Project Pseudocode and Complexity Analysis v1.1

By Brett Huffman
Oct 16, 2021

Overview

For the Gifting project, multiple versions of the algorithm have been used trying to decide exactly how to attack the problem. As of now, the course of action seems to give very advantageous results.

The goal has been, as quickly as possible, to drastically prune branches as early as possible. So, in the current application version, it is pruning at the same time it builds the search trees. This yields the advantage of not having to go through the trees multiple times to build and prune.

On initialization, it sets a "best" average gift amount to infinity. It also creates an empty "best" combination vector. These "best value" objects will be used to keep track of the best solution as all the possible combinations are explored.

The program starts creates a wide tree which is basically the number of children high. However, it is very wide as it considers each permutation of child and gift combinations. The good thing is that it prunes branches on-the-fly to get rid of bad combinations as early as possible.

Once a possible solution is found, it is compared against the current best-value combination. If it has a lower P/N value than the current best-value combination, it becomes the new best-value, and the branches by which the combination was found is kept.

At the end of the analysis, the best-value has been found and is displayed to the user.

Along with early pruning, a number of heuristic tactics were investigated. The best tactic seemed to be to order the children oldest-to-youngest and the gifts youngest to oldest with the "any age" gifts at the end. Through investigation, that seemed to yield good results.

My hope is that this early-pruning approach will yield faster results and arrive at the results as early as possible.

Pseudocode and Analysis

There are really three portions of code that must be analyzed: Project Setup, FindPermutations, and the Calculation Phase.

Project Setup

In setup for this phase, the structures and the lookup table will be created to support the creation of the final solution.

Pseudocode	Complexity Analysis	Space Complexity Analysis

// Initialization
Vector Children
Vector GiftItems
Vector MediumGifts LargeGifts // indexes
Vector<Vector> ChildBranches
Int** childGiftLogicTable // Fast lookup table
Float overallAvgPrice
// Counters
nChildCount = 0
nGiftCount = 0
 	
LoadArrayFromFile(inputFile)
Foreach Child => Add To Children
Foreach Gift => Add to GiftItems
If medGift => Add index to MediumGift
If lrgGift => Add index to LargeGifts
 	
If lrgGift.count < child.count || medGift.count < child.count
exit
// From running totals
overallAvgPrice =
totalPrices/totalChildren
	All O(1)
	
	O(C)
	O(MG)
	O(LG)
	
	O(1)
	
	O(1)
	
	Overall: O(C + MG + LG)
		O(C)
		O(G)
		
		Overall: O(C+G)

// Create the Logic Table
Init Table to Children.Count * (LgGift+MedGift).Count
 	
Foreach child c =>
Foreach totalGifts g =>
If c.age within gift.age ranges
Table[g,c] = 1
Else
Table[g,c] = 0
	O(1)
	
	O(C * totalGifts)
		O(C * totalGifts)

	Overall: O(C * totalGifts)	Overall: O(C * totalGifts)

FindPermutations

One of the keys to successfully pull off this implementation was to build a function that will always return the vector of allowed permutations for a given scenario. This function is called at every branch to give an accurate list of possible sub-branches.

As stated in my Academic Integrity Statement, the following website was used as a model for this Permutation function:

NewBeDev.com. 2021. https://newbedev.com/generating-combinations-in-c

<table>
<tr><th>Pseudocode</th><th>Complexity Analysis</th><th>Space Complexity Analysis</th></tr>
<tr><td><pre>FindPermutations(n, exclusionSet)

// Initialization (n is possible elements)
  Create vector<int> permutations

  for each item from 1 to n =>
    // Create mask
    let vector<bool> v(n)
    fill vector to j length with 1

    do
      let vector<int> trackingVector
      for each i in 0 to n
        if v[i] not in exclusionSet
          trackingVector.push(i)

      if trackingVector not empty
        add trackingVector to permutations

    while (std::prev_permutation(v.begin(),
      v.end()));

  return permutations // All possible
                      // values
                     // not in exclusionSet</pre></td><td>O(1)

O(n)

O(n)

O(1)
O(n)
O(log n)*
O(1)

O(1)
O(1)

O((v.end-v.begin)/2) => O(v)
(per docs**)

Overall: O(n²)</td><td>O(n)</td></tr>
</table>

* The search of the exclusionSet is done with the Binary Search algorithm yielding O(log n) complexity

** Complexity analysis found in system documentation here:
https://en.cppreference.com/w/cpp/algorithm/prev_permutation

The overall time complexity of this FindPermutations function is $O(n^2)$.

