# Fresh House of Belair

Version 1

04/12/2020

Group 10

Gilbert Cherrie - cherrieg@mcmaster.ca

Ethan Dhanraj - dhanraje@mcmaster.ca

Christian Hsu - hsuc13@mcmaster.ca

Hasan Kibria - kibriah@mcmaster.ca

Nicholas Lobo - lobon3@mcmaster.ca

2XB3: Software Engineering Practice and Experience: Binding Theory to Practice

McMaster University, Department of Computing and Software

# Revision Page

**Report Revision History**

Mar. 25, 2020:

Decided the responsibilities of all team members to put in the report in the future.

Apr. 1, 2020:

No further progress since March 25th.

Apr. 10, 2020:

Report primarily finished, left certain contribution columns and class descriptions to finish later, as they were undecided as of then.

Apr. 12, 2020:

Final finishes on the report, and collective agreement upon its completion.

**Team members and Their Respective Responsibilities**

| Name/Student Number/ MacId | Roles and Responsibilities |
|---|---|
| Gilbert Cherrie/400085411/cherrieg | Implementing graph algorithms |
| Ethan Dhanraj/400185166/dhanraje | Implementing sorting algorithm, reading data |
| Christian Hsu/400210638/hsuc13 | Implementing search algorithms |
| Hasan Kibria/400200557/kibriah | Implementing data class |
| Nicholas Lobo/400179304/lobon3 | Implementing interface, heatmap |

**Attestation**

*By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.*

# Contribution Page

| Name | Role | Contributions | Comments |
|---|---|---|---|
| Gilbert Cherrie | Developer | Implementing graph algorithms, including the graph, edge and bestState classes. | |
| Ethan Dhanraj | Developer | Implementing MergeSort, ReadState, and WriteCSV modules | |
| Christian Hsu | Developer | Implementing search algorithms. Error correction to other modules as well. | |
| Hasan Kibria | Developer | Reading over modules for general code review, logistical work | |
| Nicholas Lobo | Developer | Implemented gui for heatmap and demo prototype video | |

# Executive Summary

Through the ever-growing threat of climate change on every aspect of human life, many buyers and sellers in the enormous real estate global market are realizing the importance of considering the effects of climate change on their housing decisions. Fresh House of Belair aims to provide a database analysis interface that induces an accurate depiction of air pollution levels and their effects on housing prices within various states throughout the United States of America. In specific, Fresh House of Belair ties a relationship between the two factors previously mentioned and provides the consumer of the software with a dynamic representation of where a user should buy property, based on pollution levels, pricing, and proximity to the ideal states which sit at a maximal in the relative relationship ties between the two factors.

# Table of Contents

# 1. Class Overview:

### 1.1 Interface

Class Description:

This class uses the unfolding maps library as well as processing for eclipse. A heat map was created onto an applet. The data from HPI/pollution csv was then read and then normalized so it could be displayed on the applet changing how the given heatmap looks depending on the data.

Interface Description:

This class has 3 public methods which is its draw setup and main method. The setup method initializes the different location variables for the heatmap for each state as well as set the size of the applet viewer. The draw method draws the heatmap on the applet and is constantly refreshing when state variables are changed. The main method simply launches the applet.

Implementation Description:

This class contains 4 private methods. These four methods are the setlocations , readfile, normalize and normalize2 method. The normalize method is used to change the data into rgb and size values so it can be displayed on the screen. The set location method changes all the location state variables given the new data for a different year. Finally the read file method is used to read the data from the given csv file and save them in state objects.

### 1.2 MergeSort

Class Description:

This class employs the recursive MergeSort algorithm to sort an array of State objects in terms of ratio of HPI/pollution. In this application, a high ratio is preferable emphasizing high HPI and low mean pollution levels.

Interface Description:

The class contains only one public method sort which takes an input array of States. The method sorts the input array in terms of decreasing HPI/pollution ratio.

Implementation Description:

Within the class, the public method sort calls upon private static methods to sort the array of States. There is another sort method that takes the array of States as well as the first and last indices as inputs. This method is called recursively and calls upon the merge method to help sort the array.

**1.3 ReadState**

Class Description:

This class reads from a cleaned and ordered data set containing the American state name and ID, latitude, longitude, HPI, mean pollution levels, and year. The data is stored in a ReadState object containing an array of State objects which are accessed for algorithmic processes.

Interface Description:

The constructor uses an appropriately formatted data set as a scanner as input and returns a ReadState object. The class has two access methods, one that returns a State object for an input state ID and one that returns the size of the array of State objects.

