Springs Part II

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open a window onto the implementation of the second part of the first assignment, namely by first and foremost extending the data inputs and by trying to find all possible valid solutions. In order to do so, dynamic programming will be introduced. Before exploring the possibility of a solution, the existence of a solution will first be checked in the memory. If it exists, computation time will be saved and the answer will be returned. Otherwise, we will do the computation and store the information in the memory.

Making the data inputs larger

In order to actually see the magic of dynamic programming, we have to of course deal with larger volumes of data, which in our case translates into longer springs description. Therefore, as indicated, each input row will be multiplied 'n' times, by also adding a '?' in between.

Having said that, the method *extend_s/3* will extend the springs description, while the method *extend_d/3* will extend the description of the damaged ones. Both methods will return a list as output.

```
def extend_s(list, 0, acc) do acc end
def extend_s(list, 1, acc) do extend_s(list, 0, acc ++ list) end
#do not add a '?' in the case above
def extend_s(list, n, acc) do
  extend_s(list, n - 1, acc ++ list ++ [:ukn])
end

def extend_d(list, 0, acc) do acc end
def extend_d(list, n, acc) do
  extend_d(list, n-1, acc++list)
end

def merge(h,t) do [h|t] end
```

The key difference between the two is that *extend_s/3* will have an additional base case so that we do not add an extra '?' for the final row extension. Both methods are tail recursive. Moreover, in order to check that the final results are okay, I called the *merge/2* method using the output of the above methods as arguments.

Implementing the memory

Since I was initially using the *Enum and Stream* modules to parse the input file, When first implementing the memory using the *Map* module, I came across the problem of actually transmitting the newly created memory as parameter. Since the functions I was using had a predefined number of arguments, I couldn't simply pass an extra argument. Therefore, I had to change the way I was loading the file as presented below. This way, the memory parameter was transmitted once I was calling the *readfile/1* method.

```
def begin() do
  mem = Map.new()
  readfile(mem,[])
end

def readfile(mem, list) do
  {:ok, input} = File.read("test.txt")
  row = String.split(input, "\n")
  Enum.map(row, fn element -> splitline(element, mem)
  end)
end
```

Deciding how to represent the values in the memory

The key in implementing the solution using dynamic programming is to first check in the memory if that computation has already been done. If yes, the result will be delivered, otherwise, it will be computed and will be stored in the updated memory. In order to do so, the *checkmemory/5* method has been created.

In order to be able to identify a given situation in the memory, I will store the computation using a key-value index, namely the numerical description of the damaged springs (d) and the list of atoms describing the situation(acc).

```
def checkmemory(list, d, acc, sum, mem) do
  case Map.get(mem, {d, acc}) do #accessing the memory
    :nil -> {answer, mem} = explore(list, d, acc, sum, mem)
            {answer, Map.put(mem,{d, acc}, answer)}
    answer -> {answer, mem}
  end
end
```

Of course, the *explore/5* method has also been updated so that in each recursive step, the memory is first checked before starting any computation. It should be taken into account that the function returns a tuple as result, which includes the updated memory.

```
def explore(s,d,acc,mem) do
  {ans, mem} = explore(s,d,acc,0,mem) end
def explore([], d , acc, sum,mem) do
  {sum + match(check(acc), d), mem} end
def explore([:op | t], d, acc, sum,mem) do
  {ans, mem} = checkmemory(t, d, acc ++ [:op], sum,mem) end
def explore([:dam| t], d, acc, sum,mem) do
  {ans, mem} = checkmemory(t, d, acc ++ [:dam], sum,mem) end

def explore([:ukn | t], d, acc, sum,mem) do
  {ans1, mem} = checkmemory(t, d, acc ++ [:op], sum,mem)
  {ans2, mem} = checkmemory(t, d, acc ++ [:dam], sum,mem)
  {ans1 + ans2, mem}
end
```

Current problems and limitations

Unfortunately, my implementation is not as efficient as I was hoping it to be and that is because although the results are stored correctly in the memory using the mentioned key-value index (I see the memory as output in the terminal when I run the code as well), I do not have enough hits so that it can actually make a difference in the execution time. Having said that, my current code version delivers correct result when I multiply the rows by 2. After that, the execution time takes too long, therefore, the compilation is automatically stopped. As far as I am concerned, I think that the problem generates not from the way I identify the case in the memory with the chosen key-value, but from the way my algorithm works from a structural point of view. I think that I should probably check more often the memory in the other methods' recursive steps as well (maybe in *match/2*?), before exploring a possible solution. I have, however, not managed to put this idea into practice.