Computer Science Large Practical Report

Traiko Dinev <s1448355>

Tue, 20 Dec 2016

1 Introduction

This report describes the simulator of a bin collection process in a city, as described here (`http://www.inf.ed.ac.uk/teaching/courses/cslp/handout/cslp-2016-17.pdf`).

2 Simulator Architecture

The simulator has a decouple, observer pipeline, where everything is governed by an *EventDispatcher* that schedules *Events* and calls attached *Observers*. The *Areas*, the *StatisticsAggregator* as well as the *OutputFormatter* all have observers that listen to events happening. This way the output functionality and the statistics operate purely on the events happening and have no knowledge of the rest of the system, making them easier to develop and maintain.

See Figure 1 for more details. The *ExperimentManager* class creates one *OutputFormatter*, *StatisticsAggregator* and *Simulation* and uses them for all runs necessary.

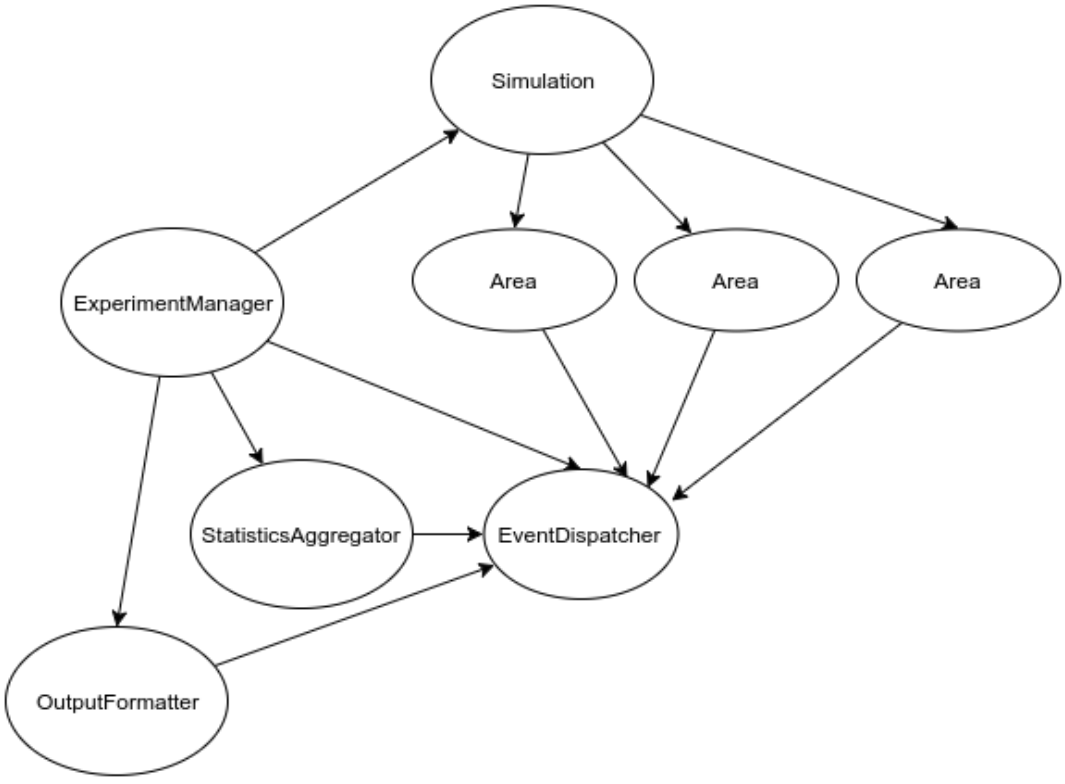


Figure 1: Project Architecture
file: resources/architecture.png

2.1 Simulation Runs and Experiments

Each **run** of the program is governed by an *ExperimentManager*, where actual experimentation behaves identically, while disabling detailed output (by not attaching an *OutputFormmater* to the dispatcher.

As such, each experiment contains a single *Simulation*, which contains as many *Area*s as necessary. Each *Area* listens to events that happen in it and reacts accordingly. It also registers new disposal events and service events, as well as lorry events. Note the algorithm itself is separated and injected as a dependency in the constructor of the *Area* class, thus separating that functionality out.

