CS 452 Final Project

Felix Fung (f2fung)
Dusan Zelembaba (dzelemba)

July 31, 2013

1 How To Run

```
> load -b 0x00218000 -h 10.15.167.4 "ARM/f2fung/p2_final.elf
> go
```

2 Submitted Files

Files listed here can be found under `/u1/f2fung/cs452/p2/`

2.1 md5sums

3 Project Description

3.1 Overview

The goal of this project was to line up some number of trains in a row and then give them a command to sort themselves in any order we choose. As is, the problem is pretty simple since we can just put all the trains in the bays and then tell them to line up accordingly. To make it more challenging, we set out to use as little of the track as possible. Our final solution only required the inner loop plus one extra "semi-loop" on top or below the inner loop.

By the end of TC2 we could already sort 2 trains, so our goal was to be able to sort 3 trains. Our stretch goal was to be able to sort 4 trains. We are confidently able to sort 3 trains, and sorting 4 trains seems to have a 50-50 chance of passing various reasons including stuck trains, hardware failures and potential bugs.

3.2 Train Sort Algorithm

3.3 Modifications

3.3.1 Train Controller

We found that switching a switches as we were going in to a merge was very helpful for dealing with cases like trains reversing while on a merge node. It simlified the code and solved the ugly case of the front wheel passing the switch and the back-wheel not passing the switch.

The sorting user task required two new pieces of functionality from the train controller:

1. Being notified when all trains stopped moving, and
2. Disabling edges for path finding.

Both of these were implemented as simple messages the sorting task could send to the train controller.

3.3.2 Deadlock Detection/Resolution

Deadlock detection/resolution was disabled for the final project. Our sorting task is smart enough to never get the trains in a deadlock, so deadlock detection/resolution just complicated things.

3.3.3 Path Following

In order to pack the trains together more tightly, the distance a train reserves behind it was reduced to 5cm. This is less than the max error we assume from the location server, but it allowed us to pack trains sensor to sensor and didn't cause major issues for our project. To pack trains even more tightly, we reduced our reservation lookahead error buffer from 15cm to 5cm (so now we lookahead stopping distance + 5cm to reserve edges).

3.3.4 Bug Fixes

The bulk of our time spent on the final project was fixing specific bugs realting to mutiple trains following the same path. This included things like:

- Trains not freeing previous edges when they became blocked on an edge
- Trains going into a DONE state when they became blocked on an edge
- Trains being stuck because they were blocked on an edge ahead of them, but their path uses an immediate reverse.
- Trains on a merge node thinking they could use the other edge of the branch in their path without reversing.
- And so on...

3.4 Missed Details from TC2

We implemented some things in TC2 that we never mentioned in our docs, so we'll talk about them here.

3.4.1 Sensor Attribution

We used a simple algorithm that assigns a sensor to the train that is "closest". By "closest", we mean the train who was travelling towards the sensor and was the least amount of edges away from it. If no train, was less than 4 edges from the sensor the sensor got ignored.

Once a sensor was attributed to the train, the train checks if it is expecting this sensor or not. The location server tracks the next sensor a train expects to see and if the attributed sensor is equal to the expected sensor, the train's location is updated. This is also where the location server checks for missed sensors. If we implemented handling of broken switches, this logic would go here as well.

3.4.2 Error Handling

To handle broken sensors, we use a form of a timeout. Instead of a normal timeout, we just use the value from the distance server since its based on time anyways. If the expected distance is ¿ 20cm past a sensor then we assume the sensor is broken and update our location. If this happens twice in a row, we assume that we lost the train and just wait for a sensor to be triggered the gets attributed to the train. To assure that we run smoothly when we assume a sensor is broken that isn't (because of large error), we store the broken sensor and if it gets triggered we reset out position to that sensor.

We didn't implement handling of broken switches because it didn't seem very useful. We can handle broken switches by removing edges from the graph and the chances of a switch breaking during a single run are low enough that it didn't seem like a useful feature for our project.