Hot Springs Part 1

William Frid

Spring 2024

Introduction

"Hot Springs" Advent of Code challenge on the 12 December 2023 consists of a two step problem where the first step lays the ground for the second and requires nothing but recursive programming (as we've used so far in the course **ID1019**). The goal is to write a problem that takes 1000 lines of inputs that each consists of two parts: a map with three states, and a pattern that must be followed in the form of numbers. What we need to do is to find the total amount of possible complete maps that satisfies this pattern.

As this document contains a rather small amount of code, it's recommended to read it alongside the code written found at https://github.com/williamfridh/ID1019_VT19_TCOMK_Programming_2. Follow the link and go to Assignments/springs.

The Problem

Let's start of by diving deeper into the problem. As mentioned i the introduction, the goal is to take an input of 1000 lines, which each contains a form of map (with the three possible states being "." [working/`:ok`], "#" [broken/`:bad`], or "?" [unknown/`:unk`]), and a series of numbers telling us how many `:bad` springs must be in a row before an `:ok` springs is found.

For example, assume we have the input "???.### 1,1,3". What this means is that we have 3 `:unk` springs, followed by one `:ok` spring and 3 `:bad` springs. The numbers tell us that the the first broken spring must be alone, meaning that it cannot be followed by a second broken spring. Same goes for the second number, and finally the third number that refers to the 3 springs we know are broken. A quick analysis of this specific case tells us that there's only one possible solution to this, ".. 1,1,3".

After calculating the amount of possible solutions, we continue to the next row of the into and keep this up until we've summed up all possible solutions for each of the 1000 entries.

Parsing

The first step of solving this problem is to parse the data. One way of doing this is to store the given data (example below) in txt- or csv-files for for effective testing and better structure. Then we write a function (`docToList/1`) that reads from the file and passes the contents to a function called `rowsToEntires/1` that sends each row of the content to `rowToEntry/1`. This functions purpose is to split up the given row (in string format) and send the symbols to `symbolsToList/1` and numbers to `numbersToList/1`, then finally returns a tuple of these two new lists.

The two functions `symbolsToList/1` and `numbersToList/1` are rather simple. `symbolsToList/1` simply converts the symbols "." to `:ok`, "#" to `:bad`, and "?" to `:unk` and returns this in a list. And `numbersToList/1` only converts the input of strings into a list holding integers (main logic seen in snippet below the example).

???.### 1,1,3 .??..??...?##. 1,1,3 ... [[:unk, :unk, :unk, :ok, :bad, :bad, :bad], [1, 1, 3], [:ok, :unk, :unk, :ok, :ok, :unk, :unk, :ok, :ok, :ok, :unk, :bad, :bad, :ok], [1, 1, 3],...]

```
def numbersToList(nums) do
    String.split(nums, ",")           # Split string into list.
    |> Enum.map(&String.to_integer()) # String to integer.
end
```

As we se above, the result from this should be a list of tuples, where each tuple holds two lists, one of the maps made out of atoms, and one integer list representing the `:bad` pattern.

Algorithm

Assume we have received a list containing all the entries. Next each entry needs to be sent to a function that evaluates each and sum up the results. This is rather simple, but what about finding each individual result to be summed up? To do this, multiple things must be taken into consideration. To easier describe (and for you as a reader to understand) we'll be using the first line of the example above, "???.### 1,1,3" which will be referred to as "input". *Note that we use this notations as is before conversion for readability.*

The input is first passed into a function named `eval/1`, which is a function that comes in many different versions with different pattern matching to handle different cases. This function send the given input alongside a boolean `false` to `eval/2`. The boolean given is called `forceBad` (will be covered later).

The second `eval/2` that'll catch these two arguments is the second seen in the snippet below. This is where the exploration begins as the first symbol on the "map" (???.###) is unknown (`:unk`). This is done through adding together the results of the two assumptions (each result is an integer representing amount of possible solutions). Starting of, we assume the `:unk` is actually `:bad` and decrease the first integer in the number list ("0,1,3") leading us to a match in the if-statement. Since the first number in the list is 0, the arguments are caught by a function that removes the 0 (first function in snippet), and calls upon `eval/2` again, this time with the argument "?.### 1,3".

```
def eval({[sh | st] = sym, [nh | nt] = num}, _)
when sh == :ok or sh == :unk and nh == 0 do
    num = removeZeroHead(num) # Simple helper function.
    eval({st, num}, false)
end

def eval({[:unk | st] = sym, [nh | nt] = num}, false) do
    num_dec = [nh - 1 | nt]
    if nh - 1 == 0 do
        eval({st, num_dec}, false) + eval({st, num}, false)
    else
        eval({st, num_dec}, true) + eval({st, num}, false)
    end
end
```

Once again, the lower function capture the arguments, the first number is decreased to 0 and the first case of the if-statement is true. Sending the arguments ".### 0,3" further. This is once again caught by the first function above, that send s the arguments "### 3" further. This time, the version of `eval/2` in the snippet below will be used. It'll pass on the argument "## 2" as well as `true`, which is the variable called `forceBad` (mentioned earlier). This will happen two more times and the same function will act all three times in total and finally send and empty argument which will be caught and be returned as 1.

```
  def eval({[sh | st] = sym, [nh | nt] = num}, _) do
      num_dec = [nh - 1 | nt]
      if nh - 1 == 0 do
        eval({st, num_dec}, false)
      else
        eval({st, num_dec}, true)
      end
  end
```

The logic behind this is to try setting every `:unk` to `:bad` and upon doing to, decrease the value. When the value is 0, we know that the next one must be an `:ok`. If it is `:unk`, we guess it's an `:ok`, but if it's a `:bad`, the math won't add up as we'll get a negative number.

Note that `forceBad` wasn't used in this example as it's meant to help in cases for for instance we have the input "??.## 2,2". In this case the first function would pass on the arguments "?.## 1,2" and `forceBad` set to `true`. This would mean that the next values **must** be a :bad, otherwise the guess is bad and thus the value 0 should be returned.

Below you can see each version of `eval/2` that returns a result.

```
def eval({[], []}, _) do 1 end
def eval({[], [0]}, _) do 1 end
def eval({[], _}, _) do 0 end
def eval({[:ok | _], _}, true) do 0 end
def eval({_, [-1 | nt]}, _) do 0 end
```

Discussion

Writing this type of algorithm can be done with different workflows. You can for instance figure out the logic beforehand, or write the code on the fly with a pre-made set of tests. In this case, the code was written on the fly resulting in not the most clean code. Take the `forceBad` as an example. It's rather redundant as the same result can be achieved with some more comparison. It does however simplify the understanding of what's going on in the code.

Conclusion

Solving code challenges like Hot Springs Part 1 is a great way to improve ones coding and problem solving skills. It shows on the importance of a proper development flow and help improve ones pattern handling skills. As this solution was made for short inputs, it's important to keep in mind that it's not suitable for longer inputs as the time it takes to explore each case increases.