Hot Springs Advanced: 1/5 på Trivago

Praanto Samadder

19 February 2024 (Extended deadline)

Terminologies

This assignment uses the terminologies as defined in the basic assignment.

I have replaced some of the original Terminologies in the problem PDF with clearer version of my own.

I make use of the word 'pattern' to denote the left-hand side of my description. I call the right-hand side 'clue'.

$$\underbrace{\texttt{\#\#.\#\#.\#\#\#}}_{\text{pattern}} \quad \underbrace{\texttt{2, 2, 3}}_{\text{clue}}$$

Every element of our clue will be referred to as `clue_index`.
The full code can be found at this gist here.

1 Bringing back Nostalgia

In this approach of solving the Springs problem, I have decided to revisit memories from Algorithms and Data Structures and re-implement our stack powered binary tree.

2 But first, the parser!

The parser isn't very difficult to implement either. The pattern and the clue is separated with a space. Splitting there leaves us with, well, the pattern and the clue.

The clue can then be parsed in to a list. Throughout this report, we will be using `clue` to call this list and `clue_index` to denote the element of the clue we are currently looking at. This will become more relevant in the future.

We can parse the pattern into a binary tree using the following steps:

- Go through every character in our pattern (which is a string)
- Use a `case` to check if it is working or unknown

- Two branches if a node is unknown, left branch if it is working, right branch if it is not working

This leaves us with a perfectly balanced binary tree (not that we care much about balancing) if we look from the root of the tree.

3 Traversing the tree

3.1 Concepts that come into play

3.1.1 'Bygg min svar'

While traversing down our tree, we 'build' our solution. But what does 'building' mean? Well, we use a `result::String()` to keep track of our progress so far. Once we hit a `:broken` branch, we add a pound (#) to our `result`. Once we hit a `:working`, we add a period (.) to `result`.

3.1.2 Counting broken hot springs

How do we determine if we have the maximum number of rows in a subsection or our pattern as allowed by the clue i.e. we have matched a clue?

Well we can use a `rows` variable to count the number of broken springs in a row. If we encounter a `:broken` spring, we increment `rows`, if we encounter a `:working` or an `:unknown` spring, then we reset `row`. Hold that thought by the way. We will explore edge cases for this in section 3.3.

3.1.3 Still got more clues?

And finally, we must determine if we have 'explored' all our clues. We use a `clue_index` to determine which `clue_element` we are currently working with.

3.1.4 And lastly, the stack

We also make use of a `stack`. We can implement a stack in Elixir using only four functions.

```
defmodule Stack do
  def push(stack, {:node, nil}) do {stack, nil} end
  def push(stack, a) do
    id = length(stack) + 1
    {[{:s_el, id, a} | stack], id}
  end
  def pop([]) do {:stack_empty, []} end
  def pop(from_stack) do
    [head | tail] = from_stack
    {head, tail}
  end
end
```

Listing 1: Simple LIFO stack

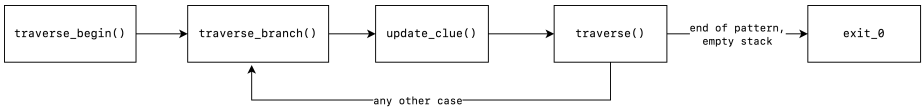

Figure 1: Message loop

3.2 The message loop

Recursion has been used but the recursive calls are structured in a, sort of, message loop.

3.3 Traversing `:unknown` branch

When encountering this branch we must (a) push our right branch into our stack (we can ignore if our right branch is a `nil` as our stack already handles it for us) and (b) traverse down the left stack. We do not increment our `rows`, we do not update our `clue_index`.

```
def traverse_branch({:n, :un, left, {:n, nil}}, {:d, clue, c_in, rw
    , res}, stack) do
    traverse(left, {:d, clue, c_in, 0, res <> "."}, Stack.push(
    stack, {:st_el, {:n, :b, {:n, nil}, {:n, nil}}, rw, res}))
  end
```

Listing 2: Example of how a branch is traversed

3.3.1 Edge cases

What happens when `rows == clue_element`? Well, that means a right branch (a broken branch) at this stage would result in an erroneous solution. We must, therefore, force-traverse the left branch.

```
def traverse_branch({:n, :u, l, _}, {clue, ci, true}, {_, _}, {
    rw_pat, res, _}, stack) do
  update_clue(l, {clue, ci, true}, {0, false}, working_res(rw_pat,
    res), stack)
end
```

Listing 3: Edge case ignoring the right branch

3.4 Traversing a `:broken` and `:working` branch

Pretty straightforward. Hit a `:working`? Go left. Hit a `:broken`? Go right.

