Minimizing hand displacement when playing the guitar: A Study of Programming Approaches

Richard Rodríguez and Sabrina Guilarte

Abstract – **This work poses the situation of playing a set of chords on the guitar as an optimization problem aiming to minimize hand displacement. It presents an analysis of the situation and explores the efficiency of using three different programming approaches (brute-force, greedy, and dynamic) to optimize hand displacement when playing a song on the guitar (following a list of chords). The brute-force algorithm has a complexity of O(3^n), which makes it an inefficient solution. The dynamic programming implementation has a complexity of O(n). It was not possible to find a greedy algorithm that correctly solves the problem.**

I. Introduction

When playing a song on the guitar, there are multiple ways to play the same chord. These variants correspond to differences in fret and finger positioning. The importance of choosing the right chord variant lies in the fact that depending on the previous chord played, the chord variant the musician chooses to play next can require a significant hand displacement, thus increasing the difficulty of the task.

With this situation in mind, one can wonder about whether this situation can be accurately modeled and approached by creating an algorithm. Is it possible to create a program that efficiently chooses the best combination of chord variants to minimize hand movement when playing a song? Which is the best programming approach to provide an optimal solution? To answer these questions, we perform an analysis of the situation depicted to find important information about the solution and the implications of using brute-force programming and dynamic programming. We also propose two implementations for these programming styles that provide a combination of chord variants to play a given song and report the overall hand displacement implied by said combination.

II. Methodology

To perform an analysis of the situation presented for the brute-force approach, we follow four steps: finding a vector representation of the solution, determining the solution space S, establishing the size of the space, and bounding the space using a constraint (S').

For the first step we use the vector σ (Figure 1). In this vector each σ_i represents a chord variant (a chord can have up to three chord variants to play it on the guitar, which is represented by numerals in Figure 1). Therefore, as the solution vector, σ provides a valid combination of chord variants that minimizes hand movement when playing a song.

$$\begin{array}{cccccc} \sigma = & \langle\sigma_1, & \sigma_2, & \sigma_3, & \ldots, & \sigma_n\rangle \\ & 1 & 1 & 1 & & 1 \\ & 2 & 2 & 2 & & 2 \\ & 3 & 3 & 3 & & 3 \end{array}$$

Figure 1. *Vector σ (sigma)*. Own design.

As for the solution space, it can be determined based on the chord variants assigned to σ_i in Figure 1, creating a T that contains them. Thus, we have that $\Gamma = \{1,2,3\}$ and $S = \Gamma^n$. Regarding the size of the space $|S|$, since each σ_i has 3 variants and there are n entries for σ, $|S|$ is equal to 3^n. S can be bounded by establishing S' as a space where there are not always 3 chord variants for every entry σ_i of the vector σ, which makes $|S'|$ smaller than 3^n.

Having this framework, we propose a brute-force algorithm implemented in Python (see Appendix A) that provides the best σ vector to play a given song. We expect it to have a behavior of O(3^n). Additionally, we test its efficiency by observing its behavior over time as it receives longer songs (an increasing number of chords).

To perform an analysis of the situation presented for the dynamic programming approach, we establish four elements: an oracle, an objective value, a base, and recursive steps that permit to build the oracle.

1. Oracle: Let F[i][j] be the minimum displacement to perform all chords j, j+1, …, n starting at variant I of the first chord j.

2. Objective value: the minimum between F[1][n], F[2][n], and F[3][n].

3. Base: F[1][0], F[2][0], F[3][0].

4. Recursive steps:

Let G be a matrix containing the variants chosen at each step F[i][j].

4.1. F[1][j] is equal to (previous centroid – current centroid)^2 added with the minimum between F[1][j+1], F[2][j+1], and F[3][j+1].

4.2. F[2][j+1] is equal to (previous centroid – current centroid)^2 added with the minimum between F[1][j+1], F[2][j+1], and F[3][j+1].

4.3. F[3][j+1] is equal to (previous centroid – current centroid)^2 added with the minimum between F[1][j+1], F[2][j+1], and F[3][j+1].

Having this framework, we propose an algorithm based on dynamic programming implemented in Python (see Appendix B) that provides the best path to play a given song, as well as the minimum displacement that corresponds to said path. We expect it to have a behavior of O(n). Additionally, we test its efficiency by observing its behavior over time as it receives longer songs (an increasing number of chords).

Taking into account the nature of the problem and the analysis developed both for the brute-force programming approach and the dynamic programming approach, we could not find a greedy solution that guarantees an optimal result. Therefore, we do not provide an implementation for this programming approach.

