TECHNICAL REPORT FOR CS 504 PROJECT: PROBLEM 2

BENJAMIN ELBERT

Abstract. This report will outline the dynamic programming solution to solving the Counting of Constrained Operator Combinations problem proposed by R. Bunescu. We will outline the four steps to the dynamic programming solution, as well as analyzing the time and space complexity.

CONTENTS

1. Introduction 2
2. Characterizing the Structure of an Optimal Solution 2
3. Recursively Defining the Value of an Optimal Solution 2
4. Computing and Constructing the Optimal Solution 2
5. Analyzing the Time and Space Complexity 3
1. Introduction

The counting of constrained operator combinations problem asks the following:

**Problem 1.** Given a vector of integers, \( A = [A_1, A_2, \ldots, A_n] \) and a positive integer \( M \), how many ways, using \( op_i \in \{+, -\} \), can we make

\[
A_1 \, op_1 \, A_2 \, op_2 \, \ldots \, op_{n-1} \, A_n
\]

into a multiple of \( M \)?

Notice that we can check for whether a sum/difference is a multiple of \( M \) by checking if the arithmetic combination \( \mod M \) is 0. This idea will be key to our dynamic programming solution.

2. Characterizing the Structure of an Optimal Solution

For simplicity of notation, let us introduce the following: \( C(A_i, \ldots, A_j) \) will be used to mean a combination of the integers \( A_i \) through \( A_j \) using \( + \) or \( - \). Then, we have the following structure for our problem:

\[
(1) \quad C([A_1, \ldots, A_n]) \text{ is a multiple of } M \iff C([A_1, \ldots, A_{n-1}]) \, op_{n-1} \, A_n \mod M \equiv 0.
\]

We will use this fact when recursively defining our optimal solution. When we implement and use the aforementioned algorithm, we will essentially be keeping track of the ways we can combine integers such that \( \mod M \) we obtain 0.

3. Recursively Defining the Value of an Optimal Solution

To find the number of ways to combine our input integers so that they form a multiple of \( M \), we will populate a matrix of size \( M \times n \) where \( n \) is the length of \( A \) whose entries tell us how many ways to combine the elements \( A_1 \) to \( A_j \) (where \( j \) is the current column) to obtain that \( \mod M \) is 0. Let us now define this formula.

\[
C(i, j) = \begin{cases} 
1 & j = 1 \text{ and } A_1 \mod M = i - 1 \\
0 & j = 1 \text{ and } A_1 \mod M \neq i - 1 \\
C_{(\mod((i-1)-A_j,M)+1,j-1)} + C_{(\mod((i-1)+A_j,M)+1,j-1)} & j > 1.
\end{cases}
\]

In doing the problem this way, we are not just answering the question of how many ways to combine the input to make a multiple of \( M \) (obtained by looking at entry \((1, n)\) of the matrix), but also the ways to combine \([A_1, \ldots, A_j]\) to make it equal to \( c \mod M \) with \( 1 \leq c \leq M - 1 \) (obtained by looking at entry \((c+1, j)\)). In the next section we provide pseudocode for populating such a table and returning the number of ways to combine to make a multiple of \( M \).

4. Computing and Constructing the Optimal Solution

There is no solution to compute, per se, since there are possibly an exponential number of ways to make a multiple of \( M \), but we do need to construct the table and return the desired value. The following algorithm returns the possible combinations, but does not give the ways that those combinations are obtained.
1: count_combinations_to_m(A, M)
2: \[ n = \text{length}(A) \]
3: if \( M > 1 \) then
4: \[ \text{Count} = \text{zeros}(M, n) \]
5: else
6: \[ \text{return } 2^{n-1} \]
7: end if
8: \[ \text{Count}(\text{mod}(A_1, M) + 1, 1) = 1 \]
9: for \( j = 2 : n \) do
10: \[ \text{for } i = 1 : M \text{ do} \]
11: \[ k1 = \text{mod}((i - 1) - A_j, M) + 1 \]
12: \[ k2 = \text{mod}((i - 1) + A_j, M) + 1 \]
13: \[ \text{Count}(i, j) = \text{Count}(k1, j - 1) + \text{Count}(k2, j - 1) \]
14: \[ \text{end for} \]
15: \[ \text{end for} \]
16: \[ \text{return } \text{Count}(1, n) \]

The Matlab implementation is not much longer than the pseudocode itself, and with Python or other precompiled languages the code may be shorter.

5. Analyzing the Time and Space Complexity

We will first analyze the space complexity, though neither of the two are very difficult to determine. Throughout the program, for an input of length \( n \), we maintain a matrix of size \( M \times n \). Since we are able to return the number of combinations by making a call to this matrix, there are no other items to hold in memory. Thus,

\[ S(n) = \Theta(Mn). \]

It takes a constant number of operations to create the matrix and initialize the correct entry of column 1 to be a 1. After that, we enter the nested loop part, the outer of which executes \( n - 1 \) times, the inner \( M \) times. Thus, our time complexity is given by

\[ T(n) = (n - 1)(M) + c = Mn - M + c = \Theta(Mn). \]

The benefit to this problem is that both the time complexity and space complexity are small, but for really large \( M \) we are maintaining a significantly large (and probably sparse) matrix.