CSLP Proposal

Traiko Dinev

Programming Language

The language of choice is Python, mostly because input formating is easier, but also since it is widely used in scientific projects. The latter means we already have statistics and data-processing libraries that have been tested in a more rigorous environment. Namely - scipy - data processing & statistics - numpy - data manipulation, but also some rudimentary probabilistic functions - matplotlib - for plotting data

In addition, python is faster than some more popular choices (java). It is slower than C, though C adds too much additional overhead than needed (memory management in particular).

Specifications

Overview

In the system, there are two objects that will be represented as such in the code: Lorries and Bins. Users will be handled for each bin and since they're not discrete, we don't need to keep track of them individually. Every bin will track the next time a bag will be disposed.

We also will have a depot/station, which will be a state machine

When Lorries are on a route, or are in the station, they will be marked unavailable. The station/depot itself will be a state machine - it will be responsible for issuing scheduling events.

To glue all of this together, an observer pattern will be used. There will be an **EventDispatcher** class, which will automatically dispatch events when they happen. The different components will be observers to those events and can schedule new ones if needed.

To implement different algorithms for scheduling, there will be a separate class to handle this, which will be injected in the constructor of the Depot on application launch.

Statistics will be collected by having the statistics class connected to events happening in the **EventDispatcher**. As such, it will have no knowledge of the system, but will only *listen* and record events.

To display graphics, a **GraphicsManager** will be used. It will have the **Statistics** class injected to it and on demand will produce its output.

The same will be true for the output of events at the end. They will be aggregated by an event-listening class (**OutputManager**) that will be attached as an observer to the **EventDispatcher**.

Both will be injected statically at the start of the application.

Note this approach allows us to easily disable/enable output, per specifications. We can also display events on the go with **OutputManager** or tell it to display it all at the end of the simulation, as needed.

Class overview

Depot

The depot will be a state machine. It will know whether a dispatch is needed, how many bins need servicing and which ones they are. It will also know which lorries are available and will issue dispatch events on the lorries.

As such, it is a management class with knowledge of the entire system.

It will attach itself and listen to all events from the EventDispatcher (like the Statistics class). Then it can issue lorries with routes when necessary.

Note lorries themselves will also be implicitly attached to the event listener.

Depot
dispatchNeeded
overflowedBins
availableLorries

Bin

Bin
static const volume
static const serviceTime
nextDisposalEvent

Lorry

Lorries will have their route pre-calculated and stored as they go, so they can respond to events, but will stop and automatically go back to the depot once full. That will be accomplished by a simple Dijkstra, which will be pre-calculated from every point of the map, unless the map is too big to do so (again, a runtime decision to be made). This is akin to many multiplayer games where the map is pre-processed to speed up the simulation. For large maps it could be possible to calculate the best path to the depot from some points only. For smaller maps it could be possible to pre-process the entire map to know all the best routes from all points to all others. Again, this is just an optimization technique that might not even be needed in the general case. Note lorries will not know about the Depot class.

Lorry
static const capacity
location
currentCapacity
route

EventDispatcher

Observer
function onEvent(Event* event)

EventDispatcher
function addGlobalEventListener (Observer* listener)
function addEvent (Event* event)
function executeNextEvent ()
function

Statistics

Statistics(Observer)
function String getStatisticsSummary()

Graphics

Graphics
constructor (Statistics* statistics)
function displayGraphics()

OutputManager

OutputManager (Observer)
bool is_realtime
function getOutputSummary()

Route algorithm

This task can be simplified to the travelling salesman algorithm. As such, the best (in time) possible algorithm we can use will utilize some heuristic. Different options can be explored, including brute-forcing the paths.

The first heuristic to be used is going to be the MST one (Rose, 1977).

In addition, a local-search heuristic can be used. All of these will be evaluated on the current domain and the best performing one will be picked as the default heuristic.

Testing

All the different modules will be tested independently by using `unittest`, the standard python testing module. If time permits, functional tests can also be made. In testing classes that have dependencies on other modules, those will be shimmed (stub classes will be created for the purposes of the test).