2.1.1 Pipeline

The input parser is the starting point of the application. Once we get a valid configuration file from the parser, we instantiate the *ExperimentManager* class, which is in charge of creating each *Simulation*, which create *Areas*, which contain the actual simulation code. To glue things together, the *EventDispatcher* is injected in the constructor of the Simulation, which in turn injects it into each area class. Statistics and output are achieved similarly. The *EventDispatcher* class is injected into an *OutputFormatter* and into a *StatisticsAggregator* class.

See Figure 2 for a sequence diagram illustrating the start of the simulation and a cycle of events.

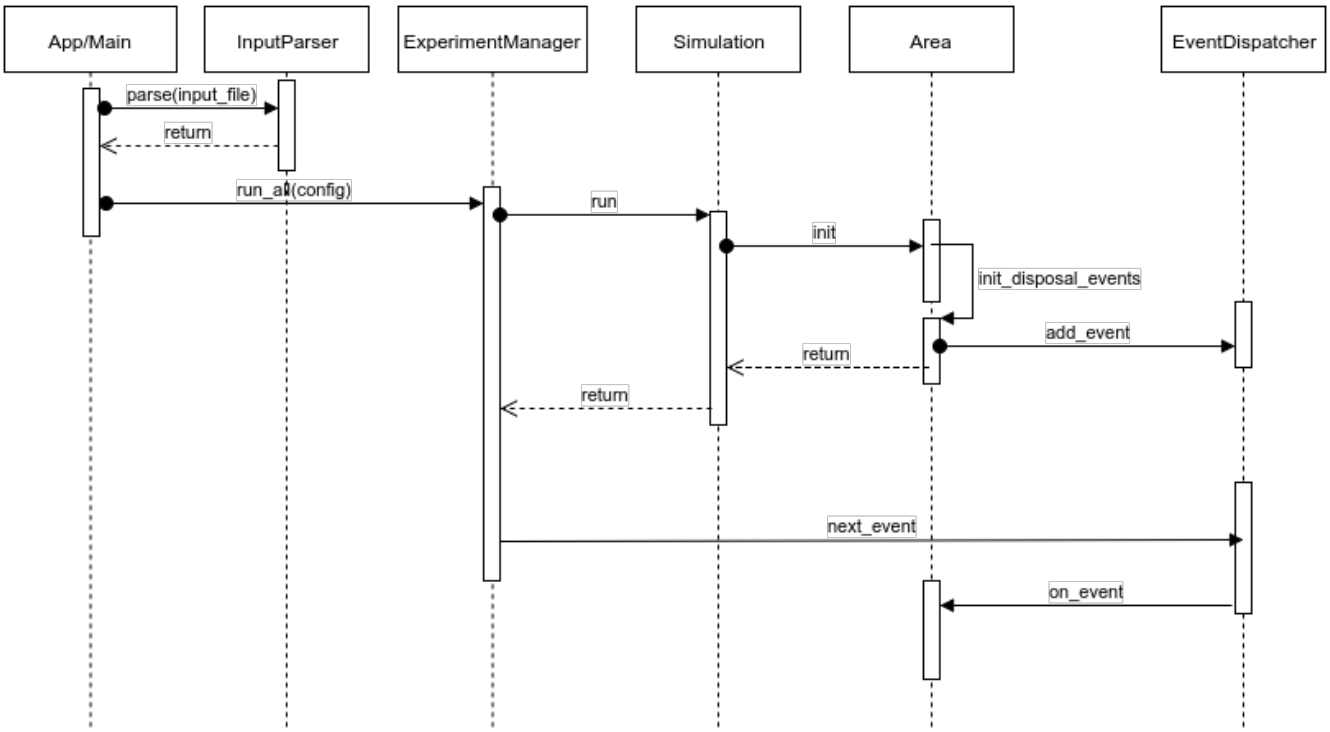


Figure 2: Startup Sequence
file: resources/start_sequence.png

2.2 Route Planning

Route planning is done separately in the **DijkstraRoutePlanner** class. The route planner uses Dijkstra's algorithm each time a route is needed and two different heuristics for selecting the next bin to be serviced. The *greedy* algorithm always services bins according to their occupancy (or volume). The *priority* algorithm always services the closest bin. Both always use only the bins that need servicing (i.e. they do not ever calculate a path that would empty a bin that has not exceeded the occupancy threshold).

