‘Interface’. The event handling of the side bar’s components is implemented in these objects. The ‘Interface’ package has one more object being ‘UI’ which extends JFXApp. This object constructs the interface app such as it has a menu bar, a side bar, and a pane for the graph to appear in. The event handling of the menu items is implemented in this object.

Algorithms

A pie diagram is divided into slices depending on the quantity it represents. The sum of each value forms a full rotation around the circle. To make a pie diagram, the area of a sector needs to match its proportional value. To get a sector’s central angle match the proportional value needs to be counted. Then the central angle of a sector is **angle = 360 * (value / SumOfallValues)**. To make the sectors go a full around the start angle of a sector needs to be cumulative value of the sector angles before this sector.

A bar chart is a chart that represents categorical data with rectangular which heights depends on the proportional value that it represents. The bar with biggest value is drawn as the tallest bar. Other bars heights are the proportional value of this value compared to the biggest value. In other words, a bar’s height is defined by **height = heightOfBiggest * (value / biggestValue)**. For adding stamps on y-axis, the distance of two stamps in the interface and in the coordinates of the diagram to give an idea of the bar’s heights. This is done by figuring out the ‘scale’ which the graph has been scaled. This is defined by **scale = (biggestValue – smallestValue) / (biggestHeight – smallestHeight).** When a stamps coordinate is x, text next to the stamp is **txt = x * scale**.

A line graph is a bar graph with the tops of the bars represented by points joined by lines. Since the data points can be any real numbers, and the interface coordinates are only until a certain limit, the datapoints need to be autoscaled to fit the measures of the interface. This is done by first figuring out a scaling factor. The scaling factor is the smaller one of the following: **scaleX = widthOfUI / (biggestXPoint – smallestXPoint)**, **scaleY = heightOfUI / (biggestYPoint – smallestYPoin**t**)**. Scaled data point is defined by **scaled = scaledPrev + distanceUnscaled * scalingfactor.** The ‘distanceUnscaled’ is the distance between subsequent data points. Implementing x- and y-axis, the distance between smallest y or smallest x to origin needs to be known. That distance is scaled by scalingFactor defined before.

Data structures

The data structures used in the 'Graph' package are mostly arrays. That is because objects in the 'Graph' package construct parts of a graph and the parts need to be hold in some data structure. Since arrays are efficient to update this is good option for keeping hold of parts of a graph. There is also many for loops used which is not the most efficient way of going through an algorithm. But since the collection are on a smaller size this is ignored. Using for loops also step-up readability which is more important here when the collection size isn't that big. An array is mutable data structure.

The datapoints are saved in a map structure where in case of a line diagram x- and y-coordinates are mapped together and in cases of a pie diagram or a bar chart category and value are mapped together. Especially in the 'BarChart' and 'PieDiagram' map structure is widely used. This is because there is a lot of traits that differ from key to key. Since map as a data structure is efficient for lookup things this is a good data structure for this usage. A map is mutable data structure.

For sorting data like in 'LineDiagram' vectors are good data structure for this. A vector is immutable data structure being efficient for going through a collection and updating it. In the object 'PieDiagram' variable 'allColorsUsed' stack is used for a data structure. This is because the user can only go change the color of sectors which chooses randomly the next colors for the sectors to be colored with. The colors used latest are stacked on top of the stack. A user can go backwards on the colors. Then the colors before the colors now are chosen for the color of the sectors. Stack is an efficient data structure for taking the head of a collection and to update a collection.

Files and Internet access

The program only accepts and reads text files for the data. A correctly formatted file has in information of for which graph type is the data intended for and the data in correct format such as key and value are separated with a comma. If this is not the case, an error will occur. Also, the graph type written on the file and the graph type chosen to make in the interface must be same even though data in is the same way formatted for both bar chart and pie diagram.

