Springs Part I

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open a window onto the implementation of the first part of the Springs problem as part of the Advent of Code 2023, which consists in reading the input file and processing the data accordingly, so that it can then be used in order to find the task's final algorithm solution.

General approach

The given input file consists of several rows, which first include a series of symbols that describe the status of the springs, followed by several numbers which are indicative of the damaged one. The final goal is to represent each row under the form of a tuple with two elements, namely two lists. In order to do so, one has to extract and process the data from each row. In other words, the following steps have been taken:

Step 1: Reading the file and extracting the rows

We start by defining a method that initializes the process, namely it reads the given file and at the same time it extracts each row at a time so that we can then process them individually by applying to it the created *splitline/1* method. In order to do this, the *Stream* and *Enum* modules have been used.

```
def readfile() do
  File.stream!("input.txt") #open the file
  |> Stream.map(&String.strip/1) #extract each line
  |> Stream.map(&splitline/1) #apply the given function to each line
  |> Enum.to_list() #return result
end
```

Note that the pipe operator has been used in order to pass the read file as the first argument to the rest of the functions and that in both cases, *Stream.map* applies the anonymous function to the given data stream. From my understanding so far, anonymous functions can be recognised thanks to

the capture operator, & and are necessary since this way one function can be passed as argument to other functions. As it is important to step by step check that everything works correctly and that data is processed the right way, in the end, the created data stream is displayed under the form of a list using *Enum.to_list* method.

Step 2: Further processing each row

Once the *splitline/1* function is applied to each row, an initial line such as "????.??.??. 1,1" will be turned into a list with two string elements as follows: ["????.??.??.", "1,1"]. Each element will then be further processed, whose results will in the end be part of the final desired tuple. This will be done by calling inside the tuple itself two created methods.

Step 3: Further processing the description of the springs

As previously mentioned, the final goal is in the end to turn the given rows into tuples but at the same time to convert the description of the springs into a list of atoms describing their status. In order to avoid using the stack, **tail recursion** has been used in the implementation of the *description/1 and description/2* methods which take as one of the arguments, the symbols under the form of a char list. The function with one argument is only used to simply call the two argument version of it, by sending the empty list as the second parameter as well.

```
def description(list) do description(list, []) end
def description([], acc) do acc end
def description(list, acc) do
  case hd(list) do
    63 -> description(tl(list), acc ++ [:ukn])
    46 -> description(tl(list), acc ++ [:op])
    35 -> description(tl(list), acc ++ [:dam])
  end
end
```

In the *case* statement, the head of the list is analyzed and depending on the scenario it matches, the appropriate atom is appended to the newly created list. The corresponding character ASCII codes have also been used, while the base case takes place when the list becomes empty and therefore, the accumulator is returned. After applying this method, one such line, '????.??.??.' will be turned into the following list: **[:ukn, :ukn, :ukn, :ukn, :op, :ukn,:ukn, :op, :ukn, :ukn, :op]**, which will then, be part of the final tuple.

Step 4: Further processing the description of the damaged springs

It is now time to turn the second tuple element into a list as well, by calling the *damaged_springs/1 and damaged_springs/2* methods. Similarly, tail recursion has been used and the one argument method simply calls the two argument version with the empty list as parameter.

```
def damaged_springs(list) do damaged_springs(list, []) end
def damaged_springs([], acc) do acc end
def damaged_springs(list, acc) do
 {n, _} = Integer.parse(hd(list))
 damaged_springs(tl(list), acc ++ [n])
end
```

The method takes as parameter a list of all the numbers splitted under the form of a string. Once again, similarly, the head is then, parsed using *Integer.parse*, which in turn returns a tuple (e.g: Integer.parse("1") returns {1, ""}. We therefore, use pattern matching to extract the integer only and ultimately append it to the newly created list. The base case takes place when the list becomes empty and therefore, the accumulator is returned, which will be part of the final tuple.

Step 5: Encoding each row

After having put into practice all the above mentioned steps, each line will take the form of a tuple. For example, **"????.??.??. 1,1"** will turn into **{[:ukn, :ukn, :ukn, :ukn, :op, :ukn, :ukn, :op, ukn, :ukn, :op], [1,1]}**.

Now that all data is fully processed, one is ready to actually find the correct solutions to the given problem.

Step 6: Checking the description

In order to be able to check if a found solution matches the description of the damaged springs, two methods have been implemented: *check/2* and *match/2*. The *check/2* method returns a list of the number of existent consecutive damaged springs, so that it can then be compared to the already given list in the *match/2* function, which will either return *true* or *false*.

```
def match(list, d) do
  case (list == d) do
    true -> 1
    false -> 0
  end
end
```

The check/2 method was the most complicated one to create since I encountered a bug that took some time to figure out. The bug took place in the situation when the last element of the sequence was the dash itself (e.g: ".#.##"). Because of that, I was losing a solution since the pattern match for appending the accumulator was not met (it was matching the :dam one). I, therefore, came up with a solution: introduce a new condition to append, namely when ***the remaining list is empty and the atom is a damaged spring***.

```
def check(s) do check(s, 0) end
def check([], acc) do [] end
def check(s, acc) do
  case(hd(s)) do
    :dam ->  case (tl(s)) do #the introduced condition
               [] -> [acc + 1] ++ check(tl(s), acc)
                _ -> check(tl(s), acc + 1)
             end
    _ -> case acc do
           0 -> check(tl(s))
           _ -> [acc] ++ check(tl(s))
    end
  end
end
```

Step 7: Exploring the possible solutions

The key in exploring all valid possibilities is of course using recursion but also, knowing that when encountering an unknown state, we have to replace it first with an operational spring, then with a damaged one in order to evaluate both scenarios. In addition, when replacing the unknown state, we will add the number of possible solution from both cases so that we can in the end, deliver the final result, namely the total number of solutions for the given row. Simultaneously, we will use pattern matching so that when encountering an operational or damaged spring, we will simply add the atom to the accumulator and continue the exploration. The base case occurs when the initial list is empty and we, therefore add the solution if it is a correct one.

Conclusion

Now that we have managed to correctly solve the problem, it's time to improve the algorithm by taking into account larger data inputs and of course to optimize the execution time by using dynamic programming.