2.3 Optimization

The greedy algorithm is generally faster than the priority algorithm and hence the default setting of the algorithm it *dynamic*, which selects one of the two versions based on how many bins are to be serviced. The algorithm also uses caching to save already calculated paths so that they are not computed twice. The cache has a limit (TODO) and all of the above settings can be overridden via command-line parameters.

2.3.1 Caching

See figure 3 [1] for effects of caching on the performance of the algorithm. All tests were done on a small area (5x5), a medium sized area (50x50) and a big area (300x300). All areas run for 100 hours and are otherwise a derivation of the *basic_input*. The simulator was run with the *-d -b* options to produce timing details and disable output. For any test, we take the average of 10 results, since the simulation is stochastic. Outliers were not removed in this scenario.

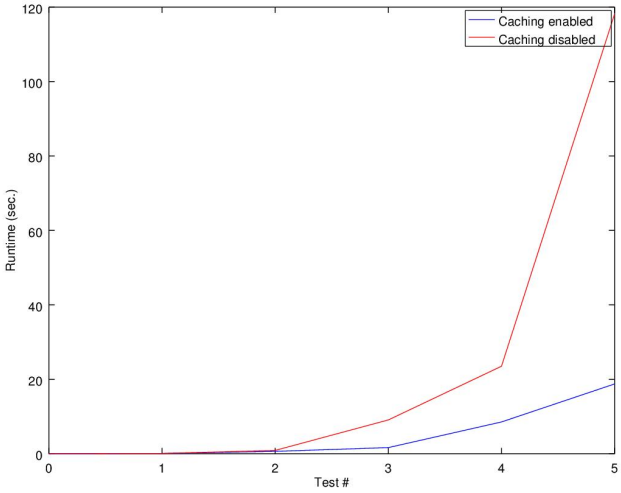


Figure 3: Performance with/without caching
file: resources/caching_performance.jpg

2.3.2 Cache size

Caching size also makes a significant different, but only up to the number of possible paths. We do not let variable cache sizes, since that is too hard to benchmark. Instead, we cap the cache size at $100,000$. This accomodates for a total number of $\frac{ceil(sqrt(100000))}{2} = 159$ vertices all connected to each other (all possible paths). However, a map of 159x159, which has every vertex connected to every other is **very** unlikely. Below in Figure 4 you can see the effect of cache size on a map of 300x300 with several experiments, average of 3 results taken for each cache size.