Implementation Description:

The constructor reads from the CSV file line by line. The state name is stored as a string, state ID and year are stored as integers, and latitude, longitude, HPI, and pollution are all stored as doubles. For each line, a new State object is created and stored in an array. The size of the array of States is constant and corresponds to data from a specific year. Thus, to incorporate data from multiple years, several ReadState objects and data sets will be used. Values from the ReadState objects can be used through access methods or copied to a separate array of States.

**1.4 Search**

Class Description:

This class creates an object that contains all the values that would be searched for by a user of a certain American state. This class implements a brute force search algorithm to find the American state and its values to compile them into a list for searching. A binary search algorithm is used to find the specific values a user wants.

Interface Description:

The constructor uses an input dataset and an input American state. The input for the dataset is String since it can be a filepath or title of a file. The input American state is also a string. The constructor can be called for every American state while the input data remains the same. It can be called for the same state but the data will remain the same. The other method named “BinarySearch” will be passed a year stored as an Integer. This method can be called on the class object multiple times as a new year will yield new results.

Implementation Description:

The state variables of the object are the input American state stored as a string and ArrayLists of years, HPI, and APL values. The constructor uses the dataset to fill these lists whenever the American state of the dataset matches the state variable “state”. The values in these lists are accessed in the “BinarySearch” method so they can be used as output however the lists will never change and the American state does not change as well.

**1.5 State**

Class Description:

This class instantiates abstract data types to represent an American state containing data about location, HPI, pollution levels, and year. State objects are the basic storage unit used in sort, search, and graph algorithmic processes using internal data where needed.

Interface Description:

The constructor takes several inputs of the following types: state name as a string, state ID and year as integers, and latitude, longitude, HPI, and pollution are all as doubles. The class contains public access methods to get various state variables. The State class also contains several compare methods returning booleans based on HPI,

pollution, and the ratio HPI/pollution. Finally, there is a method that finds the distance between two states and returns it as a double.

Implementation Description:

The class stores each input parameter in the constructor as a state variable. This includes the state name as a string, state ID and year as integers, and latitude, longitude, HPI, and pollution are all as doubles. The distance method uses formulas to calculate distance between two points on a sphere using latitude and longitude of states in addition to the radius of the Earth. The various compare methods are implemented by comparing state variables for HPI and pollution.

**1.6 WriteCSV**

Class Description:

The class writes to a CSV file from an array of States objects. All the data from the State object is written to the CSV including the state name and ID, latitude, longitude, HPI, mean pollution levels, and year.

Interface Description:

The class contains one public static method writeCSV that takes an array of States and file path as inputs. The method writes to the argument file and does not return an output.

Implementation Description:

The class creates a new file setting the input file path as the destination. A header is first written to the file followed by the State object data which is writedges parsed to a string separated by commas within a line.

**1.7 graph**

Class Description:

This class will construct a graph that has edges from the parameter user State object to every other State object, one for each state. The edges are weighted and directed.

Interface Description:

The client can call the graph class to construct the graph and can then either add an edge if needed or access information like the adjacency list, number of vertices and edges. Can also get a string representation of the graph for testing purposes.

Implementation Description:

This class is implemented using a directed edge weighted graph. The vertices are State objects and the edges are DirectedEdges objects.

**1.8 DirectedEdge**

Class Description:

This class is used to create a directed edge for the graph object. It takes a start and end State object parameter and a weight value for the edge, which is produced using the homeValue() function on the end State with the parameter being the Start state.

Interface Description:

This class returns a DirectedEdge object from the constructor and also has accessors for the start and end State objects of the edge, the double weight value of the edge and a string representation of the edge for testing purposes.

Implementation Description:

This class is implemented using an abstract object which is the DIrectedEdge to store the start State, end State object and the double weight value of the edge between them.

**1.9 bestState**

Class Description:

This class is used to produce the best state for a user to move to based on their current state and every other state in the United States, finding the state with the best hpi to pollution ratio and is not too far in terms of physical distance from the user.

Interface Description:

The constructor for this class is called to produce the bestState object and then the getShortestPath() function can be called on this object returning the optimal state for the user to move to.

Implementation Description:

This class is implemented using Dijkstra's algorithm to find the shortest path on the edge weighted directed graph of the United States starting from the user state and ending in every other state. Dijkstra's algorithm will find the best state by essentially using the lowest weighted path between the user state and all other end states.