III. Results

As expected, the program was able to provide a valid combination of chord variants that can be used to play a song in a way that aims to minimize hand displacement from chord to chord. However, something worth noting is the fact that the algorithm becomes increasingly slow as the number of chords it receives becomes slightly bigger. Therefore, we were not able to test it with more than fifteen chords. Figure 2 shows that the behavior of the algorithm seems to match the expected complexity of O(3^n).

To have the same reference point for both algorithms, we tested the dynamic programming implementation with the same number of chords used for the brute-force implementation. Figure 3 shows that, for the most part, the behavior of the algorithm seems to match the expected complexity of O(n).

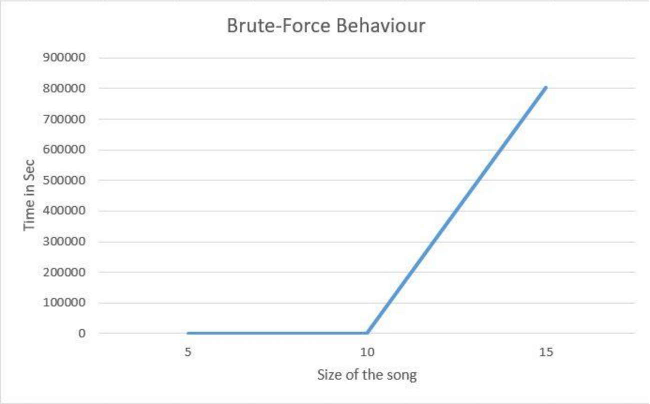

Figure 2. *Behavior of the proposed brute-force algorithm*. Own design.

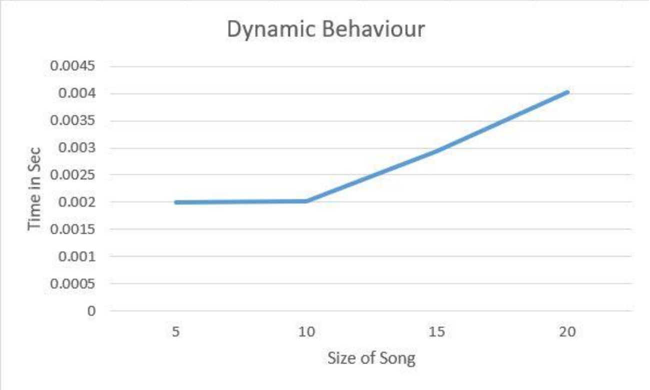

Figure 3. *Behavior of the proposed dynamic programming algorithm*. Own design.

IV. Conclusions

Based on the results obtained, we can assert that a brute-force algorithm can be used to address the situation presented and propose an optimal solution. Nevertheless, regardless of the accuracy of the combination of chord variants proposed and the optimization achieved by the algorithm in regards to hand displacement, it becomes clear that a brute-force programming approach is not the most efficient way to provide a solution for this situation. Even with slight changes to the number of chords it receives as input, the running-time of the algorithm drastically increases, which could be avoided by using a different, more efficient programming approach like dynamic programming. Dynamic programming offers an optimal solution in a significantly smaller wait period. It was not possible to find a greedy implementation that provides an optimal solution.

V. Bibliography

Kleinberg, J., & Tardos, E. (2005). *Algorithm Design*. Cornell University.

Appendix A: Code

The following algorithm shows the code used for the brute-force program proposed (note that the list *song* was modified for the different tests performed).

