Task 9 - Board Cutting

Casper Kristiansson

February 22, 2022

Introduction

Task nine of the course Programming II consisted of solving a wood cutting problem using dynamic program. The goal of the task is that the author should be able to firstly solve and find the lowest cost of a certain wood cutting scenario and then improve that algorithm. This can be done by taking apart the algorithm and improving its different counterparts. The author will be discussing the different algorithms that was develop and the performances of them.

Method

The author started off by watching the lecture on "Dynamic" which covered the basics of dynamic programming. The lecture includes in the basics understanding of dynamic programming and how different type of algorithms could be improved using numerous different type off strategies.

Result

The author will be dividing up the result section into four different sections, one for each sub task. The author will mostly focus on discussing the last algorithm because it has the best performance compared to the other solutions.

The basics

The first task of the problem is to implement basic wood cutting where the goal is to find all the different ways a tree could be cut depending on the input. This could be done by constantly dividing the remaining wood pieces by left or right and calling the function again. The algorithm will find all the different scenarios that the pieces could be divided into.

```
def split(seq) do split(seq, 0, [], []) end
def split([], l, left, right) do
  [{left, right, l}]
end
def split([s | rest], l, left, right) do
  split(rest, s + l, [s | left], right) ++
  split(rest, s + l, left, [s | right])
end
```

Figure 1: Basic wood cutting

The lowest cost

The second task of the exercise consisted of implementing a function which should be able to return a configuration with the lowest cost of cutting the tree into the desired pieces. The function does this by checking if either the left or right path has the lowest cost and choosing the correct path. The upcoming algorithm in the report builds on the same fundamentality. Figure two shows the basic case for comparison between the different costs. Unfortunately, this type of brute force solution is incredible slow and has an incredible hard time trying to solve a wood cutting problem with more than nine pieces.

```
def cost([s | rest], l, left, right) do
  costLeft = cost(rest, s + l, [s | left], right)
  costRight = cost(rest, s + l, left, [s | right])
  if costLeft < costRight do
    costLeft
  else
    costRight
  end
end
```

Figure 2: Lowest cost

The memory

The memory is implemented by creating a check function which will check if the current calculation has already been made. If it has the cost is just returned and if not, the algorithm will calculate it and add it to the memory. This improvement improves the algorithm by a lot and the algorithm is now able to solve wood cutting problems up to around fifteen within a reasonable run time.

```elixir
def cost(seq) do
  {cost, tree, _} = cost(seq, Memo.new())
  {cost, tree}
end
def cost(seq, mem) do
  {c, t, mem} = cost(seq, 0, [], [], mem)
  {c, t, Memo.add(mem, seq, {c, t})}
end
def check(seq, mem) do
  case Memo.lookup(mem, seq) do
    nil ->
      cost(seq, mem)
    {c, t} ->
      {c, t, mem}
  end
end
```

Figure 3: Improving using a memory

Improvements

The last task includes making a couple of changes to the improved algorithm. The first thing that was improved was where mirror solutions was still explored because they were just added to the memory without adjusting the key. For example, the algorithm would make two different calculations for the same problem, [1,2] [2,1] where they have the exact same cost. This could be solved by simply sorting the current tree (key) before adding it to the memory. Because of the order of the list, the algorithm is required to reverse the list before performing the memory lookup.

```elixir
def cost(seq) do
  {cost, tree, _} = cost(Enum.sort(seq), Memo.treeNew())
  {cost, tree}
end
def cost([s], l, left, [], mem) do
  {cost, tree, mem} = check(Enum.reverse(left), mem)
  {cost + s + l, {s, tree}, mem}
end
```

Figure 4: Solving mirror problems

The second thing that could be improved is to implement a better type of memory. The memory will have a much better lookup and insert time

than a normal map. But the new memory will unfortunately be extremely unbalanced because the keys are sorted before they are inserted into the memory. Because of this the first node :one will have most of the costs stored in it.

```
def treeInsert([], [current | []], value) do
  [{current, value, []}]
end
def treeInsert([], [current | next], value) do
  [{current, nil, treeInsert([], next, value)}]
end
def treeInsert([{treeN, _, _} | _] = tree, [current | []], value) do
  if treeN == current do
    tree
  else
    tree ++ [{current, value, []}]
  end
end
def treeInsert([{treeN, treeValue, branches} | rest],
      [current | next] = key, value) do
  if treeN == current do
    [{treeN, treeValue, treeInsert(branches, next, value)}] ++ rest
  else
    [{treeN, treeValue, branches} | treeInsert(rest, key, value)]
  end
end
```

Figure 5: Improved memory

Figure five consists of the method which inserts a key and value into the memory. The memory is built on the structure that it has different layers where each layer is represented as the length of the keys. For example, if the key [1,2,3,4,5] is inserted into the memory the function will create a depth of five layers.

Discussion

Figure six consists of a table which includes the benchmarks for the three different algorithms. The algorithm is dramatically improved by just using the memory where the run-time is approximately improved by 12000x when ten pieces are inserted. The last algorithm still does improve the run-time by quite a bit, by solving the mirror problem and implementing an improved memory.

Amount	**Brute force**	**Basic Memory**	**Improved Algorithm**
7	0.03	0.0006	0.00041
8	0.49	0.0016	0.0006
9	11	0.005	0.002
10	200	0.017	0.006
11	-	0.056	0.02
12	-	0.19	0.067
13	-	0.73	0.2
14	-	2.6	0.7
15	-	9.5	2.2

Figure 6: Benchmark of the different algorithms in seconds (s)

This problem touches the basics of dynamic programming where the goal is to find improvements to different type of algorithms. But even with the different improvements to the algorithm the solution is still not able to solve problems with more than thirty pieces within a reasonable amount of time. A runtime improvement with more inaccurate result would be to declare a maximum depth to the comparison method where it will count the cost up to a specific number of cuts. This way the result will be worse, but the runtime will be extremely improved. This type of algorithm can be seen in chess bots where the algorithm often will look x amounts of moves ahead. The bigger the depth the better the bot.