Calculation Phase - Generate and Calculate Tree

This phase is actually building the tree. This tree is theoretically children.count deep and Children*Gifts wide. However, because so much pruning happens during the run, many branches are never made. The tree is still very wide, but shallow.

As will be shown, pruning – essentially not adding them to the final branch Vector – happens if any gifts in that branch do not meet the age requirements of the respective child or if a child does not have both a large and medium gift in their respective branch.

Additionally, the tree is built reclusively depth-first to limit the size of the tree in memory.

<table>
<tr><th>Pseudocode</th><th>Time Complexity Analysis</th><th>Space Complexity Analysis</th></tr>
<tr><td>
<pre>
// Kick off the tree building process
populateTreeAndPrune()

  // keep track of used gifts
  let vector<int> = vecUsedGifts
  processChild(0, vecUsedGifts)

// Recursive function to process each child
processChild(currentChild, vecUsedGifts)

  vector<int> GiftItems
  vector<int> GiftCombos
  vector<int> NewUsedGifts
  vector<int> NewChild

  let NewChild = currentChild + 1

  // Terminating clause *success*
  // A good branch was found. Now
</pre>
</td><td>Overall call:
O(Children*
Gifts)

O(1)
O(1)
O(1)
O(1)</td><td>O(Gifts)

O(Gifts)
O(Children*
Gifts)
O(Gifts)
O(Children)</td></tr>
</table>

`// calculate the path cost`		
`if(newChild > vecChildren.size())`		
`  // Calculate`	O(1)	
`  for i = 0 to GiftCombos.size()`		
`    // Iterate through each gift`	O(Children)	
`    For j = 0 to GiftCombos[i].size()`		
`      childTotal =`	O(Gifts)	
`     vecGiftItems[GiftCombos[i][j].price;`	O(1)	
`     // Subtract child total from P/N`		
`    fChildTotal = abs(childTotal –`		
`                  calcGiftPN)`	O(1)	
`    GrandTotal += ChildTotal`	O(1)	
`  // If new found Average is best so far,`		
`  // keep track of it as the new best val`		
`  If GrandTotal < bestFoundAvg`	O(1)	
`    vestFoundAvg = GrandTotal`	O(1)	
`    vecBestGiftCombos = GiftCombos`	O(1)	
`  return   // End of terminating clause`	O(1)	
`// Recursive Clause`		
`// Get the permutations for the gifts`		
`// left`		
`let vector<it> v =`		
`   FindPermutations(GiftItems.size(),`	$O(n^2)$	
`     UsedGifts)`		
`// Make a branch for each combo`		
`// unless we prune the branch`		
`let hasMedGift = false`	O(1)	
`let hasLrgGift = false`	O(1)	
`for i in 0 to v.size-1`	O(Gifts)	
`  vector<int>row = v[i]`	O(1)	
`  // If not 2 gifts, purge immediately`		
`  If row.size() < 2`	O(1)	
`    continue`	O(1)	
`  // for each gift combo`		
`  if MedGifts contains v[i]`	O(1)	
`    hasMedGift = true`	O(1)	
`  if LrgGifts contains v[i]`	O(1)	
`    hasLrgGift = true`	O(1)	

`// Prune if branch does not have both`
`// a large and medium gift`
`if not hasMedGift or not hasLrgGift`
`  continue`
 	
`NewUsedGifts.add(v)`
 	
`Sort(newUsedGifts). // Much better alg`
`                    // performance`
`// Recursively call into processChild`
`processChild(newChild, NewUsedGifts)`
 	
`// Finished branches (without children)`
`// just drop out of this function`
	O(1)
	O(1)
	
	O(1)
	
	O(log n)
	
	O(1)	

* The search of the newUserGifts vector is done with the Binary Search algorithm yielding O(log n) complexity

The overall worst-case time complexity for this function is O(Children* Total Gifts)
The overall worst-case space complexity for this function is O(Children* Total Gifts)