CS 452 Train Control 1

Louis A. Burke (laburke) and Taras Kolomatski (tkolomat)
July 28, 2017

Overview

Structure

Our code is, at the time of this submission, in a fractured state. There are four main branches which have active development work on them.

The first branch is `dev-ai` which contains the logic for neural networks as well as fuzzy logic. These algorithms are fully functional and sufficiently performant, however they do prove to be ill suited for the task at hand.

The second branch is `dev-routing` in particular a local branch called `isolation` which contains the various servers and tasks which maintain the state of the track and choreograph train movement. One of the major servers is the position server, which maintains an accurate position for each train at any given moment. In reality it only maintains histories of sensor readings, switch positions, and train speeds, and dynamically calculates position when called for. Another important set of servers are the notification servers, these servers allow tasks to be dynamically called for various events. For example tasks can register with the senser notification server to be sent a message whenever a sensor is tripped. Similar servers exist for delays and switches.

The third branch is `taras-track-extras`. This is an active development task for calibration of trains with a simple text UI attached to it. This was the branch used for the demo as all current calibration code exists here as well as the specific code for determining stopping distance.

The fourth branch is `dev-ui` which contains a graphical user interface written in Ada as well as a set of tasks for communicating with it. This works by communicating directly over the serial port just like any other terminal, special terminal codes correspond to updates in the UI.

Algorithms

We used a number of powerful algorithms for this milestone. For routing we used dijkstra's search algorithm. Given the nature of the track, an A* search could be possible, but given that dijkstra gave very good performance we deemed it unnecessary to upgrade.

In addition to dijkstra's algorithm for routing, we have arbitrary fuzzy logic systems and multilayer perceptrons for data interpolation. These were found to be largely unnecessary. However, the neural network was mostly ported from a previous project so did not take too much time and the fuzzy logic may yet find some use.

Finally we have a simple linear interpolation producing our stopping times. This was created by having sagemath produce a linear regression on data gathered from sensors. In order to improve the accuracy of the stopping time, a long-running dynamic algorithm routes the train over a loop multiple times to precisely calibrate a stop.

Stopping

Results

While we have extremely accurate stops directly on sensors that are easily extensible to offsets, this comes at a cost. In particular our stopping algorithm is an adaptation of an algorithm designed to train a neural network, and so it only works on ideal data. This means that it cannot work on paths that do not contain a loop. Additionally it gets very poor results unless it is allowed to take a full calibration loop before attempting to stop.

Meanwhile since the branch with the stopping code lacks a proper routing task, it is unable to deal with self-intersecting paths from dijkstras algorithm. Additionally it refuses to reverse a train, so at the time of presentation there were many sections of track that could not be used as stopping points.

On other branches we have a working GUI, and working notification servers, as well as partially functional position and routing servers.

Going Forward

From this milestone we have learnt that accurately modeling the trains is both easier and harder than previously anticipated. In one respect, the relationship between the physical world and the computer turned out to be remarkably straightforward. Going into this project we thought that the physical properties of the track were sufficiently disparate so as to require either many hours of data collection or else artificial intelligence (with dynamic data collection) to accurately model. Having now collected the relevant data we have found that a simple linear interpolation suffices for nearly all aspects of the system, while linear regression can accurately model everything about the tracks without need for large sample sizes.

In the coming days we intend to assimilate all of our disjoint systems and combine them with more notification and track maintenance utilities to create an accurate model.

The benefit of our current disjointed structure is that most of our utilities were designed in isolation with multiple trains in mind. As such once we successfully bind them together they should, at least in theory, work out of the box with multiple trains. Of course we expect many edge cases to present themselves, but we are hopeful that our systems will work as designed.

SHA of commit

The commit hash of the submission commit (which is on taras-track-extras) is:

`4b2fbbf521cbd1b327bde31903ffe5b3aae4ebee`

The repository can be cloned from:

`gitlab@git.uwaterloo.ca:laburke/cs452_kernel.git` (SSH)

or

`https://git.uwaterloo.ca/laburke/cs452_kernel.git` (HTTPS)

There is a README in the root directory of the repository which outlines the loading command. To run the program once it is compiled just restart the machine and run `load -h 10.15.167.5 "ARM/path/to/kernel.elf"` to load it. Note that the loaded address is different from the default value.