Computer Science Large Practical Report2

Traiko Dinev <s1448355>

1 Aim

Aim

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.

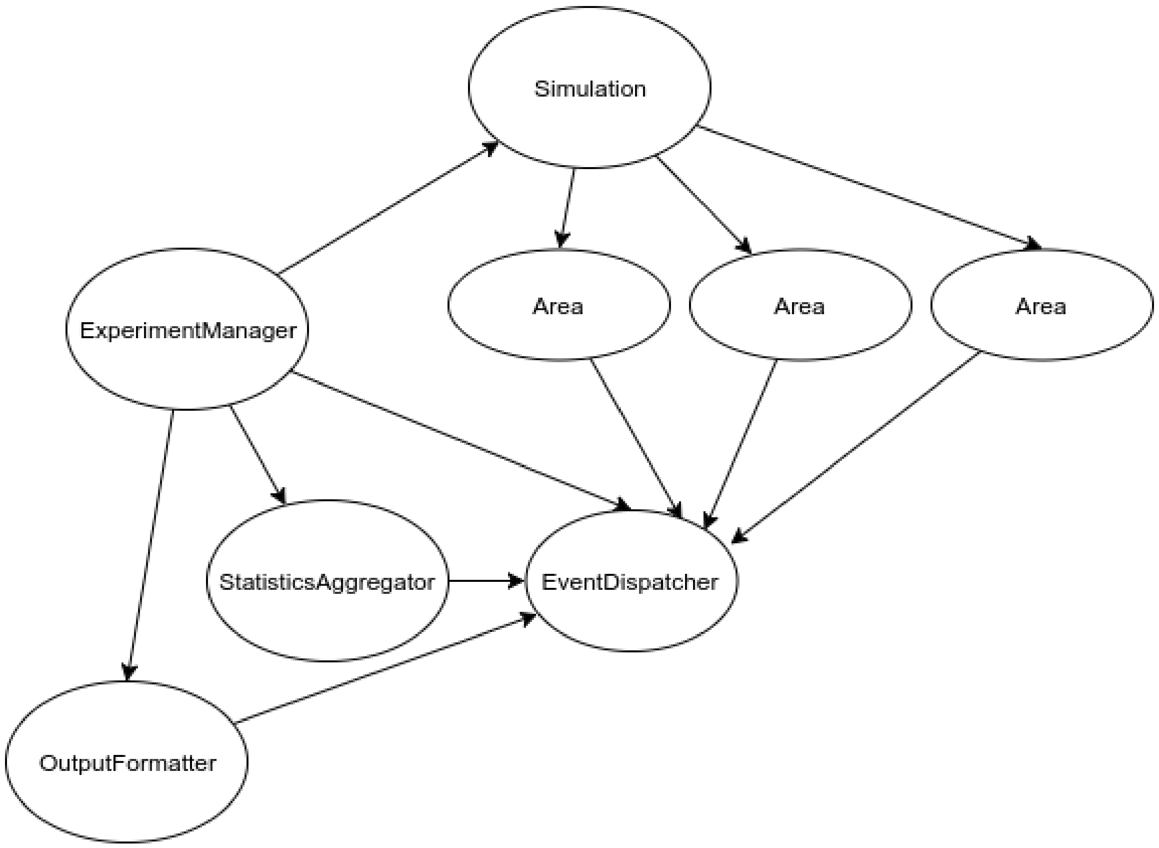

Figure 1: Project Architecture
file: resources/architecture.png

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.