Algorithm 1: Brute-force programming approach

```
import time
import copy
import math
import pandas as pd

#function to find best path and calculate minimum
displacement
def bruteforce_stage(current_displacement,
min_displacement, curr_path, best_path, centroid_prev,
counter):

    #----- BASE CASE -----#
    if counter == len( song ) + 1:

        if current_displacement < min_displacement:

            min_displacement = current_displacement
            best_path = copy.deepcopy( curr_path )

        return min_displacement, best_path

    v1, v2, v3 = get_variant_centroids(song[counter - 1])

    #----- VARIANTS -----#
    for centroid in [ v1, v2, v3 ]:

        if counter != 1:
            #update displacement by considering the new chord
            temp_displacement = current_displacement
          temp_displacement += ( centroid_prev - centroid )**
2

        #add variant to current path
        if centroid == v1:
            curr_path[counter - 1] = 1
        elif centroid == v2:
            curr_path[counter - 1] = 2
        else:
            curr_path[counter - 1] = 3

        #save a copy of the current path
        curr_path_COPY = copy.deepcopy( curr_path )

        if counter != 1:
            min_displacement, best_path =
bruteforce_stage(temp_displacement, min_displacement,
curr_path_COPY, best_path, centroid, counter+1)
        else:
            min_displacement, best_path = bruteforce_stage(0,
min_displacement, curr_path_COPY, best_path, centroid,
counter+1)

    return min_displacement, best_path

#function to get the centroids of the variants of a given
chord
def get_variant_centroids(chord):

    #get hand placements
    placements = df.loc[chord].values.tolist( )
    placements = [ -1.0 if math.isnan( x ) else x for x in
placements ]
    variants = [] #list to save the centroids of the chord
variants

    for i in range(0, 20, 7):

        if placements[i] > 7: #variant with invalid fret
            fret = 10000000.0 #make it "infinity" (really big
number)
        elif placements[i] == -1: #variant does not exist
            fret = 0
        else:
            fret = placements[i]

        #variables for calculating the centroid
        addition = 0
        divisor = 6

        for x in range((i + 1), (i + 7)):
            if placements[x] == -1: #string is not played
                divisor -= 1
            else:
                addition += (fret + placements[x])
        if divisor != 0:
            variants.append(addition / divisor) #save centroid of the current variant
        else:
            variants.append(0) #signify that the variant is invalid

    #the 1st variant always exists
    if placements[8:14] == invalid: #check if 2nd variant exists
        variants[1] = 10000000.0
    if placements[15:20] == invalid: #check if 2nd variant exists
        variants[2] = 10000000.0

    return variants[0], variants[1], variants[2] #return variant centroids

#-------------------------------#

invalid = [ -1, -1, -1, -1, -1 ] #reference array to be used by get_chords function
song = ['E', 'Em7', 'A'] #list of chords to be analyzed

#save chord info in a pandas dataframe
df = pd.read_excel('GuitarDict_excel.xlsx')
columnNames = ["Chords"] + list(range(1, 22))
df = df.rename(columns = dict(zip(df.columns, columnNames)))
df.set_index("Chords", inplace = True)

#create lists to be used as inputs by function
path1 = [-1] * len(song)
path2 = [-1] * len(song)

#call brute-force function
start_time = time.time()
final_displacement, final_path = bruteforce_stage(0, float('inf'), path1, path2, 0, 1)
end_time = time.time()

print("Time taken: ", (end_time - start_time) * 10**3, 'milliseconds' )
print('\nMinimum displacement obtained = ', final_displacement)
print('\nPath:')
for j in range(len(song)):
    print(song[j], ' = play with variant ', final_path[j])
```

