Trainshunting

Sam Khosravi

Spring Term 2022

Introduction

In this task we are in charge of the shunting wagons of a train. The shunting stations are used to rearrange any given train to a desired train. When rearranging we will also give the the sequence of moves used. We will have a ”main” track and two shunting tracks called ”one” and ”two”. The moves will descrive how one wagon of a train moves from one track to another track.

Some list processing

Here we define a few functions that will help us further ahead. Two major functions used are take and drop. The function take/2 returns the list containing the first n elements of a list.

```
def take(_,0) do [] end
def take([h|t], n) do
    [h|take(t,n-1)]
end
```

Drop works more or less the same way with the recurssion, but this one will however return the list xs without its first n elements.

```
def drop(xs, 0) do xs end
def drop([h|t], n) do
  drop(t, n-1)
end
```

Other than this there are functions such as append/2 which takes in two lists and returns these two lists appended to eachother, member/2 which will check if a element is a part of a list and then position/2 which returns the first position of a given atom in the list we send in.

Applying a Single Move

The function single/2 takes a move, which is represnted by a tuple, and an input state which is a tuple of three lists that define our ”main”, ”one”, and ”two” tracks. Whenever a move is performed on a state we will perform single, which makes it a crucial part of the code. The single/2 code was made both for when we move from ”main” to ”one” and for ”main” to ”two”. This was the only things needed because if we want to go from ”one” to ”two” we would have to pass ”main” anyways. If we have a move with the atom

:one and a positive integer, which states how many wagons are supposed to move, we will use our take/2 function and drop/2 function to aid us. The move :one, 1 will for example take one wagon from the ”main” and put it in the front spot of ”one”. To remove the wagon from main, we will use take/2 to return the first n elements of our main list. By taking the length of our main function subtracted by the number of wagons we want to move, we will get back a list with that length. In this example, the main list will have all its elements except the last one, which will be moved to ”one”. To do this we will just take the last element in our main by using our drop/2 function, to then append it with the ++ method to our list one. If we however want to move a wagon from ”one” to ”main”, we would do the same thing but opposite.

```
 def single({:one, n},{main, one, two}) do
    if n > 0 do
      {take(main, length(main)-n), drop(main, length(main)-n) ++one, two}

  else
    {main ++ take(one,n * -1), drop(one, -1 *n), two}
  end
  end
```

Applying Several Moves

The function move/2 takes an input state and a list of moves. It results in a list of states by applying each move on the state. Each new state is returned. This was a simple implementation. We take each move and apply the single/2 function on it and the state, we then append the state to a list and revursively call on move with the rest of the moves and the updated state.

```
def move([],  {main, one ,two}) do
  {main, one ,two}
end
def move([moveshead|movestail],{main, one, two}) do
```

```
 [{main, one, two}, move(movestail, single(moveshead, {main, one, two}))]

end
```

Finding Moves

The function find/2 takes two trains as input and returns a list of moves, such that the moves transform the one state into the other. This was by far the hardest part of this task, not because of the it being complex, but because I had such a hard time understanding the instructions. The split function was easy to implement, but the find function felt almost impossible to me as however much I tried, I could not really understand the instructions. However, I got it explained to me by my classmates. The split/2 function takes a list of wagons xs and the wagon y and returns the pair hs,ts, where hs is the list to the left of the wagon y and hs is the list of wagons to the right of y. The way this was implemented was that I would use the take/2 function to return the elements infornt of y by taking its position as the number of elements who are supposed to be returned. This is done in a similar fashion for the elements to the right of y, by using drop/2.

```
  def split(xs, y) do
    {Trainshunting.take(xs, Trainshunting.position(xs,y)-1),Trainshunting.drop(xs,
    Trainshunting.position(xs, y))}
  end
```

The find function was also pretty hard to implement for me. I mostly just followed the instructions and implemented everything step by step without really understanding a lot of what I did until the end. The split function would give us hs and ts. Then I would put ts and y into the same list. The tail would also be put to ts and hs appended. Then I would just return the moves which are more or less hardcoded because of the instructions, and then recursively call on find again with the tail of our desired list.

```
 def find({[],[],[]}, {[],[],[]}) do
    []
  end

  def find({xs,[],[]}, {[yshead|ystail],[],[]}) do
    {hs,ts} = split(xs, yshead)
    ts = [yshead|ts]
    [_|remainingtail] = Trainshunting.append(ts, hs)
    [{:one, length(ts)},{:two, length(hs)}, {:one, -length(ts)},
    {:two, -length(hs)}|find({remainingtail,[],[]},
    {ystail,[],[]})]
  end
```

Finding Less Moves

The few/2 function is implemented here, and it works just the way as find does, but if we find that a wagon is in its correct place we will do no moves and continue on. The only difference from find is that here we will have a condition. If the two heads are equal we will just call on few again as if we are done with a change. If they are not equal we will do what we normally did.

```
def few({[],[],[]}, {[],[],[]}) do
  []
end

def few({[h|xstail],[],[]}, {[yshead|ystail],[],[]}) do
  {hs,ts} = split([h|xstail], yshead)
  ts = [yshead|ts]
  [_|remainingtail] = Trainshunting.append(ts, hs)
  cond do
    h == yshead -> few({xstail,[],[]}, {ystail,[],[]})
    h != yshead -> [{:one, length(ts)},{:two, length(hs)},
    {:one, -length(ts)}, {:two,
    -length(hs)}|few({remainingtail,[],[]}, {ystail,[],[]})]
  end
end
```

Move Compression

Here we basically just optimize the moves done by few. The compress code was given in basically pseudo code, and I used a case clause instead of if like the code given. The rules/1 function will take in the list of moves, and apply the rules to them. So if a move is being applied to the same track we will add the number of wagons moved together, then send it back recursively with this new move implemented. If the number of moves if 0, we will just remove this move as it does nothing. Then we have the list return at the end which is our basecase basically.

```
def compress(ms) do
  ns = rules(ms)
  case ns do
     ms -> ms
    _ -> compress(ns)
end
  end
```

```
 def rules([{h1, n}, {h2, m}|tail]) do
  cond do
      h1 == h2 -> rules([{h1, n + m}|tail])
      n == 0  -> rules([{h2, m}|tail])
      true -> [{h1, n} | rules([{h2, m}|tail])]
  end

end
```

Conclusion

This was a pretty fun task to do as soon as I understod what moves, tracks, trains, etc was. However, I feel like this task took a bit too long just because of the instructions being a bit weird during the later parts of the lab. I always enjoy doing the tasks where we actually solve a ”real life” problem.