3.4.1 Edge cases

No edge cases. These functions behave nicely and are well-trained.

3.5 The deal with the clues

All of our `traverse_branch()` methods call the `update_clue()` method. Update clues essentially checks if the `rows` equals the maximum number allowed. But not without it's edge cases it doesn't.

3.5.1 Rows allowed met but left node is `nil`

We are expected to take a left branch when our `rows` reaches maximum allowed. But what happens if our left node is `nil`. Well, this means we have reached a dud end and we must restart traversing from our stack.

```
1 def update_clue({:n, _, {:n, nil}, _}, {clue, clue_index, _}, {_,
      true}, {rw_pat, res, _}, stack) do
2     traverse({:n, nil}, increment(clue, clue_index), {0, false}, {
      rw_pat, res, true}, stack)
3 end
```

Listing 4: `update_clue() when left node is nil, maximum rows have been reached`

3.5.2 Forcing a left traversal

When we have reached the maximum number of rows (and our left branch is not `nil`), we must force traversal down the left node. We simply ignore that the right branch exists, no adding to any stack.

```
1 def update_clue({:n, :u, l, _}, {clue, clue_index, _}, {_, true}, {
      rw_pat, res, _}, stack) do
2     traverse_branch(l, increment(clue, clue_index), {0, false},
      working_result(rw_pat, res), stack)
3 end
```

Listing 5: `update_clue()` Forced left-branch traversal

3.6 The `traverse` method

The role of the `traverse` method is to determine if our program should continue execution or exit out of recursion.

If we have reached the end of our pattern (meaning the length of our `raw_pattern` equal `result`) and our stack is also empty, well then we've had a good run but it's time for exit code 0.

If we have reached the end of our pattern (meaning the length of our `raw_pattern` equal `result`) and our stack still has elements in it, we go check them out! We have a stack traversal. We can also take this opportunity to see if the result we have in hand is a valid or a failed result (we can make this check by seeing if our clues have been fully explored and our `clue_index` == length of our clues list.)

In any other case, things will be normal and we can continue our recursive excursion by calling `traverse_branch()`.

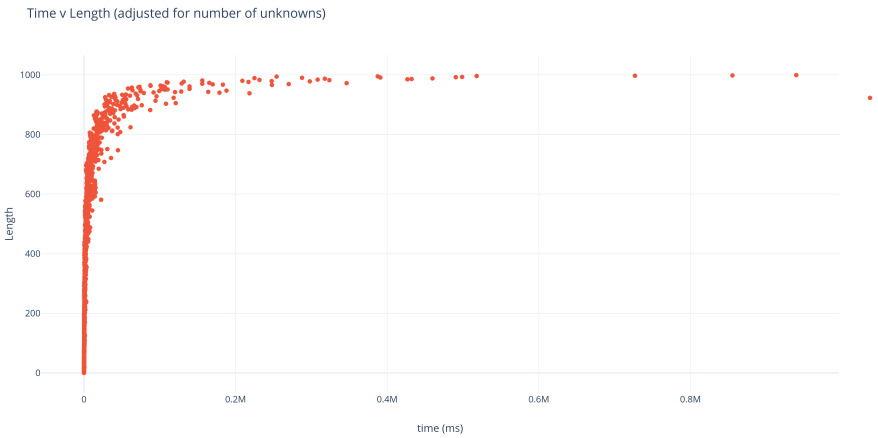

Figure 2: Graph, time taken vs length (adjusted for the number of unknowns)

4 Benchmark

Benchmarking this can be quite tricky since the time taken to find all possible solutions depends on not only the length of the pattern but also the number of unknowns.

Figure 1 shows the time taken to find all possible solutions as the the length of the pattern goes up. This graph is adjusted for the number of unknowns. When adjusted for the number of unknowns in our graph, our tree traversal shenanigans should resemble an average-case binary tree traversal. Figure 1 shows a time complexity of $O(\log n)$ which matches our hypothesis.

Time taken to traverse the list will go up with the number of unknowns and the length of our pattern.

But one thing is straight from our benchmark: which so many edge cases, pattern matching nightmares and no many broken hot springs, I have left my review on this resort.

Once again, the final version of the code can be found at this gist here.