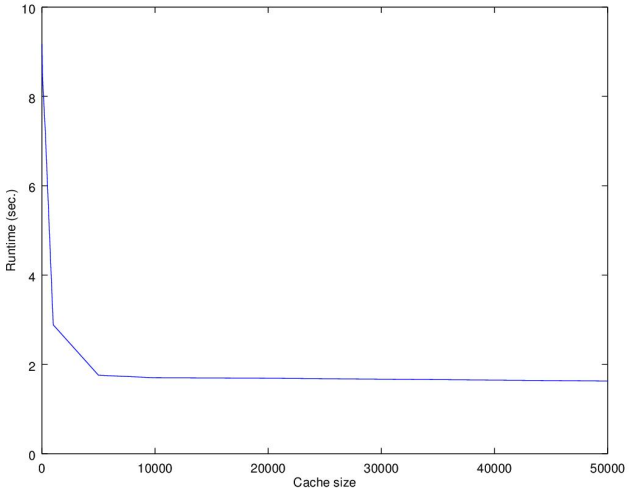


Figure 4: Effects of cache size on performance
file: resources/cache_size.jpg

[1] check 4.2.2 for description of each test

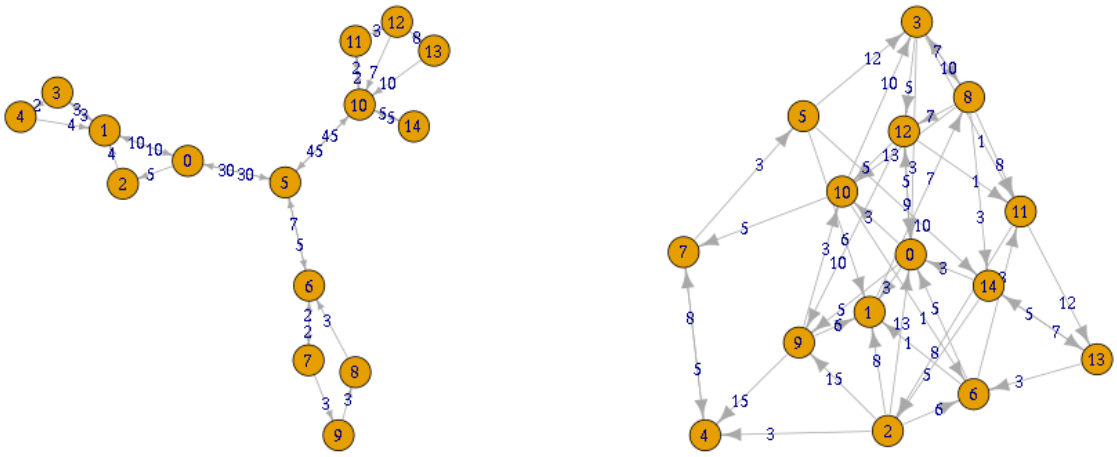
3 Simulation Results

3.1 Test inputs

To evaluate the simulation, two distinct input files were used. They model two different realistic scenarious. The first one (*Three Neighborhoods*) [2] models three separate connected neighborhoods, where the paths within the neighborhoods are shorter than the path between them. (See figure 5a)

The second input (*Big Cluster*) [3] models one big area where all of the bins are interconnected and paths within it are comparable in distance/duration. (See figure 5b)

We expect the priority algorithm, which selects bins based on their distance from the current location, to work better on the *Three neighborhoods* area and for them to work comparably well on the big cluster area.



(a) Three Neighborhoods
file: resources/three_neighborhoods.png

(b) Big Cluster
file: resources/three_neighborhoods.png

Figure 5: Areas used for evaluation

3.2 Bin Occupancy Evolution

The first metric of interest is the bin volume plotted against time. In figure 6a and figure 6b we can see the average volume of the bins for the *Three Neighborhoods* and *Big Cluster* areas. In both $disposalDistrShape = 3$ (3 bags per hour) and $serviceFreq = 0.04167$ (one service daily), but in 6a $disposalDistrRate = 5.0$, while in 6b $disposalDistrRate = 3$. Both of the simulations stop after a week with a warm-up time of a day and both have a bag volume of 0.05 with a maximum bin volume of 2.

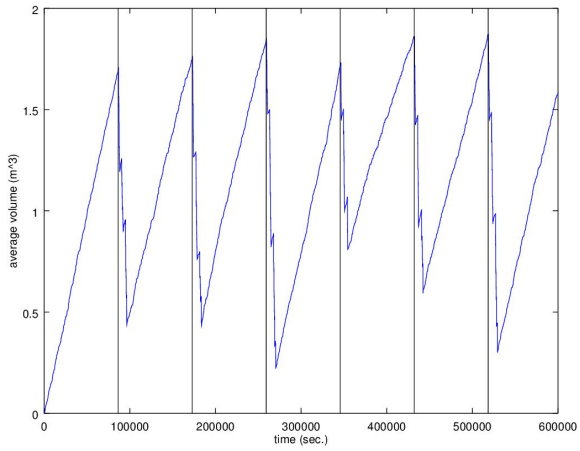
Since the mean of the Erlang-K distribution is $\mu = \frac{k}{\lambda}$, in each case we would expect the average bin content before each service period to follow:

$$bin\_content = \frac{time\_between\_service\_periods}{\mu_{erlang}} * bin\_volume \tag{1}$$

