Minimize Hand Displacement in a Song: Project 2 - Brute Force and Dynamic Programming

1st Esteban Murillo
Department of Computer Science
Texas Tech University
Lubbock, TX
estmuril@ttu.edu

2nd Daniel Marin
Department of Computer Science
Texas Tech University
Lubbock, TX
danimari@ttu.edu

***Abstract*—This paper presents an algorithmic solution to optimize guitar chord transitions in musical sequences. We developed algorithms using both brute force and dynamic programming approaches to minimize left-hand displacement during chord changes. The system takes as input a sequence of chords in standard notation and a specification file containing multiple fingering positions for each chord. Our solution analyzes various possible chord positions and determines the optimal combination that minimizes the total hand movement throughout the song. The algorithms consider factors such as fret positions, string usage, and transitional distances between successive chord shapes. The results look to the difference and analyse the efficiency of the brute-force and dynamic programming approaches, based on the principal of demonstrating an effective method for reducing physical strain and improving playability in guitar performances through computational optimization.**

***Index Terms*—dynamic programming, optimization algorithms, guitar chord transitions, musical computing, computational musicology, fingering optimization, performance automation**

I. Introduction

The optimization of musical performance through computational methods has become increasingly relevant in modern music technology. In this paper, we address a specific challenge in guitar playing: minimizing the left-hand movement during chord transitions.

When playing guitar, the positioning of the left hand on the fretboard significantly impacts both the physical effort required and the smoothness of the performance. A single chord can often be played in multiple positions on the fretboard, and the choice of these positions directly affects the distance the hand must travel when transitioning between chords. For instance, a C major chord can be played in several configurations, each requiring different finger placements and fret positions.

The challenge lies in determining the optimal sequence of chord positions that minimizes the total hand movement throughout an entire song. This optimization must consider various factors:

- Multiple valid fingering positions for each chord
- The physical distance between successive chord positions
- The practical playability of the chosen sequences
- The specific requirements of open strings and unused strings in chord formations

To solve this problem, we developed two distinct algorithmic approaches. The first utilizes a brute force method, examining all possible combinations of chord positions to find the global optimum. The second employs dynamic programming techniques to efficiently compute the optimal solution by breaking down the problem into smaller subproblems and avoiding redundant calculations.

Our solution takes two inputs: a sequence of chords in standard musical notation (e.g., C, Am, Dm, G7) and a specification dictionary that details the various possible fingering positions for each chord. The dictionary uses a numerical representation system where each chord is defined by a sequence of numbers representing the fret positions for each string, with special notation for open strings (0) and unused strings.

II. Methodology

The Methodology of this project, is based on explaining the problem definition, each solution, and the comparisons between both. We also look to solve the following problem:

"Polynizer has hired the excellent students of CS-3364 to create an algorithm that allows to find the optimal way to play the chords that the application infers, on the guitar. Specifically, we want to minimize the amount of movement made by the left hand. The algorithm takes as input the list of chords of a song in standard notation (e.g., C, Am, Dm, G7, and C) and a file specifying different ways to play each chord, and should print on the screen how to play each chord in order to minimize the total movement of the hand over the song."[1]

A. Problem Outline

The algorithm we are looking to develop processes a song's chord sequence in standard notation (e.g., C, Am, Dm, G7) alongside a file detailing various fingerings for each chord, with the end goal of outputting the optimal sequence of chord fingerings that minimize the total hand movement throughout the song. Before understanding how the algorithm's solve this problem we first need to understand how solutions and inputs look like.

[1]Problem Definition stated in the project definition document by Arturo Camacho.

Both algorithms take in the same arguments. These arguments are as follows:

- Chord List: that represents the sequence of n chords in a song. This is what the user writes in a '.txt' file. It has the expected format of a list/set of type: $C = \{c_1, c_2, c_3, \ldots, c_n\}$ where the c_i element in the list represents the i^{th} chord of the song.
- Chord Dict: is a helper dictionary that contains all possible fingerings of each chord instance; considering their offsets. Thus providing a mapping from chord to fingerings available.

These inputs remain consistent to both the brute-force search and the dynamic programming solution. Also, understanding the format of the inputs is required for comprehending the algorithms.

