Train Shunting

Aditya Gupta

February 2023

Introduction

In this report, we solve the train shunting problem. We begin with a naive approach and improve it step by step.

Train Processing

We write a few functions in the *Train* module which will be later used when we work with trains. The following functions are part of the Train module:

- *Member* → This function is used to check if a wagon is part of a train or not. We keep comparing the head of the list with the element we are checking for and if we find a match then we return true otherwise we recursively keep going through the list. In the case, a match isn't found then we just return false.

```
def member([], y) do
  false
end
def member([head|rest], y) do
  if(head == y) do
    true
  else
    member(rest,y)
  end
end
```

- *Position* → This function is used to find the position of a wagon with the first wagon in a train numbered as position 1. Here I again use a tail recursive approach where I use the accumulator to find the position of the argument passed to the function. We keep going until we find the element for which we want to find the position. If we locate it then we just return the accumulator. Otherwise, we recurse further with the accumulator value incremented by one.

```
def position(train, y)do
  position(train, y, 1)
end
def position([], _, acc) do acc end
def position([head|rest], y, acc) do
  if(head == y) do
    acc
  else
    position(rest, y, acc + 1)
  end
end
```

- *Split* → This function is used to return the tuple of all wagons before and after the wagon given to us as an argument. We keep checking if the head of the list is the element on which we want to split. If it is then we just return the elements which we accumulate and the rest of the list. Otherwise, we keep appending the current wagon to the end of the accumulator list and call the function recursively for the rest of the list with the updated accumulator value.

```
def split(train, y) do
  split(train, y, [])
end
def split([], _, _) do :error end
def split([head|rest], y, acc)do
  if head == y do
    {acc, rest}
  else
    acc = append(acc,[head])
    split(rest, y, acc)
  end
end
```

- *Main* → I had a lot of difficulties while writing this one. At first, I didn't realize that we needed to write the solution as just one function, so I attempted to solve it using other helper functions and functions we wrote before for the *Train* module. But after failing to properly implement the function after many attempts spanning a few hours, I looked at the solution on the course GitHub repository. But the approach wasn't very intuitive at first glance and only made sense upon careful evaluation and drawing it out with pen and paper. It's a very clever way of using recursion and I'm pretty sure I couldn't come up with that mrestelf. I have omitted how I implemented the append(), take() and drop() functions as I felt their implementations were trivial.

Applying Moves

Now we implement 2 more functions: single/2 and sequence/2. The single() function takes the current state of the train and a *move* argument and returns the final state of the train. We are given moves of the form:

```
{Track, Number}
# 1. Track can be :one or :two
# 2. "Number" is the number of wagons from the beginning of the train
#    on track main which we need to move to the specified track.
#    In case the "Number" = 0 then we don't move anything.
# 3. The remaining wagons after a move remain at their original track
# 4. if the number is negative we shift from the specified track to main
```

The single() function

This function makes heavy utilization of pattern matching. Here we start to make use of take/2, drop/2, append/2 and main/2 functions from the *Train* module which we made before. If the move that we want to do is {:one, n} then we check if n is positive or negative. If n is negative then we take n wagons from track one and then we append these wagons to the end of the main track's train. We use drop/2 to remove these wagons from track one. And track two remains unaltered. If n is positive then we make use of the main/2 function in order to take n wagons from main. Then we keep the remaining wagons on main and append the taken wagons to the end of the train on track one. Track two still remains unaltered. We follow exactly the same approach if the move is {:two, n} instead.

```
def single({_, 0}, state) do state end
def single(move, {main, one, two}) do
    case move do
      {:one, n} ->
        cond do
          n < 0 ->
            mvd_wgs = Train.take(one, -n)
            single({:one, n + 1}, {Train.append(main, mvd_wgs), Train.drop(one, -n), two})
          true ->
            {0, drop, take} = Train.main(main, n)
            {drop, Train.append(one, take), two}
        end
      {:two, n} ->
        cond do
          n < 0 ->
            mvd_wgs = Train.take(two, -n)
            single({:two, n + 1},
            {Train.append(main, mvd_wgs), one, Train.drop(two, -n)})
          true ->
```

```
            {0, drop, take} = Train.main(main, n)
            {drop, one, Train.append(two, take)}
        end
    end
end
```

The sequence() function

This function takes the list of moves that we want to apply to a state and returns all the states that we go through as we apply each of the moves provided to us. When there are no more moves left in the move list then we just return the current state. Otherwise, we recursively construct the move list. We make use of the | operator in order to add the current state with the recursive call to sequence/2 with the rest of the moves left and the result we get upon applying the move using the single/2 function.

```
def sequence([], state) do [state] end
def sequence([move|rest], state) do
    [state | sequence(rest, single(move, state))]
end
```

Shunting

First, we examine the naive approach and the problem associated with it. We implement this by making use of the find/2 function. Here the main problem with the approach is that we do unnecessary moves even when we have reached the desired state as we follow a specific pattern in order to find the moves for the shunting problem.

```
def find([], []) do [] end
def find(train, [y | rest]) do
    {hs, ts} = Train.split(train, y)
    {hl, tl} = {length(hs), length(ts)}
    [{:one, tl + 1}, {:two, hl}, {:one, -(tl + 1)}, {:two, -hl}] ++
    find(Train.append(ts, hs), rest)
end
```

To solve this issue we can make a simple optimization to the find/2 function's approach. Now we keep track of y's position. If it's at the first position then we just skip over to the recursive call. Otherwise, we just proceed as we did before. Even with this simple optimization, we end up with significantly fewer moves. We store this approach in the few/2 function.

```
def few([], []) do [] end
def few(train, [y | rest]) do
    {hs, ts} = Train.split(train, y)
```

```
    {hl, tl} = {length(hs), length(ts)}
    case Train.position(train, y) do
      1 ->
      few(Train.append(hs, ts), rest)
      _ ->
      [{:one, tl + 1}, {:two, hl}, {:one, -(tl + 1)}, {:two, -hl}] ++
      few(Train.append(ts, hs), rest)
    end
end
```

We are asked to compress the moves further and are provided with the compress/1 function. If the head has n = 0 then we call rules on the rest of the moves(or the tail of the list containing the moves) to omit all moves with n = 0. Otherwise, we check the elements within the tail of the move list. If the tail is empty we just return the head. Otherwise, can split the tail further and check the first two moves of the move list. If both the moves concern the same track then we just add their n values and call rules recursively on the rest of the move list(t). Otherwise, we keep the head of the list(this means it has been reduced as much as it could have been) and append it to the result of recursively calling rules on tail.

```
def rules([]) do [] end
def rules([head | tail]) do
    case head do
      {_, 0} ->
        rules(tail)
      head ->
        case tail do
          [] ->
            [head]
          [h | t] ->
            case {head, h} do
              {{track, n}, {track, m}} ->
                [{track, n + m}] ++ rules(t)
              _ ->
              [head] ++ rules(tail)
            end
        end
    end
end
```