<u>An example file for making a bar chart:</u>

```
Graph Type: BarChart

DataPoints:

Helsinki, 11002
Espoo, 9822
Vantaa, 7002
Sipoo, 2388
Lohja, 800
Porvoo, 2928
Salo, 8191
```

An example file for making a pie diagram:

```
Graph Type: PieDiagram

DataPoints:

Badminton, 109
Boxing, 9
Diving, 49
Hockey, 97
Sailing, 79
Tennis, 88
Volleyball, 44
Triathlon, 19
```

An example file for making a line diagram:

```
Graph Type: LineDiagram

DataPoints:

(-11, 191)
(102, 110)
(2, 10)
(-1, -22)
(23, 400)
(3001, -13)
(1329, 1309)
(22, 211)
(311, 11)
(222, 11)
(-22, 999)
(200, 1000)
```

Testing

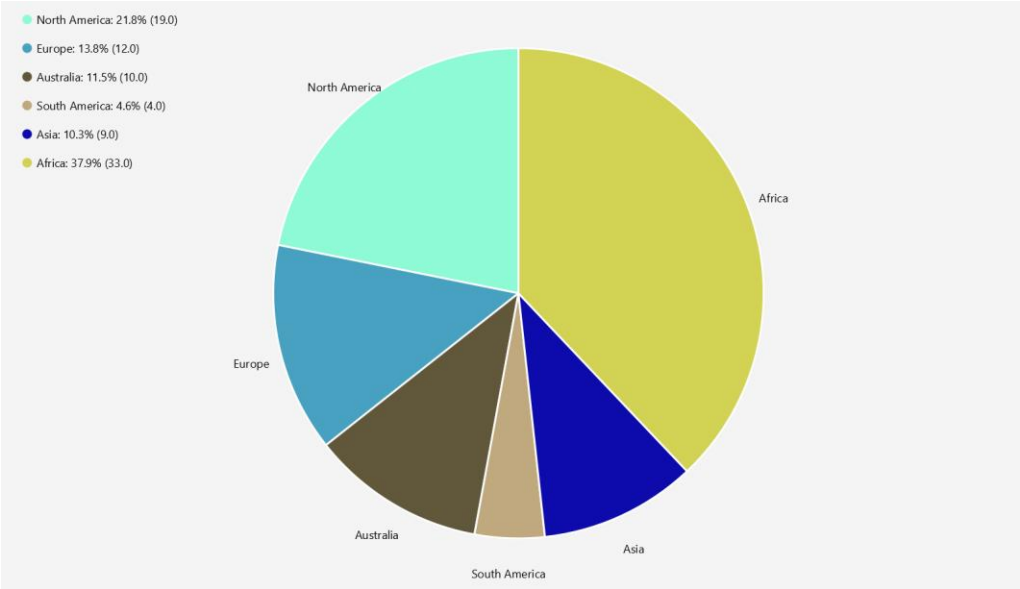
During the process the project was tested with test data points that weren't read from a file but coded straight to the graph type objects. I had few data sets with different sizes containing data from various ranges to test the graph types. Unlike what was planned I implemented the interface app early on the project to help seeing what is working and what is not in the algorithms.

