Springs Part II: Revisited report according to the received comments

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. As we go, the possibility of a valid solution will first be checked in the memory. If it exists, computation time will be saved and the associated 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
```

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.

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/2* method.

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

def readfile(mem) do
  {:ok, input} = File.read("test.txt")
  rows = String.split(input, "\n")
  descr = Enum.map(rows, fn row -> splitline(row,3)  end)
  result = Enum.map(descr, fn({springs_list, damaged_list }) ->
      explore(springs_list, damaged_list, mem)end)
  result

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/3* 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. **Unlike my previous attempt** where I did not have memory hits because I was describing how I got to a specific situation (path which will most likely always be unique because of all the possible combinations), this time I will properly describe the situation itself by using the list of atoms describing the springs (*list*) and the numerical description of the damaged springs (*d*).

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

Of course, the *explore/3* 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([], [], mem) do {1, mem} end
def explore([], _, mem) do {0,mem} end
def explore(_, [], mem) do {1, mem} end
def explore([:op | t], d, mem) do checkmemory(t, d,mem) end
def explore([:dam| t], [n |d], mem) do #major change in the approach here
  case damaged(t, n-1)   do
    {:ok, rest} -> checkmemory(rest, d, mem)
    :no -> {0, mem}
  end
end
def explore([:ukn | t], d, mem) do
  {ans1, mem} = checkmemory([:dam| t], d, mem)
  {ans2, mem} = checkmemory([:op | t], d,mem)
  {ans1 + ans2, mem}
end
```

Once again, **unlike my previous attempt**, this time I do not generate a full solution and then check its validity against the numerical description. Instead, once I reach a damaged spring, I analyze the possibility of that particular solution by comparing in **parallel** the numerical description. This is done by using a separate method, namely *damaged/2*.

```
def damaged([], 0) do {:ok, []} end
def damaged([], n) do :no end
def damaged([:dam | t], 0) do :no end
def damaged([:dam | t], n) do damaged(t, n-1) end
def damaged([:op | t], 0) do {:ok, t} end
def damaged([:op | t], n) do :no end
def damaged([:ukn | t], 0) do {:ok, [:op | t]} end
def damaged([:ukn | t], n) do damaged([:dam | t], n) end
```

In the *damaged/2* method, apart from the base cases, the solution may be okay only if you encounter a damaged spring and the number I am looking at is greater than zero. Therefore, the exploration continues using the recursive call. If one encounters an operational spring, the solution may be valid only if the number is equal to zero, while when encountering an unknown spring, depending on 'n', the spring is either replaced with a damaged spring and the recursive call continues or it is substituted by an operational one an returned as okay.

Benchmark

Once the implementation is ready and the execution time is analyzed, dynamic programming is quite impressive. The presented below benchmark has been executed for the input file provided by Advent of Code 2023 which is made out of approximately 1000 rows.

n	ExecutionTime[us]	Time/PrevTime
1	70K	-
2	122K	1.7
3	296K	2.42
4	477K	1.6
5	800K	1.6
6	1100K	1.3
7	1500K	1.3

Table 1: Benchmark analysis of the dynamic implementation for the 6 line code example.

As it can be noticed, with dynamic programming, a large data set can be processed correctly, pretty fast. What's particularly interesting is that as the number increases, more and more memory hits take place and therefore less computations have to be done. This is best seen in Table 1, when multiplying by 6 and then, by 7 since there is a quite small difference in the execution time.