# 2. UML Class Diagram

## 2.1 Diagram:

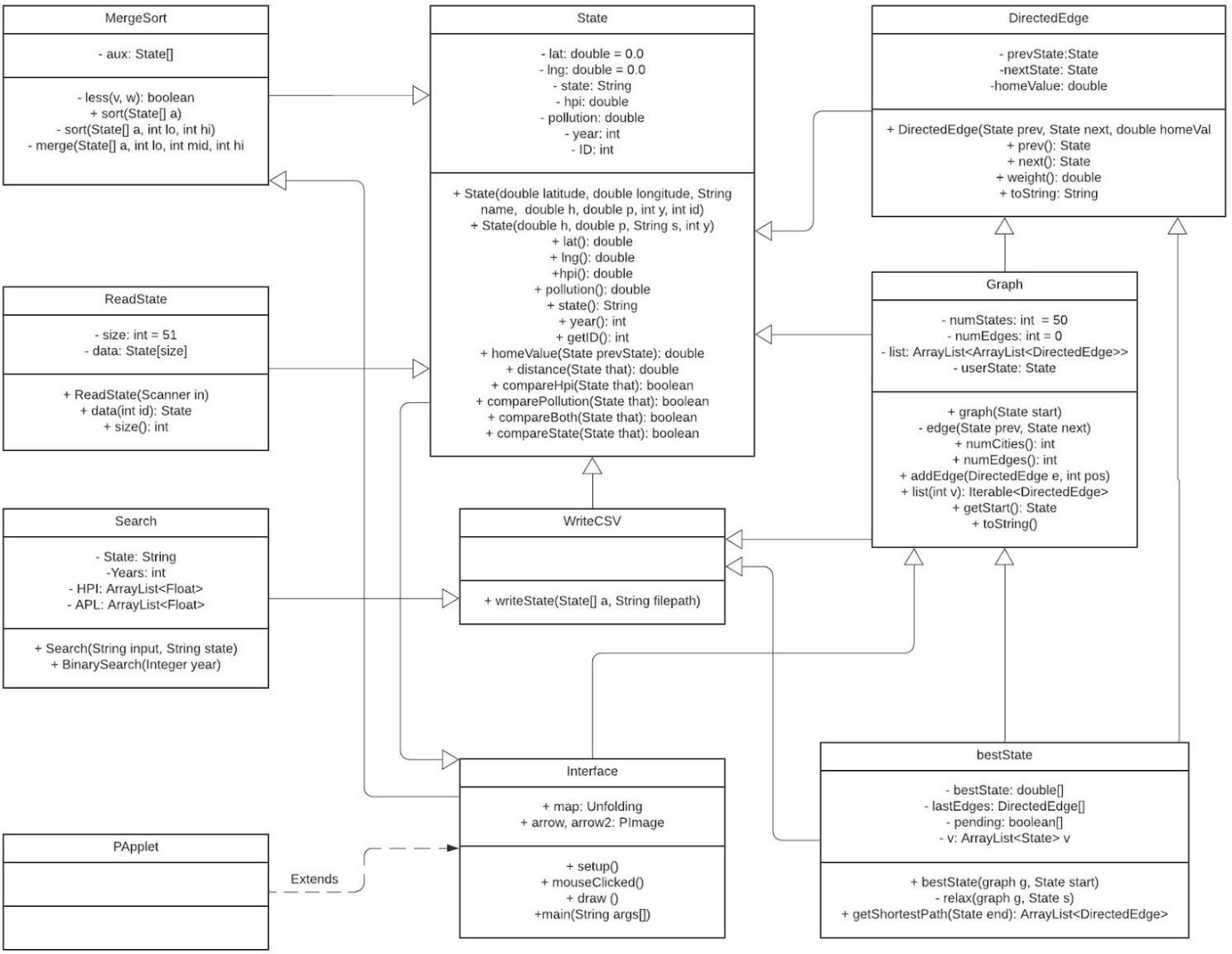


**Figure 1: This a UML Class Diagram of how the desired product would function.**

**2.2 Explanation:**

The product breaks down into the modules demonstrated in figure 1 as they allow for separate implementation and to maintain information hiding. The State class would create objects that stored all the necessary information that needed to be used or accessed. The MergeSort class is a module to implement the sorting algorithm outlined in the requirements document. This class exists apart because the sorting algorithm can be changed (name would need to be changed) but the overall product would not be affected by the sorting. The “graph” class required DirectedEdge to create the graph and the bestState class required “graph”. This grouping of classes are created to implement the graphing and shortest path algorithms outlined in the requirements document as well. If the graph object requires more edges there is a method that allows it to simply add a new DirectedEdge. Also there can be multiple instances of bestState as the starting American state can change. Thirdly, the Search class is created as it’s own module as well in case the searching algorithm changes. Also multiple instances of search objects can be created to be used as well. The output of the search and graphing algorithms are created using a separate CSV writing file which can be seen in figure 2 as WriteCSV. Lastly the interface uses the sorted data by MergeSort and the graph to create a heatmap output. The interface is created by using an external java library represented by PApplet.