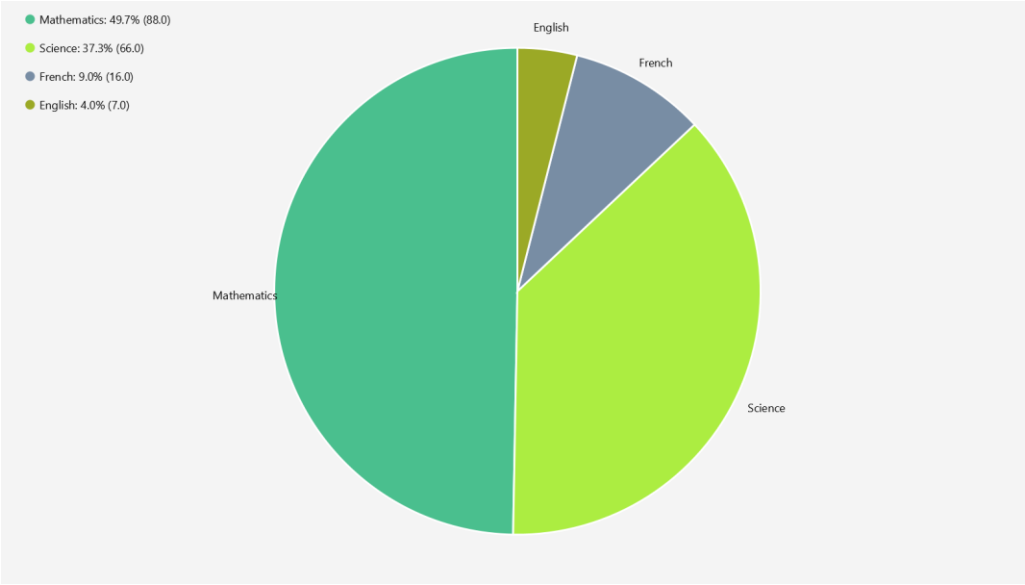
For the final testing I made test files that are added to the project's repository in package 'Test files'. There are at least seven test files added for each graph. The test files contain data set with different sizes and different ranges. I also tried to take the edge cases in consideration. There are files containing data from very different ranges, data with wrong format or no data. All in all, test files of a graph type are implemented to be as different from each other as possible to get a good idea of if the implementation for a graph type is working. I didn't implement automated tests for the project due to lack of time, so I tested the test files manually. I tested manually different scenarios such as making a wrong graph type on a test file.

Known bugs and missing features

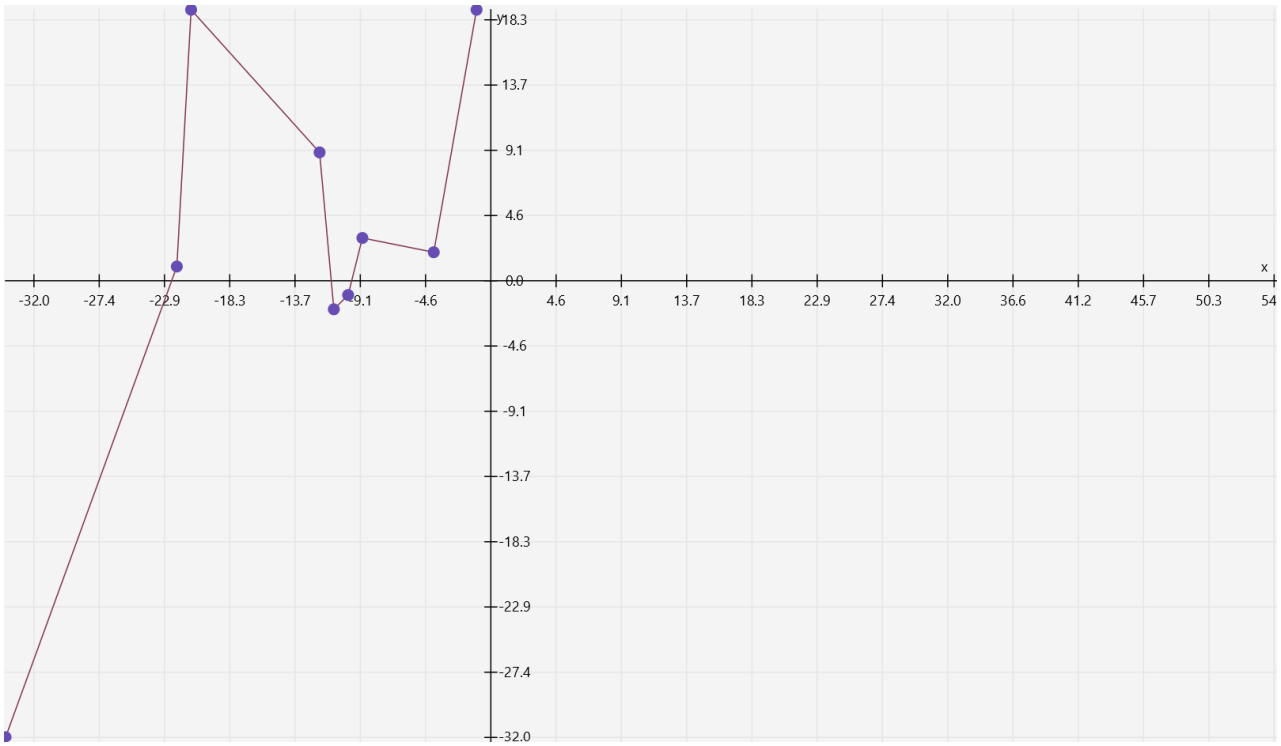
Even though especially for me the interface app is easy and straight forward to use, there could be some information that could appear in the interface if some error occurs. That could tell the user what has been done wrongly. For example, if the file added was wrongly formatted there could be a text 'Error while reading a file' to inform the user. Now if an error occurs there is nothing happening in the app even though the error occurs in the console.

