Hot Springs part 1 - ID1019

Pontus Morin

Spring Term 2024

Solution

To solve this problem, dynamic programming can be utilized. Dynamic programming breaks down a large problem into much smaller problems and at each point the focus lays on what can be done at this current time.

Before that the input needs to be parsed into a more workable representation compared to a long string. First, the string is split by each newline. Next, each row is split by the white space between the sequence and the mapping. The last parsing splits each symbol into an array and the numbers in the mapping was converted to integers. The final representation contains a list of rows where each row is represented as an tuple with two lists.

```
[{ ["#", ".", "#", ".", "#", "#", "#"], [1, 1, 3] }, ...]
```

The first entry in the list ie the first row is the first to be evaluated in the function `check_row`. The function checks what type if symbol which is located in the front of the list and if "." is found we just move on to the next symbol. If a list consist of only dots this would be done recursively until the list was empty. When the recursion reaches an empty list a return will be made. If the map list is also empty the accumulated value + one is returned, this is the only "winning" case. Otherwise the acc is returned.

```
def check_row({[], []}, acc) do acc + 1 end
def check_row({[], _}, acc) do acc end
```

The next possibility is if the symbol found is a broken spring, represented by "#". For this, more considerations needs to be done as the broken springs needs to match the pattern from the mapping. At first, this logic was handled by the same function which resulting in a bunch of overloaded methods and some difficulties to handle certain edge cases. This was exchanged for a verification method called `verify_next`. The function `verify_next` works as a "peek" function an looks at the next element in the list. Depending on what the next symbol is the function either returns an ok and what to proceed with or a fail which invalidates that option. The key concept of this function is that if a "#" is found the mapping needs to match the next symbol. Meaning if the next symbol is a "." the mapping must be 0. If the next symbol is a "#" it the mapped number must be greater than 0 or the sequence is invalid.

```
def verify_next([], num) when num > 0 do :fail end
def verify_next([], _)  do {:ok, [], :next} end
def verify_next(["." | rest], n) when n == 0 do
  {:ok, rest, :next} end
def verify_next(["." | _], _)  do :fail end
def verify_next(["#" | _], n) when n == 0 do :fail end
def verify_next(seq = ["#" |_], n)  do {:ok, seq, n} end
```

The only case where the `check_row` does not call `verify_next` when a "#" is found is if it is found and the mapping list is empty since,

How about unknown springs

In the task, an unknown condition is identified as "?", ie it can be either a working (".") or broken ("#") spring. When a "?" is reached it has two option, and the dynamic approach is simply to just test on of the options and see if it results in a valid combination. When the first choice has been evaluated, the second alternative is controlled as well. This is done for all found unknown symbols and every evaluation resulting in a match increments the accumulator.

```
def check_row({["?" | rest], map},acc) do
  acc = check_row({["#" | rest], map}, acc)
  check_row({["." | rest], map}, acc)
end
```

For this to work the "peek" function also needs to know how to handle an unknown state as a sequence may consist of a "#" followed by a "?". In this function there is two conditions to consider. The first is where the sequence ["#","?", ...] is being evaluated and the mapped number is 1. In this case the "?" must be a "." and is therefore skipped. The other case implies that the mapping requires a sequence of "#" and for this the "?" is exchanged for a "#".

```
def verify_next(["?" | rest], n) when n == 0 do
  {:ok, rest, :next} end
def verify_next(["?" | rest], n)  do {:ok, ["#" | rest], n} end
```

Conclusion

This approach of solving a problem works like searching a tree with a depth first approach. All possibilities are being tested and is therefore more of a brute force approach. With a small sample, this approach works good enough but as the sample grows the execution time grows exponentially since all nodes in the "tree" is "evaluated". To solve scaling problems, a more true dynamic approach with a cache where the found solutions are stored is needed and will be used for part 2.