Springs 5.1

Simon Yllmark

February 16, 2024

1 Introduction

In this assignment I am going to do the same thing as the previous but with memory. The idea is that the program should remember previous solution to specific problems so it does not have to solve the same problem twice. This would then decrees the time complexity in comparison to the previous problem.

2 Memory solution

The previous program work like a backtracking algorithm in the sense that it calculates all of the possible outcomes. The time complexity of the previous algorithm is $\mathcal{O}(n^2)$ because the search structure is similar of a tree and it calculates all the possible outcomes.

The memory solution make the time complexity of the new algorithm is closer to $\mathcal{O}(n)$ because it remember the searches done previously in a linked key-value list. Searching one element in a the list has a time complexity of $\mathcal{O}(n)$, because searching a linked list is a linear operation. That help make the new algorithm's time complexity closer to $\mathcal{O}(n)$.

3 Check function

To solve this problem a check function check() will need to be implemented. The idea is that function will take in 4 parameters element, numbers, key and module. The first parameter is the characters, the second the integer list, the third the value in the memory mapped list and last is a reference to the module who contain all the necessary key-value linked list function. Here is an example of how the check() should be implemented:

```
def check(element, numbers, key, module) do
case module.lookup({element,numbers}, key) do
  nil ->
    {solution, key} = counting(element, numbers, key, module)
```

```
      {solution, module.store({element,numbers}, solution, key)}
    solution ->
      {solution, key}
  end
end
```

First the program check if the search is in the list, if it is in the list then it returns the solution and the key, the list is a key value mapped list. If not then the function search for the solution and return a tuple containing the solution and the return value from the function storing the new solution in the list. This is the general idea of how the function should be implemented.

4 Merge check() and counting()

One problem that need fixing now is that the counting function counting() need to return the solution and a key as a tuple. The general idea is that the check() should be added as an extra helper function to the counting but merging the two functions should not change how the counting() algorithm thinks, meaning the algorithm's operation and logic should not be changed even if what it return should.

I could create a memory function memory() that call the counting function and the counting function should then take in two extra parameters as a key and the list module. I tried it and it actually worked. Here is the new counting function counting():

```
def counting([],[], key, module) do {1,:ok} end
def counting([],_, key, module) do {0,:ok} end
def counting(_,[], key, module) do {1,:ok} end
def counting(list1,list2, key, module) do
  [element|rest] = list1
  [num|rest2] = list2
  case element do
    ?. -> counting(rest,list2, key, module)
    ?# -> case broken(list1,num) do
      {:ok, back} -> counting(back,rest2, key, module)
      false -> {0,:ok}
    end
    ?? -> case broken(list1,num) do
       {:ok, back} -> {num1,:ok} = counting(back,rest2, key, module)
       {num2,:ok} = counting(rest,list2, key, module)
       {num1+num2,:ok}
       false -> counting(rest,list2, key, module)
    end
  end
end
```

The logic is the same but counting returns a tuple instead of an numeric value. Here is the new memory function that calls counting():

```
def memory(m, t) do
  memory(m, t, Memory)
end

def memory(m, t, module) do
  mem = module.new()
  {solution, _} = check(m, t, mem, module)
  solution
end

def check(element, numbers, key, module) do
  case module.lookup({element,numbers}, key) do
    nil ->
      {solution, key} = counting(element, numbers, key, module)
      {solution, module.store({element,numbers}, solution, key)}
    solution ->
      {solution, key}
  end
end
```

So, in the eval() function the function memory() is called instead of function counting(). Then the function check if a solution to the given problem already exist. If it exist the it returns the answer and if not it search for the answer by calling function counting() and the stores the solution along with a key the key value list module module.

5 Conclusion

The program would be better implemented if for example it receives problem b then it also solve problem a and c. This would help the program time complexity closer to $\mathcal{O}(n)$. That would be better but how to implement that would be the next step.

The functions in section 4 are tested and thus I know that they work. The algorithm of the memory() function mentioned in section 4 would improve the time complexity of the counting() function if it works. It does work because it has been tested and thus can handle cases of larger trees, meaning longer sequences of elements and number for example:

```
['???.###????.###????.###????.###????.###', [1,1,3,1,1,3,1,1,3,1,1,3,1,1,3]]
```

Which it does because I tested it and here an the example

```
iex(19)> a = '???.###????.###????.###????.###????.###'
'???.###????.###????.###????.###????.###'
iex(20)> b = [1,1,3,1,1,3,1,1,3,1,1,3,1,1,3]
[1, 1, 3, 1, 1, 3, 1, 1, 3, 1, 1, 3, 1, 1, 3]
iex(21)> S3.memory(a,b)
1
iex(22)>
```

Thus, the end result is an improved program to the program from the previous assignment that can solve larger recurring sequences by just remembering the solution.