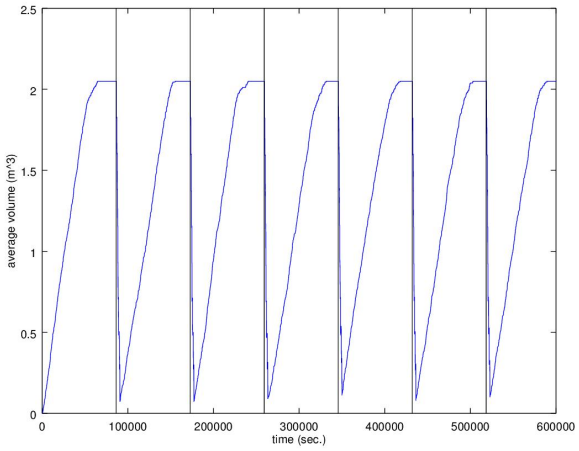
Given that all bins are independent[4]. This yields 1.8 m^3 average volume for *Three Neighborhoods* and 3 m^3

[2] File: test/inputs/statistics_collection/three_neighborhoods.txt
[3] File: test/inputs/statistics_collection/big_cluster.txt
[4] By the specifications

for *Big Cluster*, which is what is observed in the graphs. Note, however, that the bins cannot have a maximum volume of 2 m^3, therefore on the second graph we see the "cap".



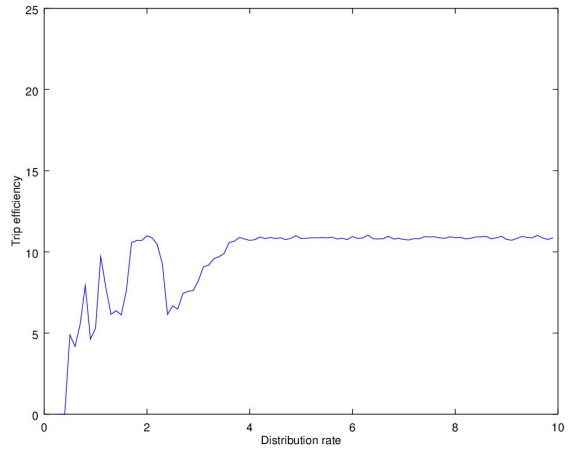
(a) Bin Volume Evolution (Three Neighborhoods)
file: resources/occupancy_three_neighborhoods.png



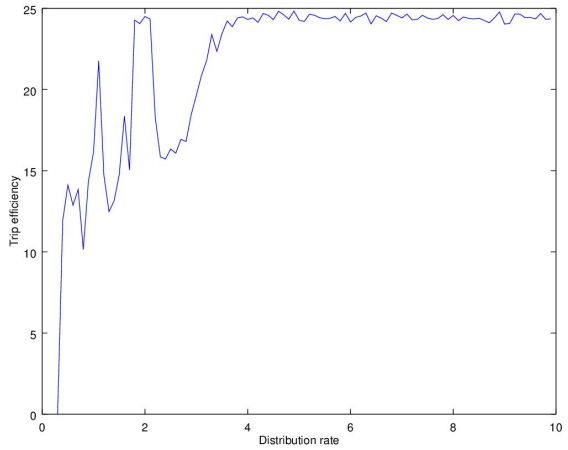
(b) Bin Volume Evolution (Big Cluster)
file: resources/occupancy_big_cluster.png

3.3 Trip Efficiency

The first metric is trip effiency. For figure 7a, 7b we hold $distributionShape = 2$, $serviceFreq = 0.0416$ (once a day) and vary *distributionRate*, recording the trip efficiency for both *Three Neighborhoods* and *Big Cluster*.



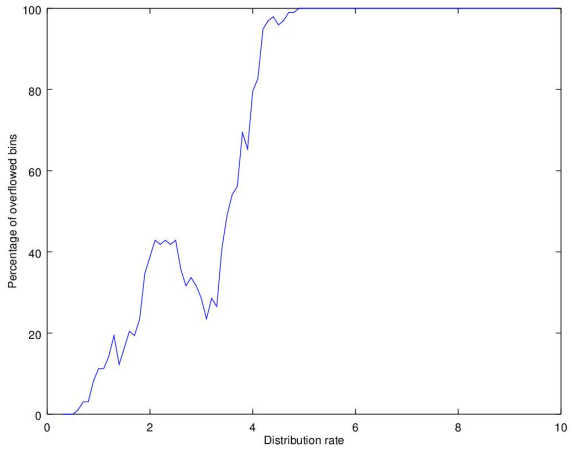
(a) Three Neighborhoods
file: resources/three_neighborhoods_dist_rate_trip_eff.jpg



