CSLP Report

s1421803

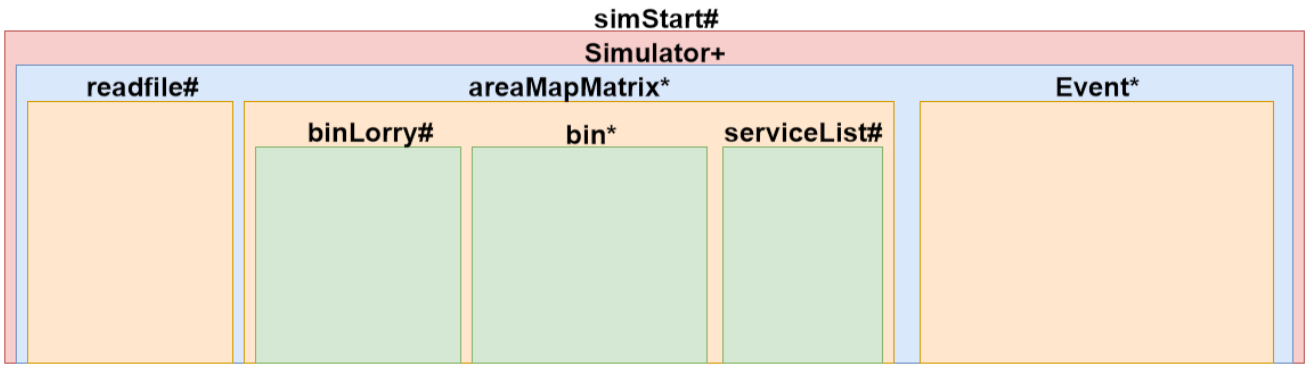
Implementation Approach

One of the first things considered when developing the simulator was whether to contain the simulation in either a single class, or multiple classes. Multiple appropriately named classes was clearly the solution, as it allows for better organisation of data and methods. It also makes the code more readable and extendable. An infographic showing the hierarchical structure is shown below with the following key:

 # = Only one instance created.

** = Multiple instances can exist.*

+ = Multiple instance could exist in future improvements, however current implementation only makes use of one.



Classes Purposes:

simStart:

- Class is used for managing the direct calling of high level simulation functions like input parsing, experimentation management, and calling of the main stochastic simulation method `simOutline()`.
- Serves as interface between command line/simulate.sh and Java application.

Simulator

- Class is used to maintain and house the main event handling simulation. The method `simOutline()` is a recursive loop that calls the methods `determineSetOfEvents()`, `chooseNextEvent()`, and `triggerNextEvent()`.
- The purpose of this class is also to manage and update the main core components of the simulator such as input parsing and validation and event management. Events are the data type used to store possible events. This class also stores all relevant information to the simulator.

Readfile

- Class is used to read files from specified input location.

Event

- Class is used to manage events. This class is essentially a data type that holds minimal information for events to make event choosing simple and manageable at a high level with simulation updating being ‘echoed’ throughout the simulator upon event execution.

areaMapMatrix

- Class is used to manage data specific to areas, such as bins, lorries, services, and route planning. This class communicates directly with Simulator to generate events for `simOutline()`.
- Contains a large amount of the data that the simulator uses. Could have potentially be broken up into more sub classes for easier readability and extensibility .

binLorry

- Class is used to store data on the current status of the lorry. For example information includes: current weight, current volume, last bin visited, next bin to visit, or if it’s currently servicing a bin.