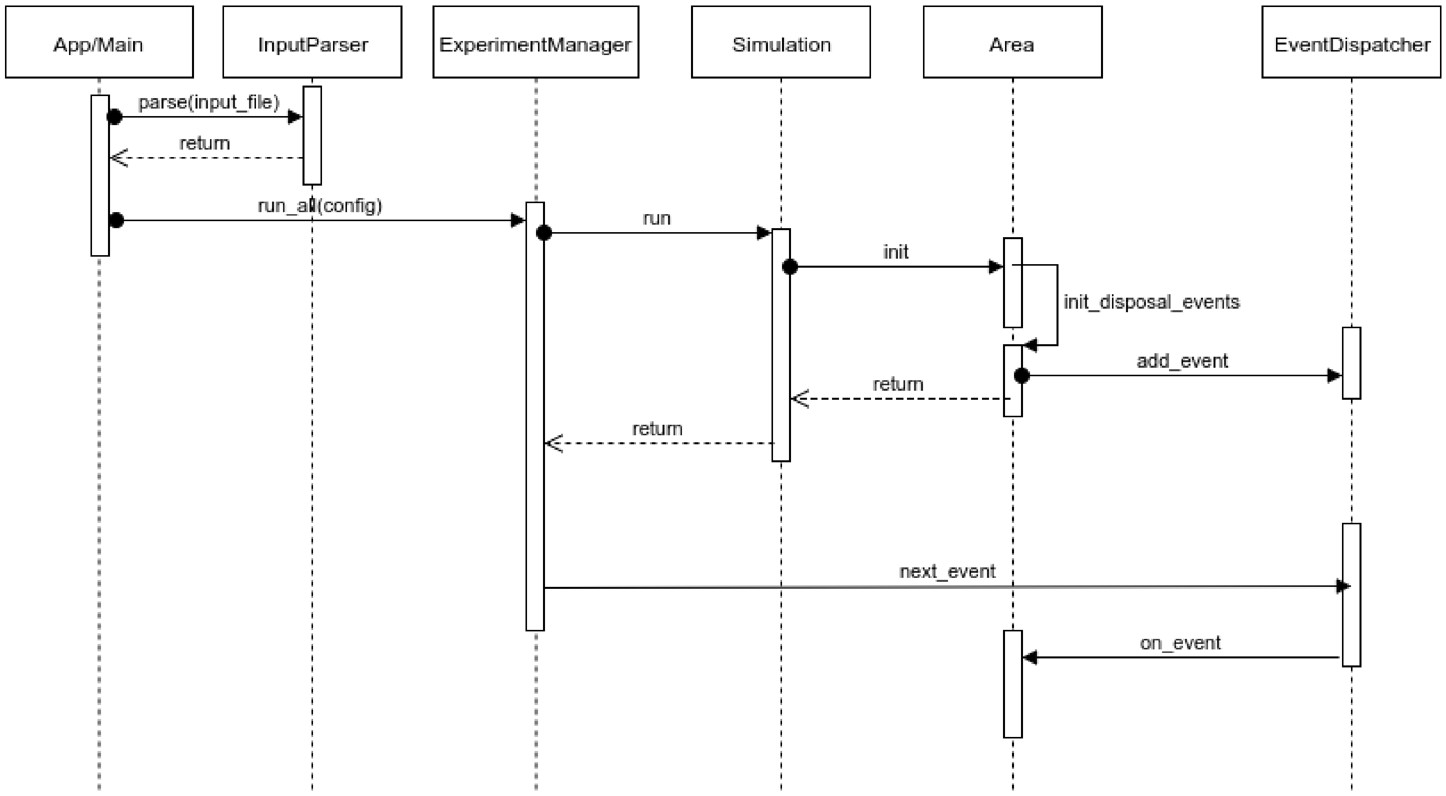

Figure 2: Startup Sequence
file: resources/start_sequence.png

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 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.