Both algorithms return the same outputs, to answer (solve) the overall problem the user has. The outputs are:

- Optimal Solution: this outputs represents the minimum hand displacement calculated for the list of chords being displaced. Results are in units of 'frets displaced'.
- Optimal Vector: this output represents the vector $\sigma = \{\sigma_1, \sigma_2, \sigma_2, \ldots, \sigma_n\}$ where the σ_i element in this vector represents the fingering to be played in i^{th} chord of the song. Where $\sigma_i \in [0, K_i - 1]$, considering that K_i is the number of possible fingerings for the i^{th} chord.[2] The σ represents the overall sequence of fingerings that minimize the overall hand displacement.
- Optimal Sequence: this output represents the optimal vector as actual fingerings, it was used to debug the program.

The outputs represent the solution towards the problem that we look to solve. They provide the how and the value of the solution to the problem.

Understanding the inputs and outputs of the black box (the algorithms), we need to look at some overlapping functions that are used to calculate displacement based on what the professor stated in the project definition document.

B. Helper Functions

The functions in this section look to provide a simple method to calculate the overall displacement between two chords.

[2]A c_i with 3 fingerings has possible values $\sigma_i \in [0, 1, 2]$; where 0 stands for the 1^{st} fingering in the .csv file, 1 stands for the 2^{nd} fingering in the .csv file, and 2 stands for the 3^{rd} fingering in the .csv file; a c_i with 2 fingerings has possible values $\sigma_i \in [0, 1]$, and c_i with 1 fingering has possible values $\sigma_i \in [0]$.

```python
def calculate_average_fret(x):
    accum : int = 0
    frets : int = 0
    for fret in x:
        if fret is not None:
            accum += fret
            frets += 1
    return accum / frets if frets > 0 else 0.0
def calculate_movement_displacement(a, b):
    average_fret_a = calculate_average_fret(a)
    average_fret_b = calculate_average_fret(b)
    return (average_fret_a - average_fret_b)**2
```

Fig. 1. Uncommented functions in charge of calculating displacement of between fingerings, used in this project. Based, on program specifications in project definition document.

The functions in Figure 1 are used to measure the displacement of the left hand on a guitar when transitioning between chord positions. The `calculate_average_fret(x)` function calculates the centroid of a chord position by adding the fret numbers of all strings where the chord is fingering, and dividing it by the number of strings used by the fingering. For example:

$$\text{Centroid}_a = \frac{0+2+2+2+0}{5} = 1.2$$

$$\text{Centroid}_b = \frac{3+5+5+5+3}{5} = 4.2$$

Where Centroid_a represents chord A and Centroid_b represents chord C.

The `calculate_movement_displacement(a, b)` looks to calculate the displacement between two chords by calculating the following:

$$\text{Displacement} = (\text{Centroid}_a + \text{Centroid}_b)^2$$

These function help the algorithm's minimize the processes being called, by centralizing them. With this understood, we may begin explaining the brute-force search algorithm that solves this problem.

C. Brute-force Search

Brute-force search is a straightforward and exhaustive problem-solving technique that systematically explores all possible solutions to a problem to identify the optimal one. It works by generating every potential candidate in the solution space, evaluating each against a given objective or constraint, and selecting the best match. While simple to implement and guaranteed to find the correct solution, brute-force search is often computationally expensive, as the number of possibilities grows exponentially with the size of the input.

1) Algorithm Design: The algorithm designed for this project, regarding brute-force search, used exhaustive exploration of all possible sequences of fingerings from the chord list. There are various considerations, that should be looked at before evaluating the implementation of the algorithm: vector representation, its possible values, the space and its size.

The vector has the same representation from the Problem Outline, that being:

$$\sigma = \{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_n\}$$

Where $\sigma_i \in [0, K_i - 1]$, and where K_i represents the number of possible fingerings of the i^{th} chord in the song. If the resultant vector represents the solution, the vector space can be expressed as:

$$S = \{0, \frac{\sum_{i=1}^{n} K_i - 1}{n}\}^n$$

This setup of the vector space leads it to be upper bounded by $S = \{0, 1, 2\}^n$, considering that the worst case scenario is a song composed of chords with 3 fingerings available. The vector space size is:

$$|S| = 3^n$$

. However, considering that there is a constraint where we should only allow chord fingerings that have base fret position of 7 or less, the space S is further reduced to a subset S' defined as:

$$S' = \{\sigma \in S \mid \text{minimum fret position of } F(C_{\sigma_i}) \leq 7, \forall(i)\}$$

This leads to a reduction in the vector space by an incalculable amount but in theory it should decrease or maintain the same.

$$|S'| \leq |S|$$

2) Asymptotic Time Complexity: This algorithm has a time complexity of $\Theta(3^n)$ in the worst-case scenario, implying that this algorithm takes exponential time and is fairly inefficient for medium to large input sizes.