Bin

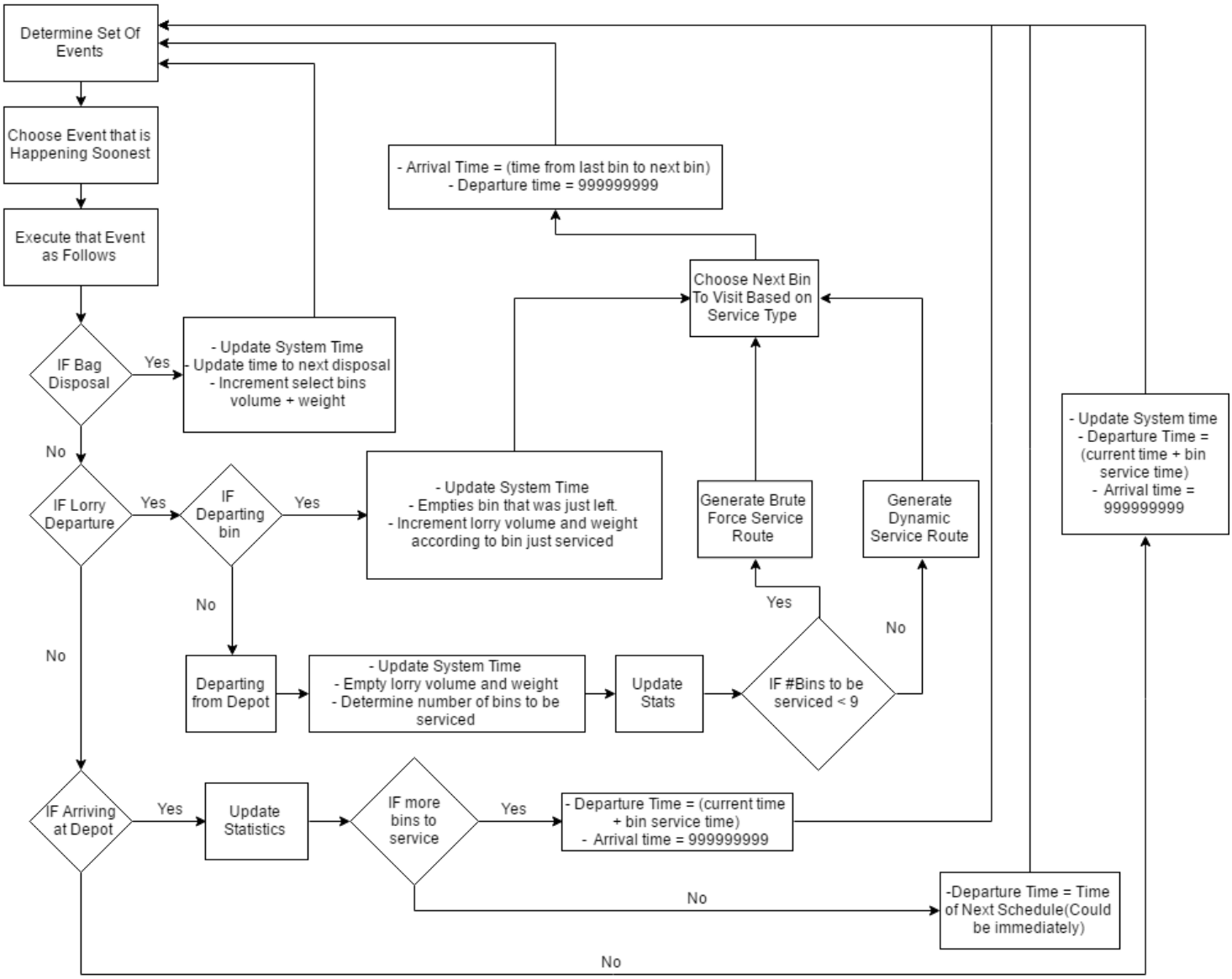
- Class is used to store data on the state of a specific bin. There will be instances of this class within the areaMapMatrix. Methods in this class use Erlang-k Distributions to determine the time between bag disposal events, and the class can be consulted by the Simulator class to find out the delay until the next disposal.

serviceList

- Class is used as a datatype to store a list of services.

Event Generation and Execution

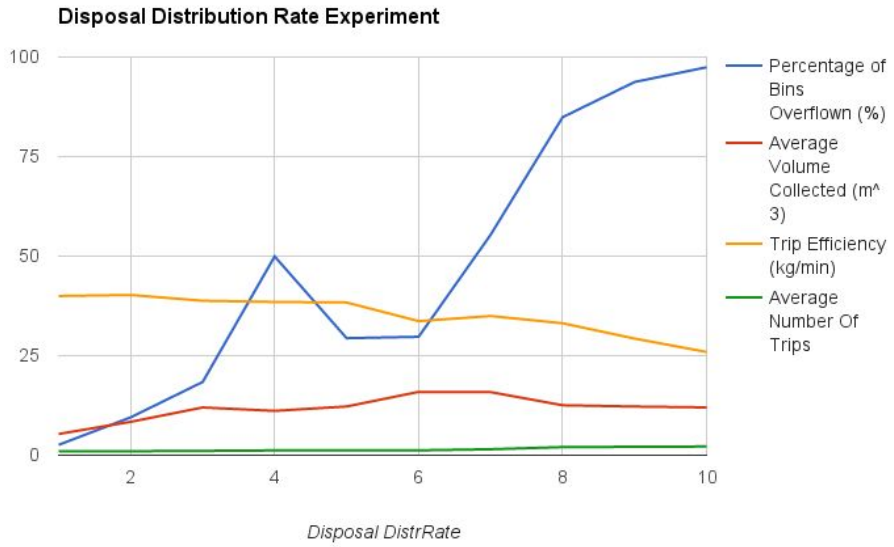
The stochastic simulator works by jumping between events until the simulation time has elapsed. The possible events are: Bag disposed in bin; Lorry departs from bin; Lorry arrives at bin. This means that services are started when the lorry departs from Bin 0, with certain flags raised, and that trips are also started when departing from Bin 0 with different flags raised. Upon starting the simulation, the next scheduled departure is calculated based on how many bins are needing serviced. If no bins are needing serviced, then the time until the next departure is set to the next service time(So if services are scheduled every 3 hours, the departure time will be set to 3 hours into the simulation). The time until arrival is then set to 1 second after this arrival, just to ensure that an arrival event cannot happen until a departure event has happened. Similarly, when bin classes are constructed, a method is called to find out the time of the next bag disposal in that bin. The event selecting process then selects the Event that will happen first, and updates the time in all classes of the simulator, and updates appropriate values. The flowchart below properly depicts how the Event generation, choosing, and executing process works.