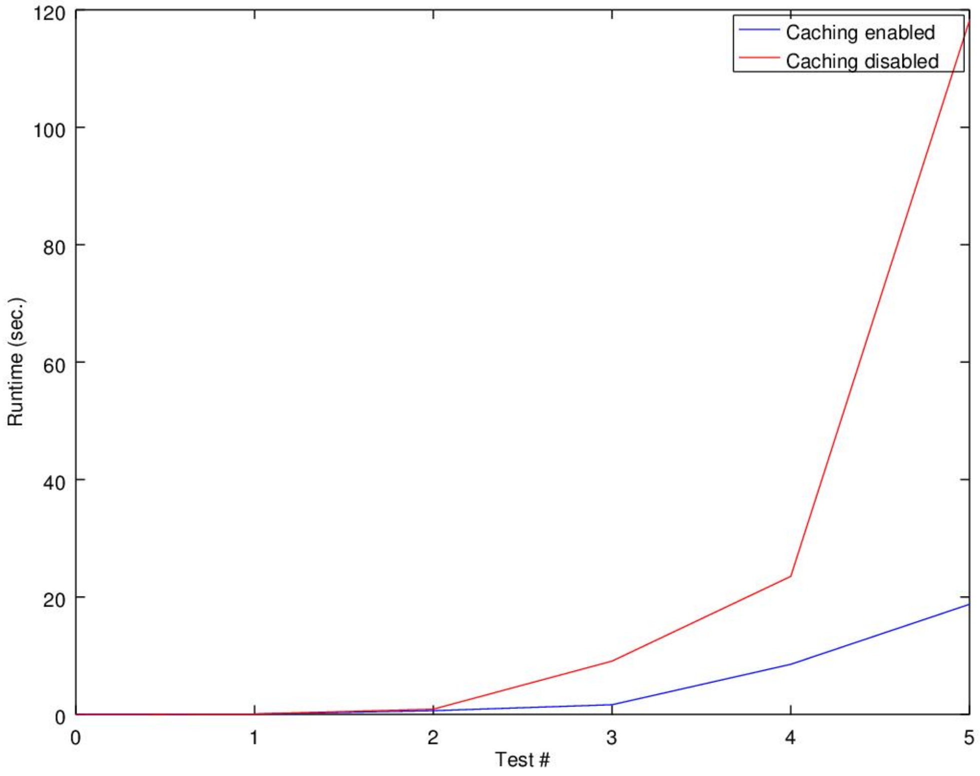

Figure 3: Performance with/without caching
file: resources/caching_performance.jpg

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 7 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.