# 3. Uses Relationships Visual

### 3.1 Diagram:

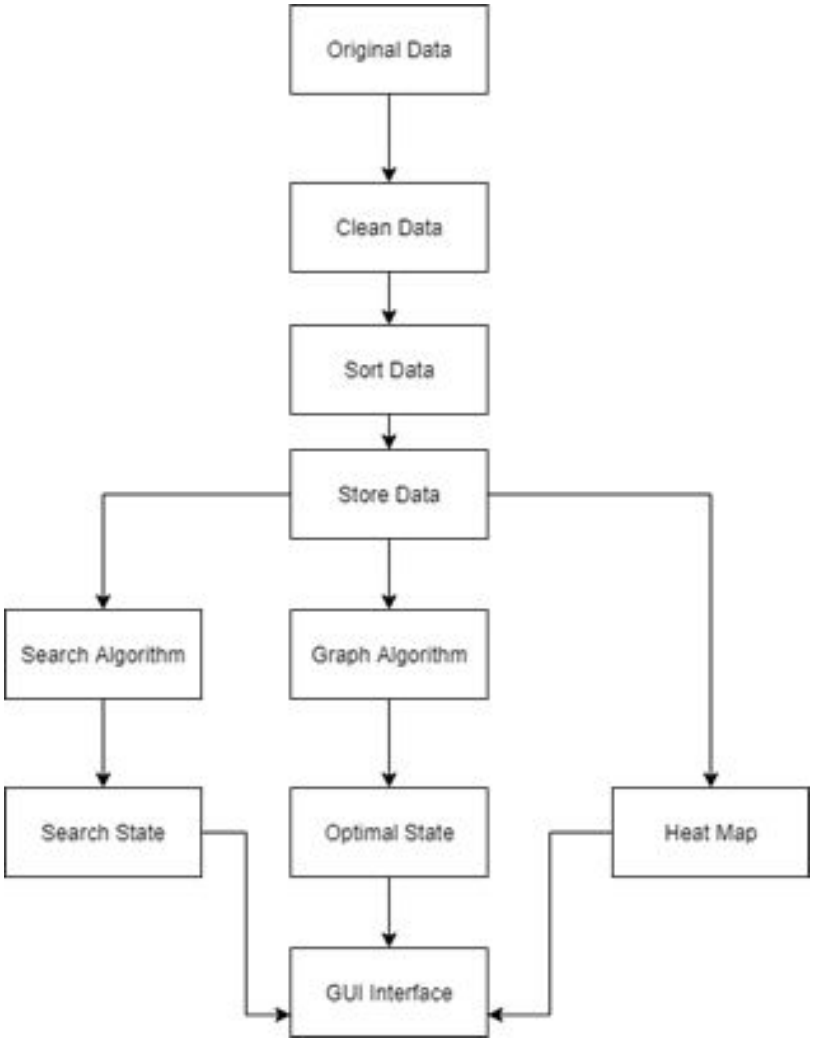


**Figure 2: This diagram shows the flow of the program from the original data to the final GUI interface.**

### 3.2 Explanation:

The uses relation diagram shown here displays the flow of data from unclean data set all the way to end user output. The direction of the arrows shows where the data starts and ends and also what each class or function depends on. Initially it starts with the original unclean data sets. Then the data is cleaned, sorted using the sort class and finally stored in a final data set. Next, the search algorithm, graph algorithm and heat map all rely on or use this stored data without depending on each other. The

search algorithm allows the user to search for a State, the graph algorithm will output the best State for a user to move to based on their current State, and the heat map displays the sorted data in a unique visual format involving a map of the United State and coloured circles of various shades over each State. Finally all these separate output options are combined into the graphical user interface that allows for the user to select for their output type, input the required values and finally display the results.