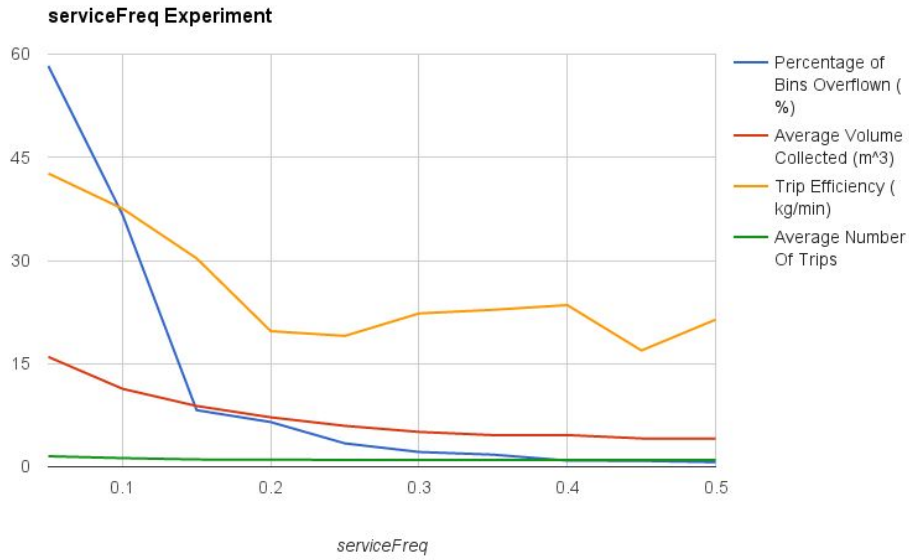
Path Finding and Route Planning

Route planning techniques such as Sorted Edges, Nearest Neighbours, and Brute Forcing were considered when deciding how to create an efficient service schedule for the Lorry to follow. Although Sorted Edges couldn't be explored and tested due to time constraints, the final implementation consists of a combination of both Brute Forcing, and Nearest Neighbours. Upon initialisation of an area, a new roadsLayout is created, where the element (i,j), is the shortest time it takes to navigate from *bin i* to *bin j*. This uses the Floyd-Warshall algorithm for computing shortest paths between all nodes of a directed and weighted graph. This option was chosen over other approaches such as Dijkstra's Algorithm, or Bellman-Ford, as it serves as one time use algorithm, which can benefit both the graph traversal algorithms later implemented. When a new service is to be started, the method `setUpService()` in areaMapMatrix is called. This calls a series of functions which ultimately compile an ArrayList of the bins needing serviced. If the number of bins is less than 9, then a brute force approach to traverse the graph is taken, where all permutations of ways of visiting all the bins, and ending at the depot are calculated, and then run through. This ultimately chooses the route with the lowest time cost. If there are 9 or more bins to service, then a nearest neighbour algorithm is implemented, referring to the Floyd-Warshall generated roadsLayout to decide the next bin. Slight tweaking and testing found that on my system, choosing 9 as the threshold between Brute Forcing and Nearest Neighbours produced runtimes that were not excessive.

Experimentation



Altering the Disposal Distribution Rate had a clear impact in the percentage of bins which overflow. The sharp spike at 4.0 may be due to a critical value, due to discrete nature of trips. The critical value may be so that moments before scheduled services, no bins have exceeded the threshold, and so by the time it gets to the next scheduled service. A large number of bins will have overflown.



Experimenting with service frequency just shows that the more frequently services are started, the less efficient trips are, and that less bins overflow. Further experimenting with disposalDistrRate and disposalDistrShape shows that they are very delicate, and even 1 integer change in the Shape can result in drastic changes to statistical values such as Percentage of Bins Overflown.