Algorithm 2: Dynamic programming approach

```
import pandas as pd
import math
import time

def get_centroids( chord ):

    invalid = [-1, -1, -1, -1, -1]

    #get finger positions and deal with NaN values
    placements = df.loc[chord].values.tolist()
    placements = [ -1.0 if math.isnan(x) else x for x in placements]

    #list to store centroids
    centroids = []

    for i in range(0, 20, 7):

        if placements[i] == -1: #originally NaN
            fret = 0
        else:
            fret = placements[i] - 1 #fix fret to change for it to work with finger values later

        sum = 0
        divisor = 6
        if placements[i] > 7: #invalid chord
            fret = 10000.0
            centroids.append(fret)
        else:
            for x in range( (i+1), (i+7) ): #go through chord variant
                if placements[x] == -1:
                    divisor -= 1
                else:
                    sum += ( fret + placements[x] )
            if divisor != 0:
                centroids.append(sum/divisor)
            else:
                centroids.append(0)

    #check for non-existent 2nd and 3rd variants
    if placements[ 8:13 ] == invalid:
        centroids[1] = 10000.0
    if placements[ 15:20 ] == invalid:
        centroids[2] = 10000.0

    return centroids[0], centroids[1], centroids[2]

def dynamic_stage():

    #displacement matrix (how much?, distance)
    F = [ [9 for a in range( len(song) )] for a in range(3) ] #second 3 should actually be len(chords)
    #variant matrix (how?, path)
    G = [ [9 for a in range( len(song) )] for a in range(3) ] #second 3 should actually be len(chords)

    #initial values for variant matrix
    F[0][0] = 0
    F[1][0] = 0
    F[2][0] = 0
    G[0][0] = 0
    G[1][0] = 1
    G[2][0] = 2

    for idx in range( 1, len(song) ):
        #idx = 1

        vA_1, vB_1, vC_1 = get_centroids( song[idx-1] ) #get centroids of the current chord
        vA_2, vB_2, vC_2 = get_centroids( song[idx] ) #get centroids of the next chord

        variants = { 0 : vA_1,
                     1 : vB_1,
                     2 : vC_1}

        #-------------------- VARIANT 1 --------------------#
        #calculate best displacements between vA, vB, and vC
        new_variant = G[0][idx - 1]
        new_variant = variants[ new_variant ]

        if(new_variant >= 10000.0 or F[0][idx-1]>=10000.0): #invalid path
            displacements = 10000.0
            min_displacement = 10000.0
            min_variant = 0
        else:
            displacements = [ (new_variant - vA_2)**2, (new_variant - vB_2)**2, (new_variant - vC_2)**2 ]
            min_displacement = min( displacements ) #get displacement
            min_variant = displacements.index( min_displacement ) #get variant

        #save values in F and G matrix
        F[0][idx] = F[0][idx-1] + min_displacement
        G[0][idx] = min_variant

        #-------------------- VARIANT 2 --------------------#
        #calculate best displacements between vA, vB, and vC
        new_variant = G[1][idx - 1]
        new_variant = variants[ new_variant ]
        if(new_variant >= 10000.0 or F[1][idx-1]>=10000.0): #invalid path
            displacements = 10000.0
            min_displacement = 10000.0
            min_variant = 1
        else:
            displacements = [ (new_variant - vA_2)**2, (new_variant - vB_2)**2, (new_variant - vC_2)**2 ]
            min_displacement = min( displacements ) #get displacement
            min_variant = displacements.index( min_displacement ) #get variant

        #save values in F and G matrix
        F[1][idx] = F[1][idx-1] + min_displacement
        G[1][idx] = min_variant

        #-------------------- VARIANT 3 --------------------#
        #calculate best displacements between vA, vB, and vC
        new_variant = G[2][idx - 1]
        new_variant = variants[ new_variant ]

        if(new_variant >= 10000.0 or F[2][idx-1]>=10000.0): #invalid path
            displacements = 10000.0
            min_displacement = 10000.0
            min_variant = 2
        else:
            displacements = [ (new_variant - vA_2)**2, (new_variant - vB_2)**2, (new_variant - vC_2)**2 ]
            min_displacement = min( displacements ) #get displacement
            min_variant = displacements.index( min_displacement ) #get variant

        #save values in F and G matrix
        F[2][idx] = F[2][idx-1] + min_displacement
        G[2][idx] = min_variant
    #--------------------#

    last_column = F[ 0 ][ len(song) - 1 ], F[ 1 ][ len(song) - 1 ], F[ 2 ][ len(song) - 1 ]
    final_min = min( last_column )

    return last_column.index(final_min), F, G, final_min

#----------------------------------------------------------#

song = ['Em','Em','Em7','A','Am','Em','Em7','A','Am','Em','A','Am7'
        ,'Am6','Em','Cmaj9','G','Cmaj9','G','Bm','Em7','Bm','Cmaj7','G','Em','Fmaj7sus2','Em'] #The last of Us

df = pd.read_excel('GuitarDict_excel.xlsx')

#include column names
column_names = ['Chords'] + list( range(1,22) )
df = df.rename ( columns = dict( zip(df.columns, column_names) ) )
df.set_index( 'Chords', inplace = True )

#function call
start = time.time()
chosen_row, Fmatrix, Gmatrix, min_displacement = dynamic_stage()
end = time.time()

print('Minimum displacement = ', min_displacement)
print("Best Path:")
count=0
```

```
for chord in song:
    print(chord,Gmatrix[chosen_row][count])
    count+=1

print('time = ', (end - start) * 10**3)

result = dynamic_stage()
```

Richard Andrés Rodríguez Montero is an outstanding, multilingual third-year Computer Science student from Texas Tech University-Costa Rica, recognized for his excellent academic performance and continuous hard work towards his scholastic, athletic, and artistic pursuits. He is enthusiastic about technology and its connection to humanity's progress, welfare, and happiness.

Sabrina Gabriela Guilarte Castillo is an accomplished, multilingual third-year Computer Science student at Texas Tech University – Costa Rica. She is passionate about technological innovation and its potential social contribution. In Spring 2021 she was nominated to "The Department of English Undergraduate Rhetoric and Writing Award" by Texas Tech University. In Spring 2022 she participated in the "Paving the way for Women in Technology" initiative, with Texas Tech University – Costa Rica and Americas Society/Council of the Americas. She currently works as Writing Consultant and Engineering Ethics Tutor at Texas Tech University – Costa Rica.