Justin Sybrandt
Permutation Visualization

Data Structures:

A majority of the interesting work data structure work done is this project involves permutations themselves. Generating permutations, traversing through collections of permutations, and finding the relationships between permutations are all very heavily used tasks. Additionally, when visualizing these structures, the MVC programming pattern aided me in developing connections between the permutations themselves, and the visual structures I use to represent them. The following document describes each of these topics in detail.

Permutations (Model):

A permutation, simply put, is an ordered series of *n* items. Because the length of a permutation is known in advance, an array seemed like the simplest choice. Additionally, the Java framework includes built in array equality and array hashing. This means that if my permutation data structure is a logical extension of the Java array, then I will be able to use collections like the hash map or set with no additional work.

But of course, there are many operations we wish to do on a permutation, such as find the inverse, the cyclic notation, or adjacent permutations. Supporting these and other operations is easiest if I make sure my permutation structure is immutable (My inclination for this likely comes from a recent large project in Scala, a functional programing language which greatly encourages immutable data structures). Therefore, when instantiated, the permutation class creates a copy of the array it was given and provides no public methods which allow for manipulation of its internal state. As a result, the permutation class is thread safe, understandable, and encourages clear programming practices.

Lastly, one of the most important features of the permutation structure is its ability to be generated from a factoradic number. By interpreting its array as a factoradic number, the permutation structure can easily map to and from the set of integers.

Permutation Generators:

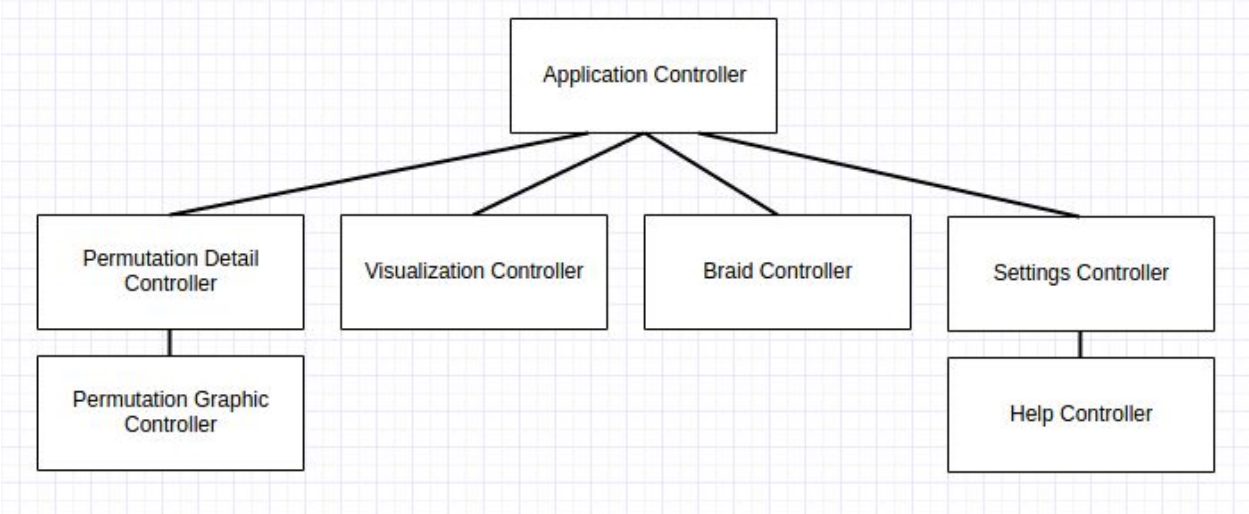
One of the main interests in this project are the generator functions which create sequences of permutations of a given length. Because we need to have three different permutation orderings, it makes sense to have a common interface for the three permutation generators. These all are able to generate a set of permutations given a size, returning a list. The selected generator functions are factoradic, swap, and insertion. The factoradic generator simply generates each permutation by factoradic number, taking advantage of the integer to permutation mapping provided by the permutation class. The swap generator uses Heap's algorithm for generating permutations based on swapping values. (https://en.wikipedia.org/wiki/Heap%27s_algorithm)

Note that this generator can be visualised as a braid pattern, this will be referenced later in the braid visualization. Lastly, the insertion generator performs a breadth first search through the set of permutations by iteratively inserting numbers in different positions of a numeric list. That list is then converted to a permutation when the desired size is reached, and stored in order.

The overall effect of these generators is that different orderings of permutations can easily be created, highlighting different relationships within their set. The factoradic generator places numerically close permutations together, while the swap generator places swap-similar permutations together, for example.

