Train Shunting

Björn Formgren

Spring 2023

Introduction

The seventh assignment in the course ID1019 consisted of solving a train shunting problem where there was one main track and two additional tracks. Given two sequences of trains, the given train and the desired train, the ambition was to re-arrange the given train using the tracks until it looked like the desired train. The goal was to find a short sequence of moves that solved the above problem.

Modeling

For starters there had to be model of the problem. The trains consisted of a list of wagons that were modeled as atoms. The entire state of the shunting stations were described using the three tracks. $[:a,b],[:c,:d],[:e,f]$. This state represents the main track to the left and track one and two respectively from left to right. The first wagon in the trains were always the left-most wagon.

A move was modeled as a binary tuple, the first element being which of the two tracks and the second an integer representing how many wagons that should be moved to or from that track from or to the main track. If the integer was positive, the wagons were moved to the track included in the tuple. If the integer was negative the wagons were moved from the track in the tuple to the main track.

Train processing

Now that the model of the problem was complete it was time to create some train processing functions. Seven functions was created to process the trains, these were:

- take/2
- drop/2
- append/2

- member/2
- position/2
- split/2
- main/2

The main function was the trickiest function to implement. Since the wagons on the main track were in reverse order, a function was required that could divide the main track into two segments, the segment that should remain and the wagons that should be moved. The assignment asked for a function that did not use any of the functions already implemented. Below is the code for the main/2 function.

```
def main([], n) do {n, [], []}  end
def main([h|t], n) do
    case main(t, n) do
      {0, drop, take} ->
        {0, [h|drop], take}
      {n, drop, take} ->
        {n-1, drop, [h|take]}
    end
  end
```

The main function returns a tuple: `k, remaining, take` The remaining variable represents the wagons that are left on the main if not all were moved. The take variable represents the n wagons that should be moved from main to either one or two. Depending on the value of n one or the other is filled with wagons.

Applying moves

In the next section of the assignment it was time to apply the moves to a state. This was done using a function single/2 that takes a move and an input state and returns a new state from which the move had been applied to the original state.

```
def single({:one, n}, {main,one,two}) do
    if n > 0 do
      {_, remaining, take} = Train.main(main, n)
      {remaining, Train.append(take,one), two}
    else
      {Train.append(main,Train.take(one,abs(n))),
        Train.drop(one,abs(n)), two}
    end
  end
```

```
  def single({:two, n}, {main, one, two}) do
    if n > 0 do
      {_ , remaining, take} = Train.main(main,n)
      {remaining, one, Train.append(take,two)}
    else
      {Train.append(main,Train.take(two,abs(n))),
        one,Train.drop(two,abs(n))}
    end
```

When the integer n was positive, n wagons were moved from main to one of the two tracks. If it was negative, the absolute value of n wagons were moved to the main track from the specified track. Next the function sequence/2 was implemented, this function took a list of moves and a state. It returns a list of states that represents the transitions when all moves are performed.

Shunting problem

Now, for the actual problem at hand. Once again, the idea is to transform a given train into it's desired constellation. The instructions did present the algorithm in pseudo code. Below is the result of the find/2 function which takes the given train and the desired train as parameters and returns the list of moves needed to reconstruct the train into the desired state.

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

However, find/2 contains a lot of redundant moves. Therefore another function few/2 was implemented with the goal to reduce these redundant moves.

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

The few/2 function simply skip the iteration if that wagon is already at it's right place.

Move compression

The last part of the assignment had to do with even more compression of redundant moves. Four additional rules were added to minimize the moves even further.

```
def rules([]) do [] end
def rules([{:one,0}|tail]) do rules(tail) end
def rules([{:two,0}|tail]) do rules(tail) end
def rules([{:one,n},{:one,m}|tail]) do
    rules([{:one,n+m}|tail])
end
def rules([{:two,n},{:two,m}|tail]) do
    rules([{:two,n+m}|tail])
end
def rules([move|tail]) do [move|rules(tail)]
```

Lastly the function compress/1 was implemented to repeat the application of rules until the list of moves did not change.

```
 def compress(train) do
    new_train = rules(train)
    if new_train == train
      do train
    else
      rules(new_train)
    end
  end
```