Known bugs in a pie diagram are related to text box positions for the names of the keys. The idea was to have them at the middle of a sector's arc at a constant distance outside a sector. To do that, there is a 'addTextMiddle' method implemented in trait class 'Graph'. If the name of a key is too long, it will not be at the same distance to a sector as the others and sometimes is above a sector. The bug can be seen in the following graphs:

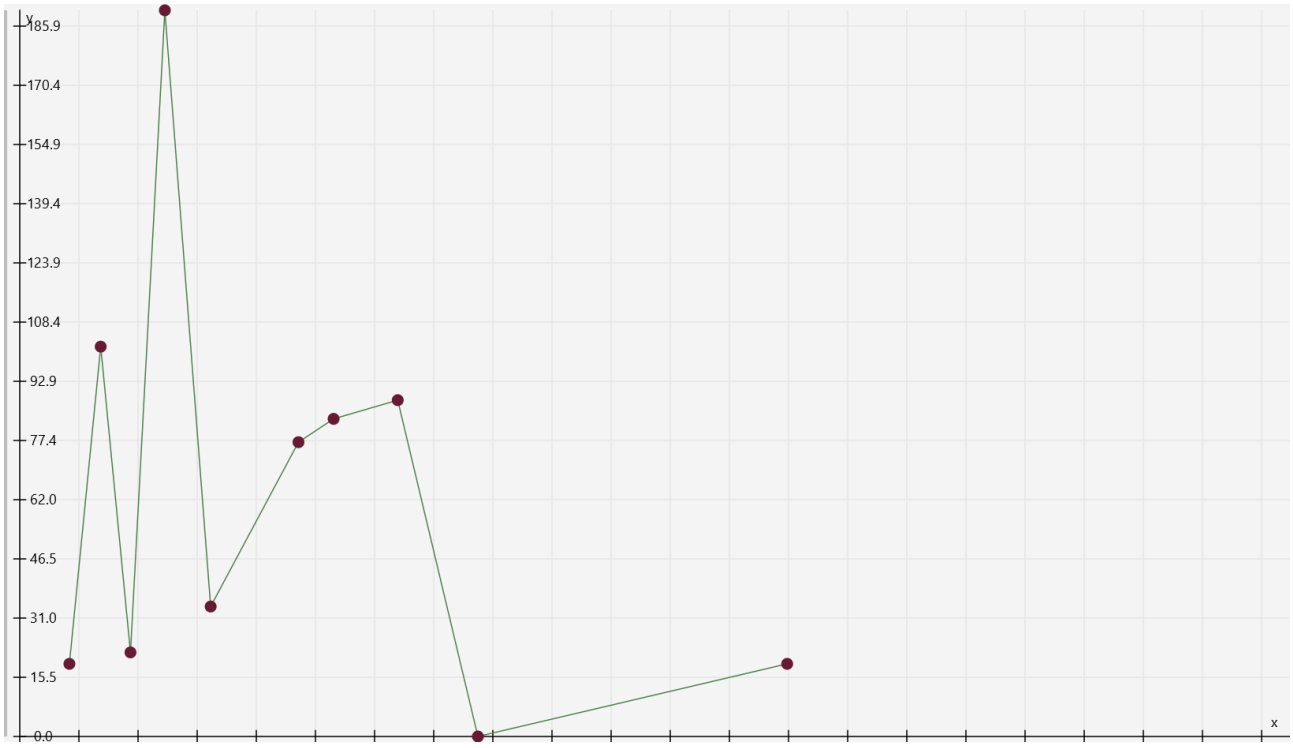




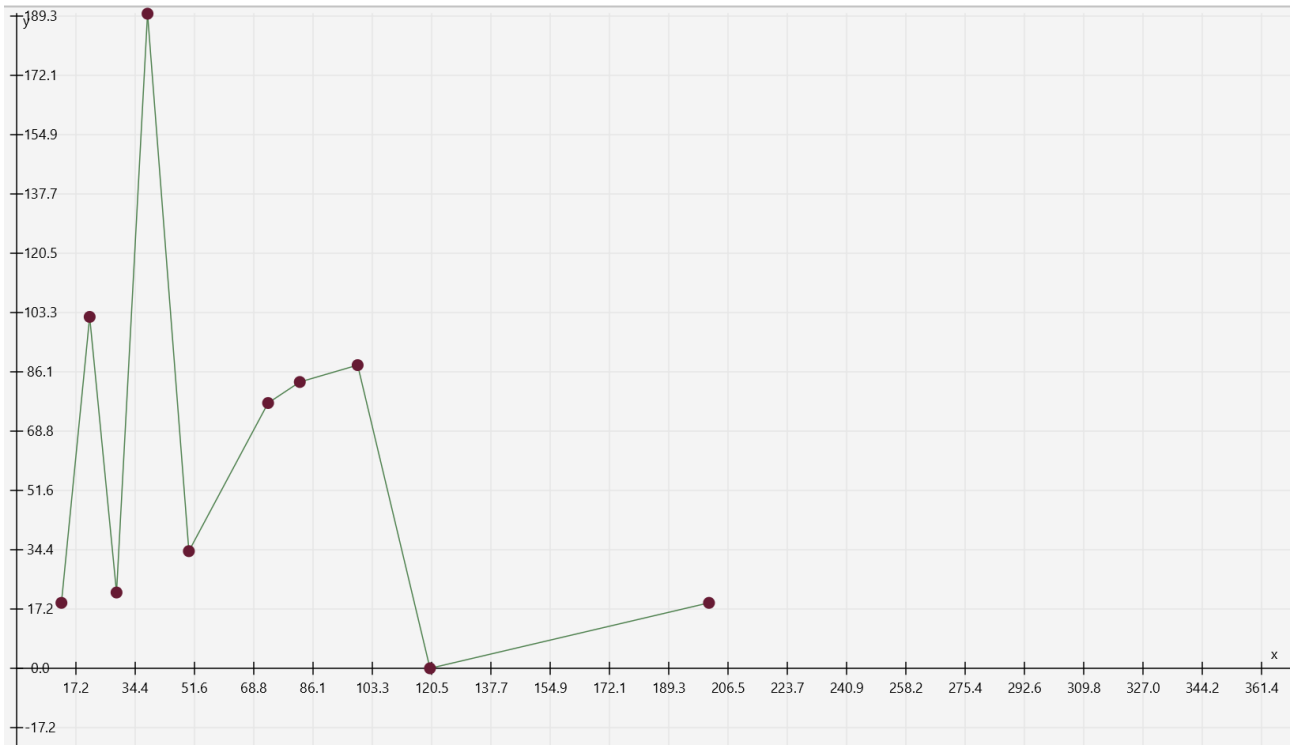
Known bugs in a line diagram are mostly related to the axes and scaling the datapoints. For some grid and graph sizes, the stamps on the axes will overlap with the name of an axis. This can be easily fixed by changing the grid size or the size of a graph. The bug is seen in the following graph:



