Task 7 Train Shunting

Adrian Jonsson Sjödin

Spring Term 2023

Introduction

The task is to write a function that calculates a sequence of moves to rearrange a train of wagons on two tracks. The train consists of a list of wagons, and the tracks are also represented as lists. The goal is to move the wagons from the main track to the two other tracks, and then back to the main track, in a specific order. The function should take the initial state of the train and return a list of moves.

In this report, we present a solution to this task based on the information provided in the task description. Our method uses the Train module to manipulate the train, the Moves module to represent individual moves, and the Shunt module to calculate the sequence of moves.

Method

The `Train` module provides functions for manipulating a train represented as a list of wagons. The `Moves` module contains functions that represent moves on a train, and the `Shunt` module provides the `find/2` and `few/2` functions that takes two lists as arguments and returns a list of moves that will move the wagons in the first list to match the order of the wagons in the second list. The difference between the two being that `few/2` optimizes the list of moves that are returned so that all zero moves are discarded and adjacent moves to the same track are combined.

Both the `Train` module and the `Moves` module where quite simple to develope, since the task gave quite clear instruction on what exactly each function in them should do, as well as examples of the expected output. The tricky part was the `Shunt` module.

The `find/2` function in the `Shunt` module takes two lists as arguments, `xs` and `ys`, representing the current configuration of the train on the main track (`xs`), and the state we want to reach (`ys`). For each element `y` in ys, we split xs using that element as the pivot, and calculate the number of wagons to move from the main track to each of the two other tracks. We

then append these moves to the result and recursively call find/2 with the updated train and the remaining pivot elements. See code overview 1

Code Overview 1: The `find/2` function

```
def find([], []) do [] end
def find(xs, [y | ys]) do
    {hs, ts} = Train.split(xs, y)
    n_hs = length(hs)
    n_ts = length(ts)
    [
      {:one, n_ts + 1},
      {:two, n_hs},
      {:one, -(n_ts + 1)},
      {:two, -n_hs} | find(Train.append(hs, ts), ys)
    ]
  end
```

The `few/2` function works in a similar way to the `find/2` function, by first splitting `xs` into two smaller lists , `hs` and `ts`, calculate the length of these and then generate the moves. However we don't immediately use recursion on the tail of the moves list and return it like in the `find/2` function. Instead we first check if `hs` is empty and if it is we recursively cal the function on `ts`, if it is not we instead concatenate the moves with the result of calling few/2 on ts ++ hs and ys. See code overview 2.

Code Overview 2: The part of `few/2` that is different from `find/2`

```
    moves = [
      {:one, n_ts + 1},
      {:two, n_hs},
      {:one, -(n_ts + 1)},
      {:two, -n_hs}
    ]
    if n_hs == 0 do
      [] ++ few(ts, ys)
    else
      moves ++ few(ts ++ hs, ys)
    end
  end
```

Since this report needs to be between 2-3 pages I've omitted the `compress/1` function here, however all code can be found on my GitHub.

Result

Code overview 3 and 4 shows the output from reversing a three wagons long train using `find/2` and `few/2`. Code overview 5 gives us the same result as in 4, meaning `compress/1` works as intended. Finally in code overview 6 we see that the found list of moves does work and reverses the order of the train.

Code Overview 3: Result from `find/2`

```
iex> Shunt.find([:a,:b,:c],[:c,:b,:a])
[ one: 1, two: 2, one: -1, two: -2, one: 1, two: 1,
  one: -1, two: -1, one: 1, two: 0, one: -1, two: 0]
```

Code Overview 4: Result from `few/2`

```
iex> Shunt.few([:a,:b,:c],[:c,:b,:a])
[one: 1, two: 2, one: -1, two: -2, one: 1, two: 1, one: -1, two: -1]
```

Code Overview 5: Result from `compress/2` on `find/2`

```
iex> Shunt.compress(Shunt.find([:a,:b,:c],[:c,:b,:a]))
[one: 1, two: 2, one: -1, two: -2, one: 1, two: 1, one: -1, two: -1]
```

Code Overview 6: Result from using `sequence/2` on the list of moves returned by `few/2` and `compress/1`

```
iex> Moves.sequence([{:one,1},{:two,2},{:one, -1},{:two,-2},{:one,1},
                     {:two,1}, {:one,-1},{:two,-1}],{[:a,:b,:c],[],[]})
[
    {[:a, :b, :c], [], []},
    {[:a, :b], [:c], []},
    {[], [:c], [:a, :b]},
    {[:c], [], [:a, :b]},
    {[:c, :a, :b], [], []},
    {[:c, :a], [:b], []},
    {[:c], [:b], [:a]},
    {[:c, :b], [], [:a]},
    {[:c, :b, :a], [], []}
  ]
```

Discussion

The trickiest part of this task was as already stated the `Shunt` module, and then in particular the second part which was the `few/2` function. At first I

could only get it to return a list that on index 0 contained another list with the correct simplified output, but then on index 1 and up we still had the values that we wanted removed. The problem was with how I concatenated and called `few/2` recursively.

The `compress/1` function that I implemented became essentially redundant since `few/2` already did the same simplification when calculating the moves needed. I did however test it on `find/2` to make sure I got the same result. Based on this it might be that I implemented `few/2` wrongly and that it was only supposed to remove zero moves.