# 4. UML State Diagrams:

State Diagram for MergeSort class:

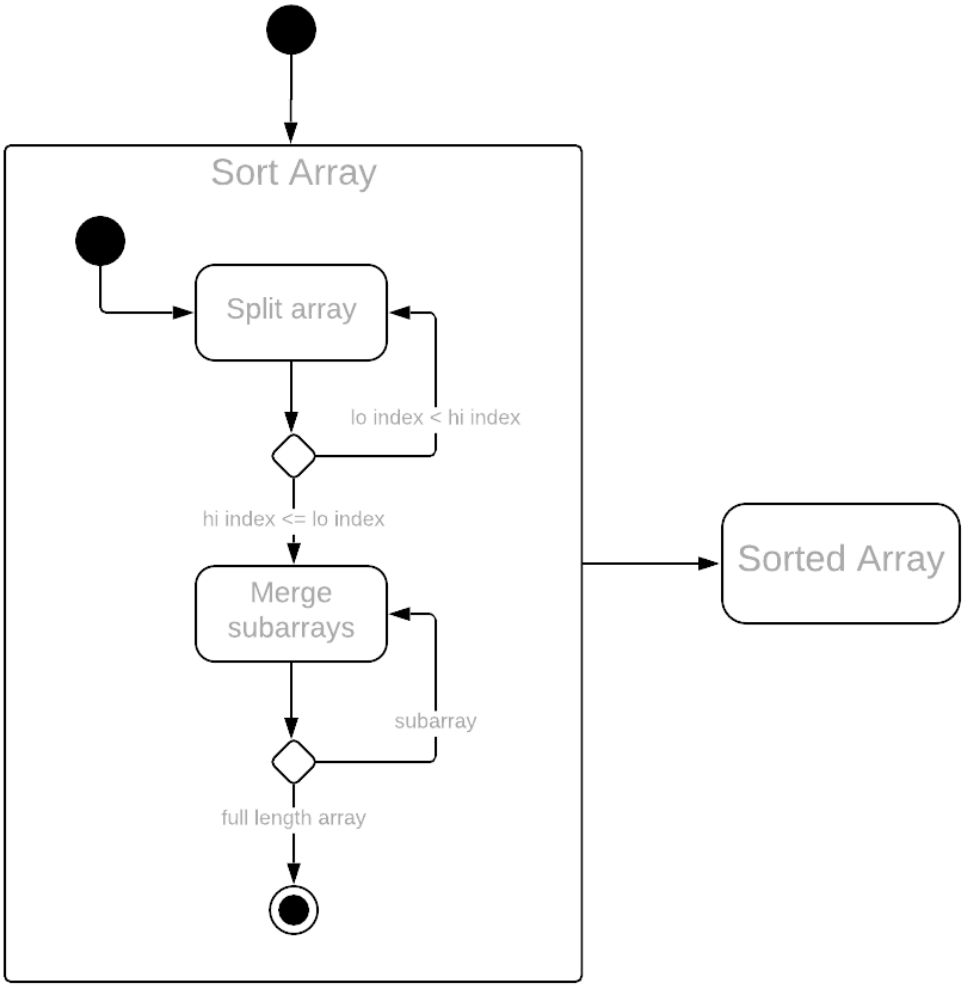


**Figure 3: This diagram shows the UML state machine diagram for the MergeSort class.**

State Diagram for bestState Class:

