Springs: Part 2

Ayda Ghalkhanbaz

Spring Term 2024

Introduction

This task is a build up on the previous assignment where we are going to implement a more efficient way to solve the puzzle. Finally we will run some benchmarks and discuss the results and execution time of these methods.

Method

This assignment is divided into two parts, where in the first part we only need to extend each line n times and then measure the execution time of solving the puzzle with a varying n, while the second part involves a dynamic implementation and its effects on the execution time.

Extending a row

The implementation of this part was quite straight forward, since we only had to define a new function that extends each line n times, where descriptions had `:unk` between each duplication. However, it seems to matter how we implement this function (the reasons will be discussed in upcoming sections). So for this part, first we have a `extend/2` function that multiplies a list with n using the in-built function `List.duplicate/2`. The `extend/2` function has been called in the `parse/2` function.

The second way of implementing this function is using `kernel++` operator and recursive call of `extend/2` function. You'll find the second implementation in the code snippet below.

```
def extend({springList, numberList}, 1) do
  {springList, numberList}
end
def extend({springList, numberList}, n) do
  springList = springList ++ [:unk] ++ springList
  numberList = numberList ++ numberList
  extend({springList, numberList}, n-1)
end
```

Dynamic programming

Implementing this part of the program was simple since the instruction was provided through the assignment step by step. The only changes we had to make was creating a memory and passing it as a parameter to the functions. This memory keeps track of every combination we made so far together with their possible solutions, in this way the program won't do the same computations over and over.

A new function was defined for this part that looks up the memory to see whether that particular line is already in the memory and in this case returns the answer; otherwise it will find the possible solutions and add them into the memory. A memory is simply an in-built `Map` and the new function `cacheSearch/3` is shown in the code snippet below.

```
def cacheSearch(springs, numbers, mem) do
  case Map.get(mem, {springs, numbers}) do
    :nil ->
      {answer, updatedMem} = count_combos(springs, numbers, mem)
      {answer, Map.put(updatedMem, {springs, numbers}, answer)}
    answer -> {answer, mem}
  end
end
```

Result

Table 1 presents the execution time of two implementations of the first part, comparing them to the execution time of the dynamic programming version for solving the puzzle as the value of n, representing the number of multiplications in each row, grows. The two first columns is the execution time of the primary algorithm with 2 different approaches of `extend/2` function. The third column (dynamic/duplicate) is the dynamic approach where the`extend/2` was implemented using `List.duplicate/2` method and the last column is the dynamic implementation using the `kernel++` for the `extend/2`.

n	**Execution time**			
	List.duplicate	**kernel++**	**dynamic/duplicate**	**dynamic/kernel**
1	0.33	3	0.59	1.65
2	0.16	0.2	0.45	0.42
3	0.38	4	0.6	0.97
4	3.4	90000	1.14	4.34
5	38.5	-	2	11.43
6	434	-	2.3	31.43
7	6500	-	2.76	114
8	-	-	3.14	451.43

Table 1: Execution time of solving the puzzle using different approaches in ms

Discussion

By comparing the first two columns in table 1, we can observe the difference in execution time between `kernel++` and `List.duplicate/2`. The `List.duplicate/2` is an in-built function specially designed for duplicating a list n times, optimised for performing this kind of operation. In contrast, `kernel++` is a more general and flexible way of concatenating two lists of any type, which offers a slower performance. Additionally we see that it is nearly impossible to get a response from `kernel++` implementation when n is 5. While the `List.duplicate/2` has no trouble handling the task up to n = 7.

The results also reveal that the execution time of solving the puzzle using only the `extend/2` function has an exponential behaviour which confirms our expectations from theory. The reason is that the algorithm explores all possible solutions recursively without memorising any of them. This leads to multiple re-computations for each subproblem and therefore the time complexity of this algorithm is $\mathcal{O}(2^n)$.

While the dynamic programming utilises its benefits of memorisation, it can immediately return an already computed solution. This is why the program doesn't have to re-compute anything at all and in this way avoids redundancy. Consequently, the program is optimised in an efficient way and offers a linear time complexity, denoted as $\mathcal{O}(n)$. The theory can be confirmed by looking at the outcomes presented in table 1 (third and fourth column).

While both approaches offer faster algorithms for solving the puzzle, it is evident that the dynamic program using `List.duplicate/2` for row extension is even faster and it behaves closer to linearity compared to the `kernel++` version.