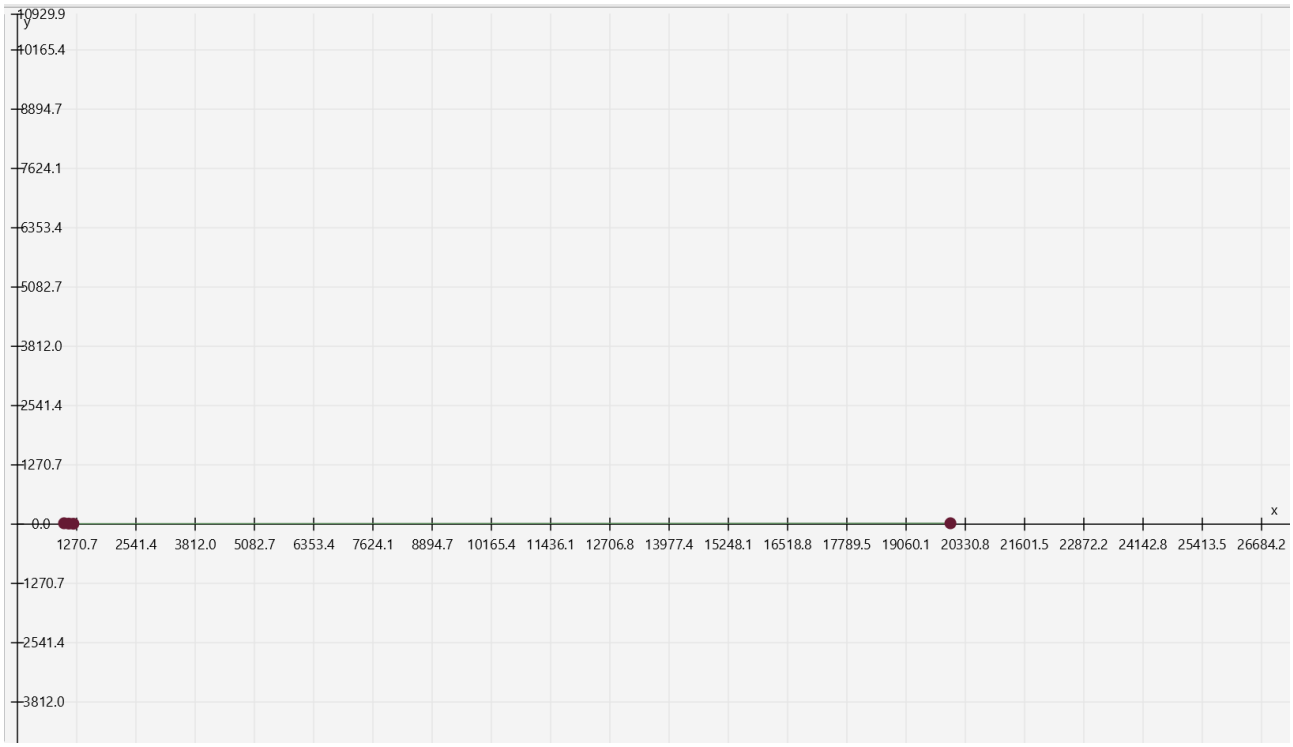
Another bug in line diagram is that if the y-coordinates of all data points are positive stamps on x-axis won't be shown. This can be fixed by making the size of a graph smaller. The bug is seen in the following graph:



When the same graph has been made smaller:

