1 USER MANUAL

1.1 Adding Your Own Algorithms

Now that you’ve become accustomed to the visualizations that ship with SAVY perhaps you’d like to start adding your own visualizations? The following section details the API that SAVY exposes to you.

1.1.1 Creating Your Own SlowData

Disclaimer

The first step to understand MySlowDataQueue is to understand that it should not be altered in any way. MySlowDataQueue should provide enough of an abstraction that any changes that need to be made can be made by creating your own SlowData that extends MySlowData. If you find it absolutely necessary to edit the way that MySlowDataQueue works please extend the class so as to avoid breaking all of the built in algorithms, as well as any user defined algorithms that you may have acquired.

MySlowDataQueue contains three lists of MySlowData, posted, others, and permanentData. One important feature of MySlowDataQueue is that it allows for the simulation of real-time acquisition of data. It does this by moving MySlowData from the others list, which is not accessible from outside of the class to the posted list which provides access to the first element of the list (via MySlowDataQueue.poll()). This movement is done by the update() method which is called periodically by each algorithm. Whenever an item is moved to the posted list it is also moved to the permanentData list. Data items that are added to the permanentData list are never removed. This allows you as the analyst to view and process the data that the algorithm has seen so far. This is useful when comparing an estimate to the actual value (as is done in the Count-Min sketch).

MySlowDataQueue also provides the utility function randomize(). This function does NOT generate random data. That is done on a per algorithm basis, since MySlowDataQueue does not know very much about the structure of the MySlowData that your algorithm will be using. The randomize function simply randomizes the order of the data in the others list. Arguably this should only be done once directly after data has been generated or loaded, though it may be called at arbitrary times during the algorithm’s execution.

1.1.2 Understanding MySlowData

The MySlowData class requires that each implementing subclass override the default toString() method with some meaningful toString() method. The MySlowData class also provides the default expired() method which is used to test if an item should be added to the posted queue or not. MySlowData also provides a utility method for setting an expiration date in the future called fixDate(long). This simply allows you to ’delay’ a data item by some number of seconds. This method should be overridden in an implementing subclass if finer granularity than seconds is desired.
1.2 Implementing an Algorithm

1.2.1 Writing a Constructor

The constructor is where the initialization of all of the data structures you intend to use happens. Note that this is NOT where the generation of random data, or the loading of data from files should happen. This IS however the place where data structures that are randomized should be generated.

1.2.2 Understanding churn()

The churn method simply calls a series of overrideable functions in the logical order. Churn calls:

- getCanvas()
- canvas.data.updateData()
- canvas.onUpdateSuccess()
- canvas.onUpdateFailure()
- canvas.otherProcessing()

getCanvas()

canvas is an abstract function which simply returns the paintable component that you have used for your visualization. Because you must override getCanvas() you can place your canvas anywhere you would like.

updateData()

updateData() simply tells MySlowDataQueue to post any new items and then returns the oldest new item.

onUpdateSuccess()

This is the code that gets executed every time a new data item is posted. This is where the majority of your algorithm should go. Don’t forget to include a call to repaint() at the end, or your algorithm won’t appear to do very much!

onUpdateFailure()

This is the code that gets executed if the environment checks for new data, but no data is found. This can be used for performing processing that the algorithm does during “down time.” (ex enriching data structures, or sorting)

otherProcessing()

This function is executed regardless as to whether there is new data or not. Generally speaking this method is more for things like timing execution and debugging, but it can be overridden if required by the algorithm.

1.2.3 Courtesy Methods

These methods are not strictly required should be implemented as a courtesy to the user of your visualization.

getInfo()
This method returns a block text which is presented to the user when they consider selecting your visualization. This text should explain which problem your visualization solves as well as anything unique about the specific algorithm that it implements. This text should also explain what the starting parameters are as well as how to control the visualization.

getLauncherDisplay()  
This method returns a container which contains all of the buttons and fields necessary to retrieve input from the user. If for example you are visualizing an approximation algorithm which has a specifiable confidence you can use this method to display a field in which the user can enter this parameter. The ordering and display of your fields are used as is and are not changed by the software. This allows you to access them directly when calling the init(...) method of your launcher class.

1.2.4 Understanding the Paint Method

paint()  
This is arguably the most important function that you need to override. This is where all of the visualization goes on. Specifically this is where code for your visualization goes (as other components like buttons are repainted automatically.)

1.2.5 Data Initialization

genData()  
This function is an abstract function which should be overridden to generate random data. Note that this function is only intended for generating the data in MySlowDataQueue. Any other data, for example a list of random nodes, should be generated when data structures are initialized during init(...).

genBase()  
This is a safety function which simply generates a blank copy of whatever component you decide to use for visualization. This is intended for use in initialization and error handling, but can be overridden by a blank function if desired.

1.3 Creating a Launcher

1.3.1 Understanding the Constructor

Launchers currently work in two phases. The first is where the launcher allows for input from the user before the visualization begins. This code exists in the constructor. I recommend setting default values and setting up error handlers which catch invalid input and substitute default values.

1.3.2 Understanding init(....)

The second phase is the initialization phase. Once the user has started a visualization the init method should be called. This method is in charge of processing the user’s input, creating an instance of your algorithm class, and setting up the screen to reflect whichever commands the user should now have access to. Don’t forget to remove the buttons you added in the constructor!
1.3.3 Writing an actionPerformed(e) method

The actionPerformed(e) method is inherited from ActionListener and is required to make your buttons work. One point that may seem confusing is that there is one actionPerformed method for both the buttons added in our launcher’s constructor, and the buttons added in init(...). Though it is not necessary for a visualization I highly recommend copying the code for “Step” “Finish” and “End,” as these actions provide the user control over the execution of the visualization.

1.4 Tips and Tricks

1.4.1 Graphs

The first tip I would suggest for graphs is to use geometric graphs. These are the easiest to visualize, and generally the easiest to understand. The second tip I would suggest is to remember that color allow essentially a “third” dimension, with “darkness” representing a fourth. The third tip I would suggest is to generate edges in an efficient manner. For example, if you have already randomly generated your nodes is it any less random to simply select all edges of certain lengths? If you choose this path you can speed up your algorithm by “boxing” your nodes. If you were to draw a grid across your plane each square would represent a box. By selecting a certain length for the sides of the boxes, you can ensure that the only edges that can exist within certain bounds must be either between nodes in the same box, or nodes in adjacent boxes. Thus in order to “generate” edges you can simply walk across these boxes performing (with high probability) a small number of comparisons for each of the at most 4 pairs of boxes. This also prevents the possibility of duplicate edges. This method is employed in both SparseSpanner.java and WeightedMatching.java.