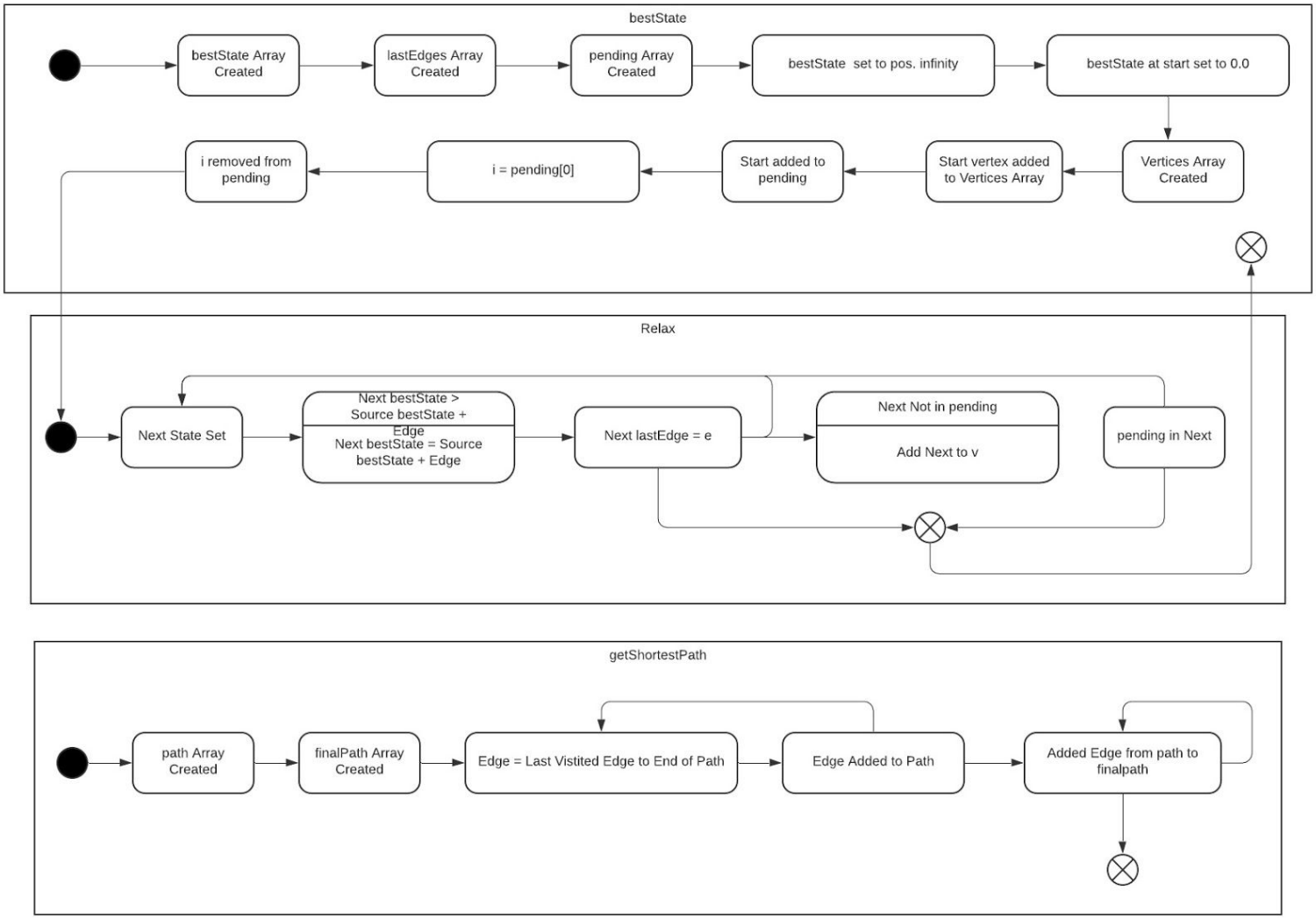


**Figure 4: This diagram shows the UML state machine for bestState class**

# 5. Internal Design Review

The system is designed to visually and numerically illustrate the relation of Housing Price Index (HPI) and mean air pollution levels in the United States. The system is decomposed into several modules supporting data storage, algorithmic processes, and user interface interactions as indicated by the uses and UML diagrams.

There are several strong points of the design, namely its procedure for data storage as well as its preservation of encapsulation and information hiding. The State module is used to store the key pieces of data including the state name, HPI, pollution levels, year, and location. This is used for sorting, graph processes, and the user interface with access routines allowing for pieces of data to be viewed and compared in a wide variety of situations. The design also preserves encapsulation through the use of private methods and state variables and only implementing necessary access routines. Furthermore, the classes with algorithmic processes, MergeSort, Search (uses binary search), and bestState (uses Dijkstra's shortest path) hide the module's secret. Thus, the design of these modules is made to anticipate likely changes.

This iteration of the system design and implementation also has some drawbacks mainly in its reading and processing of data. The original data sets are not used in this iteration because they were uncleaned and of great magnitude. As an alternative, fabricated data representing a clean data set was used to demonstrate the functionality and potential of the system.

The ReadState module is used to read the fabricated data set, data.csv, and store it as a sequence of State objects. The drawback of this module and design is that the data needs to be formatted in a very specific way beforehand compiling multiple data sets. A future change to this process would be the implementation of multiple reading modules that would retrieve fewer pieces of data from more files to reduce the work involved in data cleansing and formatting. Finally, this iteration of the product implementation does not have modules to process data. In future iterations where HPI and pollution is collected from states for multiple years, processing modules would be required to average out and modify these values to ensure their accuracy. In doing so, trends about HPI and air pollution within states can be found and used to guide users.