Springs 5.1.1

Simon Yllmark

February 21, 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. The counting function counting() should as well return a tuple containing the memory and the solution. This will enable the program to remember the solution to a problem that have been executed. Here is the new counting function counting():

```
  def counting([],[], mem, module) do {mem, 1} end
  def counting([],_, mem, module) do {mem, 0} end
  def counting([?.|rest],[], mem, module) do
    check(rest, [], mem, module)
  end
  def counting([??|rest],[], mem, module) do
    check(rest, [], mem, module)
  end
  def counting([?#|rest],[], mem, module) do
    {mem, 0}
  end
  def counting(list1,list2, mem, module) do
    [element|rest] = list1
    [num|rest2] = list2
    case element do
      ?. -> check(rest,list2, mem, module)
      ?# ->
        case broken(list1,num) do
        {:ok, back} -> check(back,rest2, mem, module)
        false -> {mem, 0}
      end
      ?? -> case broken(list1,num) do
         {:ok, back} ->
            {mem1, num1} = check(back, rest2, mem, module)
            {mem2, num2} = check(rest, list2, mem1, module)
            {mem2, num1+num2}
         false ->
          check(rest, list2, mem, module)
      end
    end
  end
```

The logic is the same but counting() returns a tuple instead of an numeric value containing memory and the solution. Here is the new function sequence where counting() is one of the functions called:

```
def evaluation([]) do [] end
def evaluation([h|t]) do
  [eval(h) | evaluation(t)]
end

def eval({:spec, a, b}) do
  mem = Memory.new()
  {_, solution} = check(a, b, mem, Memory)
  solution
end

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

The check() function will check if a previous solution is stored in memory and if not then the problem will be executed by the counting() function and then that solution will be stored in memory. This will make the program more efficient in cases of repeating patterns but if all the input is unique then this program might be a bit slower than the previous program from the E graded task.

5 Conclusion

The program most optimal case of inputs is the case where there are multiple repeating patterns in the input and in the case of an large input where the sequence length of the input is around 8 characters then it is quite likely. This would help the program time complexity closer to $\mathcal{O}(n)$, because repeating problems would not have to be executed.