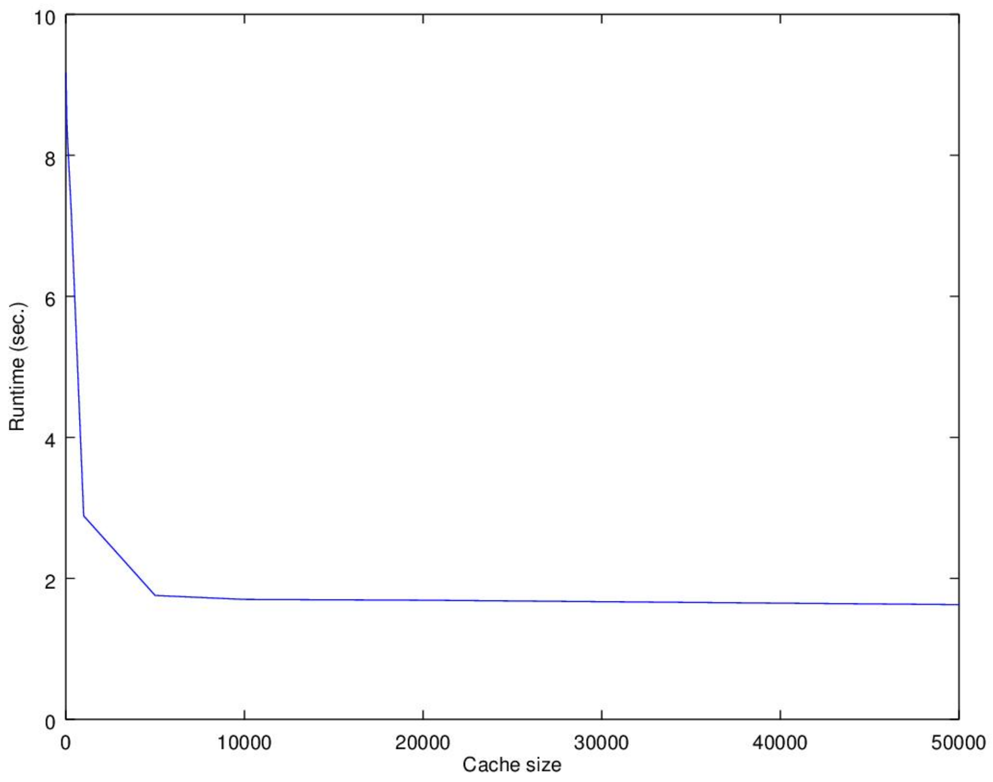

Figure 4: Effects of cache size on performance
file: resources/cache_size.jpg

3 Simulation Results

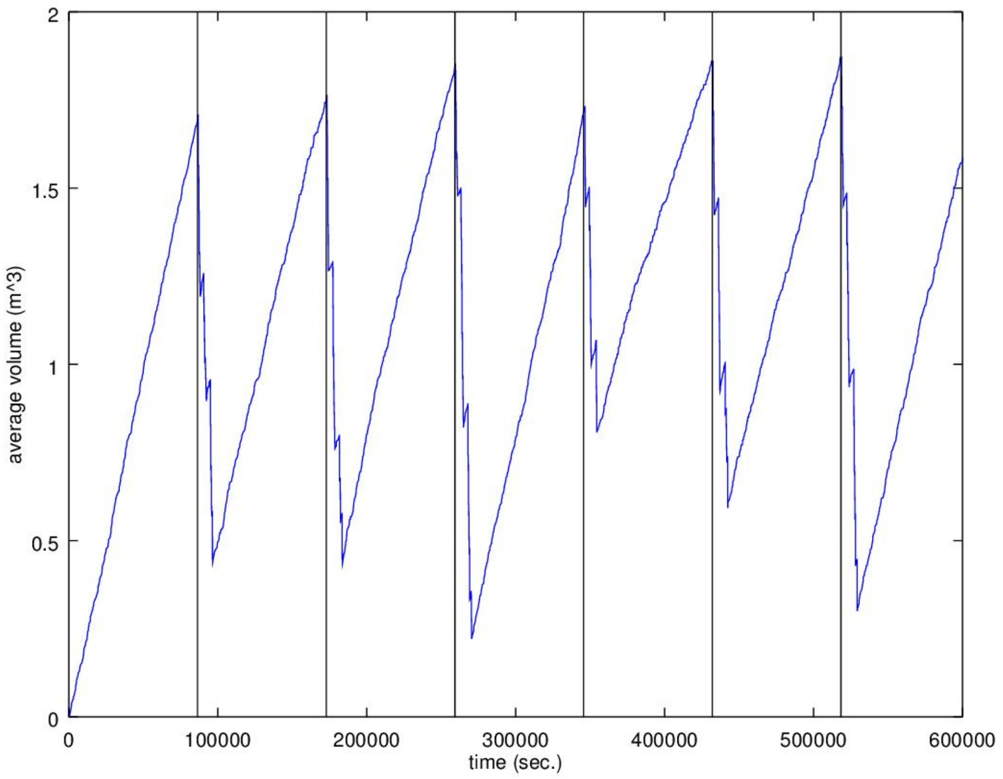

Figure 5: Effects of cache size on performance
file: resources/cache_size.jpg