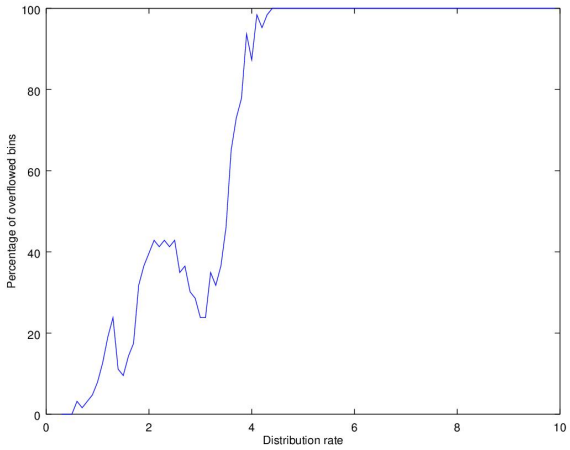
(b) Big Cluster
file: resources/big_cluster_dist_rate_trip_eff.jpg

Figure 7: Trip Efficiency/Distribution Rate

3.4 Bin Overflow



(a) Three Neighborhoods
file: resources/three_neighborhoods_dist_rate_overflow.jpg

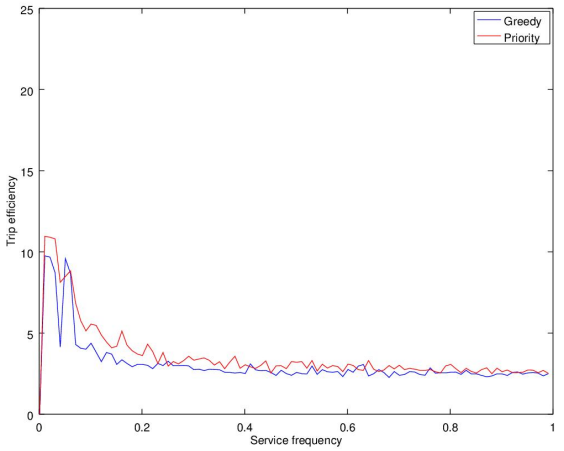


(b) Big Cluster
file: resources/big_cluster_dist_rate_overflow.jpg

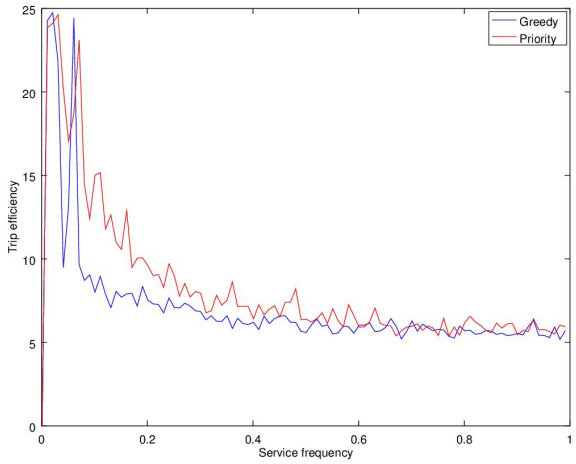
Figure 8: Bin overflow percentage/Distribution Rate

3.5 Algorithm Efficiency

The last metric is the efficiency as archieved by the greedy and priority algorithms. We measure efficiency against service frequency, holding $distributionShape = 2$ and $distributionRate = 3.0$. We would expect trip effiency to go down as the service frequency increases, but also the priority algorithm to perform better than the greedy one on the *Three neighborhoods* area.