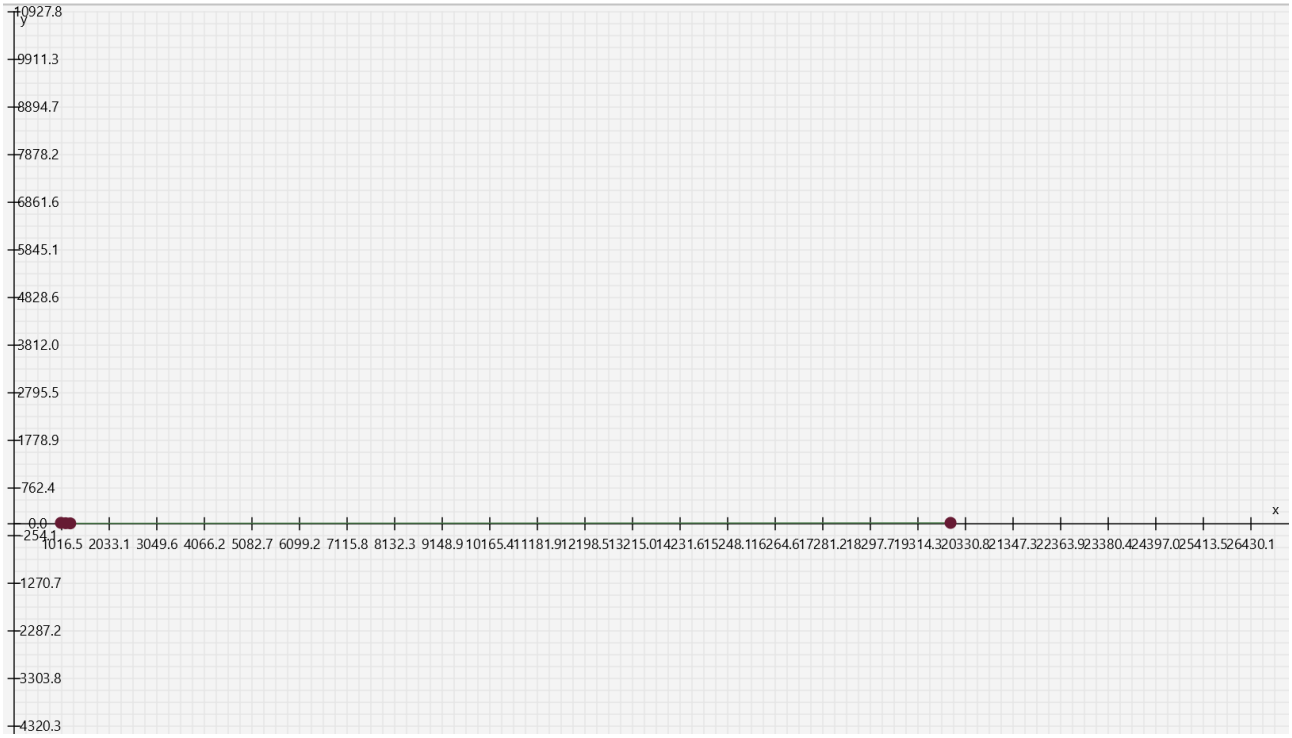


Also, if the x-coordinates and the y-coordinates are on very different rate than the x-coordinates, a graph can end up looking like the following graph:



As seen, the graph is not very readable.

Sometimes if the x-axis has very large numbers, the stamps can start overlapping each other with small enough grid size. The bug is seen in the following graph:



There are no known bugs in making a bar chart.

3 best sides and 3 weaknesses

Best sides:

One of the best sides is the whole implementation for line diagram. It seems to be working except the very edge cases mentioned before. The grid resizing works seamlessly and the axes bring more readability to the diagram.

The second of the best sides is the coloring of the sectors for pie diagram. Even though since the colors are chosen randomly and they may be close to each other this implementation for changing and unchanging them seem the best. Since the number of sectors can vary a lot, I didn't see it very user friendly that the user could choose the color of each sector separately.

The last best side is the class structure. I think the implementation for it is now easily extended to make more graph types since every graph type has their own objects in 'Graph' and in 'UI' packages.

Weaknesses:

One of the weaknesses has to do with the file management. The original plan was to have to data read from a json or csv file. These file formats would've been more intuitional for containing and reading numerical data. Since I ran out of time, I had to stick with a text file format since I knew for sure how to work with that file format. Also, a weakness in the file reading is that a title is not read from the file. This wouldn't have been a big thing to implement but again because of lack of time this isn't implemented.

Second weakness has to do with autoscaling data points for a line diagram. It would've done more sense if the graph would've been centered in the middle and not on the left of the interface when the data points don't fill the whole width of the interface. Other weakness in the autoscale is that the x-axis isn't always shown in the interface when a graph is at its biggest.

Third weakness is the sizing of diagram pane. The height and width that the 'Graph' package is using is the measures of the pane when the interface is at its full size on my computer. Since measures may vary from computer to computer, the graph may not appear in the interface on full size.