3) Code Implementation: The brute force algorithm is implemented through several key components:

```
def minimum_movement_brute_force(chord_list,
    chord_dict):
    chord_fingerings = []
    for chord in chord_list:
        if chord in chord_dict:
            fingerings = []
            fingerings.append(chord_dict[chord]['
    Finger A'])
            if 'Finger B' in chord_dict[chord]:
                fingerings.append(chord_dict[chord][
    'Finger B'])
            if 'Finger C' in chord_dict[chord]:
                fingerings.append(chord_dict[chord][
    'Finger C'])
            chord_fingerings.append(fingerings)
```

Fig. 2. First part of the brute force implementation: gathering possible fingerings

The first part of the implementation (Figure 2) focuses on gathering all possible fingerings for each chord in the sequence. The algorithm creates a list of available fingerings for each chord position, supporting up to three different fingering options (A, B, and C) per chord.

```
    for idx_combination in product(*[range(len(f))
    for f in chord_fingerings]):
        sequence = [chord_fingerings[i][
    idx_combination[i]]
                    for i in range(len(chord_list))]

        cost = total_movement_cost(sequence)

        if cost < min_cost:
            min_cost = cost
            min_cost_sequence = sequence
            min_cost_indices = list(idx_combination)
```

Fig. 3. Core loop of the brute force implementation: evaluating all combinations

The core of the algorithm (Figure 3) systematically evaluates every possible combination of fingerings. It uses Python's `itertools.product` to generate all possible combinations of fingering indices. For each combination:

- Creates a sequence of actual fingerings using the current combination
- Calculates the total movement cost for this sequence
- Updates the optimal solution if a lower cost is found

```
def total_movement_cost(sequence):
    cost = 0
    for i in range(1, len(sequence)):
        cost += calculate_movement_displacement(
    sequence[i-1], sequence[i])
    return cost
```

Fig. 4. Helper function for calculating total movement cost

The movement cost calculation (Figure 4) iterates through the sequence of chords, summing up the displacement cost between consecutive chord positions. The first chord position serves as the starting point and doesn't contribute to the total cost.

The bounded version of the algorithm adds an additional constraint check:

```
    if any(min(fret for fret in fingering if fret is
     not None) > 7
          for fingering in sequence):
        continue
```

This optimization ensures that only fingerings within the first seven frets are considered, which both reduces the search space and produces more practical solutions for guitar players.

D. Dynamic Programming

Dynamic Programming (DP) is an algorithmic paradigm that solves complex problems by breaking them down into simpler subproblems. It is applicable when:

- The problem can be broken down into overlapping subproblems
- The solution to the problem can be constructed from solutions of its subproblems
- The same subproblems are encountered multiple times when solving a larger problem

1) Oracle Definition: Let $C_{i,j}$:

- i represents the index of the current chord in the list
- j represents the index of the chosen fingering of the current chord, some may have one or more
- $C_{i,j}$ represents the minimum total movement cost from the first chord up to the i-th chord, assuming the i-th chord uses the j-th fingering

2) Recurrence Relation:

3) Algorithm Design:

4) Asymptotic Time Complexity:

5) Code Implementation:

E. Greedy Approach

A greedy algorithm was not implemented for this problem because a greedy approach cannot guarantee an optimal solution for chord sequence optimization. Here's why:

- A greedy algorithm would make locally optimal choices at each step, selecting the chord fingering that minimizes movement from the current position
- However, these locally optimal choices do not necessarily lead to a globally optimal solution
- Consider this example:
  - For a sequence "C-Am-G", a greedy approach might choose:
  - The closest fingering of Am to the initial C chord position
  - Then the closest G fingering to that Am position
  - This could result in a larger total movement than choosing a slightly "worse" initial transition that enables a much better final transition
- The problem requires considering the entire sequence of transitions to find the true optimal solution, which is why we implemented:
  - Brute force approach: Guarantees optimality by checking all possibilities
  - Dynamic programming approach: Efficiently finds the optimal solution by considering all possible combinations systematically

Therefore, this project focuses on the brute force and dynamic programming solutions as they are capable of finding the globally optimal solution to the chord sequence optimization problem.

III. TESTS

IV. RESULTS

V. CONCLUSION