3.1 Test inputs

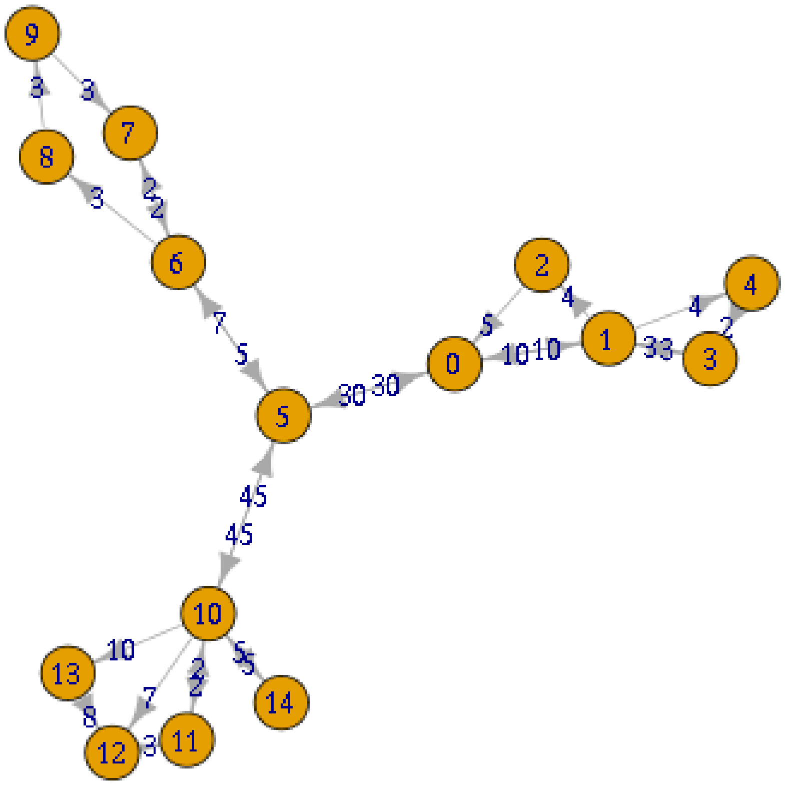

Figure 6: Effects of cache size on performance
file: resources/three_neighborhoods.png

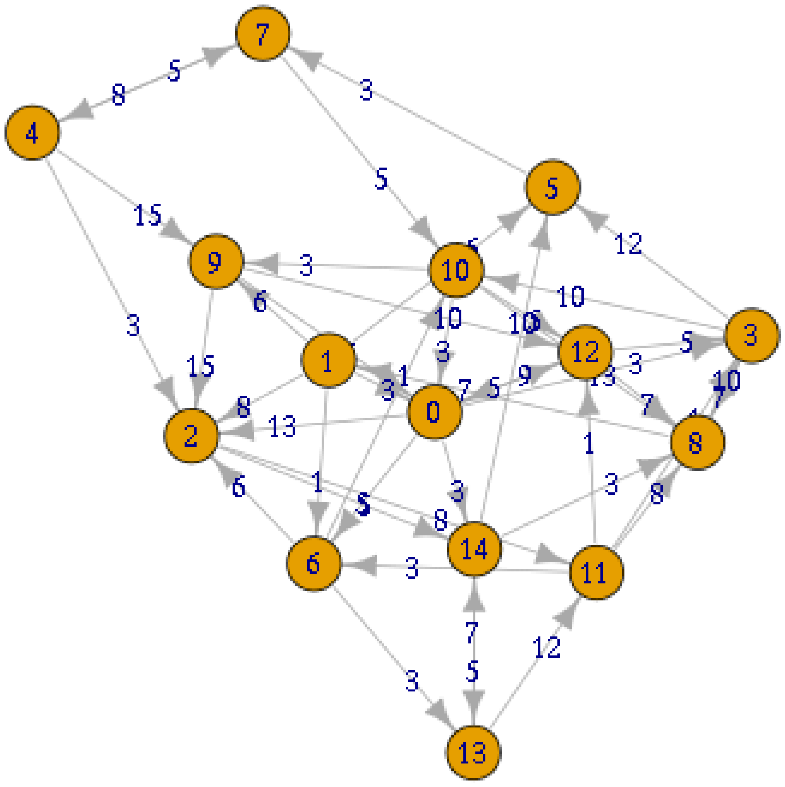

Figure 7: Effects of cache size on performance
file: resources/big_cluster.png