Deviations from the plan, realized process and schedule

I started doing the project couple weeks late to the schedule. For first couple of weeks, I tried to figure out how the interface app worked with line diagram. Even though I had the class structure done for the 'Graph' package, I tried to first figure out how to the line diagram. When a skeleton of that was done, I started to do the other graph types. When the skeletons of other graph types were done, I started adjusting them such as doing the grid and changing colors methods and fixing known bugs in them.

I had planned to first implement the algorithms doing the graphs before doing the interface app. This ended up not working since seeing if the graphs worked was a big part of implementing the algorithms. That led to that I had to do the interface early on the project. This made doing the algorithms faster since I could see what was right or wrong on the algorithms.

Unlike what was planned, I implemented the file management on the last week. In the original plan I had it planned early on the project. I managed to test the graphs with various test data points until the very last week. I also didn't have two weeks only to test the program. But that wasn't a problem since I had been testing it along the process. On the last weeks I focused more on the event handling of the interface.

For the hours used on this project I have counted around 80 hours. The number of hours spent on the project started to grow exponentially when getting closer to the deadline. Also, more hours were spent on the graph type objects than in file management or testing unlike what was planned.

During the process I learned that it is difficult to get to start but when that is done it's easy to get to workflow. What I also noticed was that the excitement of doing the project started to go away at the end and the hours used on the project started not to be so efficient.

Final evaluation

Overall, I think that the project does what it is required to do. There are little things that can be done to improve such as taking the edge cases in line diagram to consideration. The logic and algorithms behind constructing a graph are carefully researched and done such as they are doing what they supposed to do.

There are also few things in the interface app that can be done such as styling the side bar better to improve user experience. When I let my friends test the app, I heard that it was easy and simple to use. What could be improved was the error cases. There is now no information of an error occurring in the interface that could tell the user what has been done wrongly.

Also as mentioned before, the file management didn't end up like I had planned. The file format could be some better for containing numerical data. What comes to efficiency, the for-loops and some of the data structures used aren't chosen while keeping efficiency in mind. Also, some methods are called many times for no reason when adjusting a graph. But since the data kept in a file is assumed to be small, I focused more on readability and making the graphs work. If I would've had more time, I could've focused on efficiency more.

In my perspective, I think the project can be easily extended since every graph type has their own object for constructing a graph and another object for the event handling in the interface. Also, the trait classes were made keeping the extensibility in mind.

If I could start the project over again, what I would do differently is that I would spend more time thinking about sustainable solutions for problems. I kind of have a bad habit of starting first write code and seeing if it works where I should be first thinking and then coding. This habit led to that I spend a plenty of time redoing methods that did work some how for some data but not in all cases. All in all, I'm satisfied with the final product even though it has its weaknesses. I have learned a lot during the process which can be seen in the original technical plan for the project. From reading that it can be seen that I was really lost with the project at the beginning.

References

ScalaFX. Available at: https://www.scalafx.org/ (Accessed 26.4.2022)

JavaFX API. Available at: https://docs.oracle.com/javase/8/javafx/api/overview-summary.html (Accessed 26.4.2022)

JavaDoc. Available at: https://openjfx.io/javadoc/11/index.html (Accessed 26.4.2022)

Java: *Tutorials for Software Developers and Technopreneurs*. Available at: http://tutorials.jenkov.com/ (Accessed 26.4.2022)

Sector of a circle. Available at: https://www.cuemath.com/geometry/sector-of-a-circle/ (Accessed 26.4.2022)

Stack overflow. Available at: https://stackoverflow.com/questions/11106886/scala-doubles-and-precision,

https://stackoverflow.com/questions/50703239/creating-random-string-to-represent-rgb-color (Accessed 26.4.2022)

Wikipedia: *Bar Chart.* Available at: https://en.wikipedia.org/wiki/Bar_chart (Accessed 26.4.2022)

Wikipedia: *Pie Chart.* Available at: https://en.wikipedia.org/wiki/Pie_chart (Accessed 26.4.2022)

Wikipedia: *Line Chart.* Available at: https://en.wikipedia.org/wiki/Line_chart (Accessed 26.4.2022)