Controllers:

In this project, I attempted to hold myself to a stricter interpretation of the MVC pattern than before. As such, I treat JavaFX primitives (such as Pane, or Canvas) as view objects, and attempt to only create generalized views (such as the zoom pane, which can pan and zoom its contents). Because of this interpretation, my controller classes grew in complexity. My controller classes form a small hierarchy, as detailed below. Each controller corresponds to a single view element to which it is responsible for.



Application Controller: Coordinates the controllers and the application pane.
Permutation Detail Controller: Shows the selected permutation details on the right hand side. Uses Permutation Graphic Controllers to create the permutation graphics at the top of that pane.
Permutation Graphic Controller: Generates the graph or matrix for a permutation given a canvas element.
Main Visualization Controller - creates a visualization of a set of permutations given a generator.
Braid Controller: Generates the left hand side braid visualization of a permutation set. Uses the swap generator to create this ordering.

Settings Controller - is responsible for the top buttons and selections. Creates the help controller for the help button. Informs the Application controller of user selections.
Help Controller - displays the help modal dialog when the help button is pressed.

Visualizations:

Many different visualizations come together to inform the user about the permutations they are seeing. Most obviously, the main set visualization in the center of the screen shows the entire set in generator order (clockwise starting at the bottom, or row-wise for large sets). For small enough permutations, the braid visualization on the left side also shows the same permutations in a different ordering. By showing two orderings (one generated by a user-selected algorithm, another shown by the swap generator) the user can clearly see the differences between each permutation generator.

When a user clicks on one of the main set icons, or on the braid visualization, the corresponding permutation is highlighted and displayed on the detail view, shown on the right side. Here we see a single permutation detailed as a matrix, as a graph, and in multiple notations. As the user traverses through the permutation set visualizations, they can clearly see changes represented in multiple diagrams.

Selection Pane:

Originally, the main visualization was the only component in the center of the display. Once generating over 5000 shapes and nearly as many labels, I found that I was pushing javafx to the limit. When a user clicks on one of the permutations, they expect to see edges to adjacent values, as well as the inverse. Unfortunately, the process of removing edges from unselected permutations proved too slow. This is because the JavaFX framework chose a list as the data structure of choice to hold child elements in a pane. Because of this (poorly chosen) data structure I had to be creative in order to reduce lag after selecting a node.

My solution was to add a second, invisible pane on top of the visualization. This pane holds the lines and dots which are shown superimposed on the permutations of size 6 and 7. When a new selection is made (the user clicks or presses left / right) instead of searching through the visualization to remove the added edges, I simply erase all children of the selection pane. This served as a lesson to me as to why I should be cautious when using lists in my program.

Zoom Pane:

Typically view elements are either off the shelf components such as the javafx pane or canvas, but occasionally they require a lot of work and data structures of their own, as was the case for the Zoom Pane I created. I wanted users to be able to pan around and zoom in on different parts of the permutation set, but there is no way to do this without making custom components.

The Zoom pane, conceptually, consists of a two dimensional vector, representing this displacement of the view, as well as a number used as a zoom factor. Once those two values are set, all elements within the zoom pane should be shifted and scaled accordingly. This is easier said than done.

In JavaFX, elements can be moved about the screed using a variety of properties. In order to be consistent, I use layoutX and layout Y throughout my design; with one exception. The JavaFX line, which I use to show edges between related permutations, has properties for its start and end, which must be set for the line to be displayed. This means that additionally setting the layout properties would misalign the lines, destroying the visualization. Although these properties can be “bound” together (changing one implies an equal change in all that are bound to it) I had persisting issues in many visualizations even after binding them together. I was able to solve this problem by utilizing the existing property data structure in a novel way.

I defined small circles, set to an invisible color with a radius of zero, and I use them as the anchor points for many lines within my large scale visualizations. The zoom pane, as a result, makes sure to only interact with unbound circles. As a result, these anchor points are manipulated by the user, and as a result lines which are bound to anchors throughout the program update automatically, even if they are not directly present on the zoom pane.

Lastly, I have to ensure that all of a zoom pane’s elements actually stay within the zoom pane bounds. Without this check, the user would likely zoom and pan permutations into other screen elements. Luckily, I can do a simply check on each element to ensure that each node’s screen bounds lie within the zoom pane. This is made simple by the javafx bounds data structure which provides and easy interface to compare axis aligned bounding boxes. It is clear how important framework design is when working with a good one.