(a) Three Neighborhoods
file: resources/three_neighborhoods_service_freq_trip_eff_algo.jpg



(b) Big Cluster
file: resources/big_cluster_service_freq_trip_eff_algo.jpg

Figure 9: Effiency/Service Frequency

3.6 Analysis

Trip efficiency increases as the distribution rate increases. They are positively correlated, though as the lorry reaches its maximum efficiency, no matter how much more bags are disposed and bins overflow, the efficiency doesn't increase.

$$\sigma(dist\_rate,\ efficiency)\ = 0.66919 \tag{2}$$

Naturally, as we add more garbage to the system, the bins start to overflow more, thus trip efficiency alone is a poor metric of the overall efficiency of the system. We can see that bins overflowing also correlate with the distribution shape (and more strongly so than efficiency):

$$\sigma(dist\_rate,\ percentage\_bins\_overflowing) = 0.87768 \tag{3}$$

We also note that trip efficiency is generally poor on areas such as *Three Neighborhoods*, since the distance between neighborhoods is much bigger that the one within a neighborhood and a lorry servicing more than one neighborhood would have to make trips around the long edges in the graph. For this reason, however, **priority** Dijkstra performs better than its **greedy** counterpart. Since making a trip to a different neighborhood is expensive, visiting bins close together first is generally a good idea. However, even the smarter algorithm cannot do anything when the bins that need servicing are spread out between areas, which might explain why the difference in efficiency is not that big. An intersting thing to note is that as the service rate goes up to one an hour almost no bins will need servicing, hence at the very extreme both algorithms would achieve the same efficiency (the extreme being only one bin that needs servicing at every schedule).

4 Testing

The majority of testing is automated using *pyunit* and separate test cases for each class in the application. Performance testing was done somewhat differently, where output was disabled via a command line flag (*(*-bod)) and multiple runs were performed using specially generated files (as discussed below). In addition, the aforementioned *Three Neighborhoods* and *Big Cluster* areas have 2 input files each, describing just the area or the area with experiments, as used in the previous section for analysis.

4.1 Automated Testing

Automated testing is divided into a single test suite (single file) that tests one class and within that test suite many test cases. Here is an outline of the test cases, for more information each suite has appropriate documentation. The location of all unit tests is *test/¡name¿_test.py*.

- *area_test.py* Tests area functionality, including disposal events generation, route planning and proper usage of the *EventDispatcher* class. Includes edge cases for lorry overflow and resheduling.
- *dijkstra_route_planner_test.py* Test the route planning algorithm, both the greedy and priority version.
- *event_dispatcher_test.py* Tests that the *EventDispatcher* class stores and execues events in the correct order.
- *input_parser_test.py* Contains various tests for the *InputParser*, including valid and invalid files. All of the files in *test/inputs/parser_tests* are used here.
- *output_formatter_test.py* Tests that the output formatter correctly displays events by watching the standard output.
- *statistics_aggregator_test.py* General statistics tests for the aggregator class.

4.2 Sample inputs

4.2.1 Input Parser

Sample inputs are in *test/inputs/parser_tests*. Check the corresponding test suite for a morecomprehensive description of each test and expected results.

4.2.2 Performance testing

The above performance tests (section 2.3)) were all done using inputs in *test/inputs/performance_tests*. Each area input was generated via the script file *test/generate_performance_test.py*. Performance testing was done using three area sizes and experimentation.

The following are the tests (ordered by *Test #*), as performed in ascending order (from fastest to slowest).

0. *small_area.txt* 5x5 area, no experiments
1. *small_area_experiments.txt* 5x5 area, 4 experiments
2. *medium_area.txt* 50x50 area, no experiments
3. *medium_area_experiments.txt* 5x5 area, 4 experiments
4. *big_area.txt* 300x300 area, no experiments
5. *big_area_experiments.txt* A 300x300 area, all vertices connected, 4 experiments. Runtimes should be slowest here.

4.2.3 Statistics collection

The tests used in Section 3 are all in *inputs/statistics_collection*. There are two files for each area, one with experiments and one without. Since more than one experiments were run, both files contain all the experiment data. If you wish to replicate, a good idea might be to comment some of the experiments out.

5 Conclusion

The algorithm used for this simulation performed reasonably well and with caching can scale effectively. In a small to medium scale scenario the simulator has reasonable runtimes and no issues dealing with complicated networks. However, a different approach using A^* and a custom heuristic, perhaps combining both of the heuristics used here could yield better results. Additionally, it would be beneficial if the simulation was run on areas derived from actual maps of cities and the above statistics recorded. For larger areas, the simulation could benefit from a parallel architecture, where different areas are run on different threads/simultaneously.

6 Resources

- Graphs plotted using *R*, other plots using *Octave*. All scripts used for plotting are in */statistics_plots*