Optimization Algorithms Comparison

Andres Aguilar, Daniel Vega, Adrian Quiros

***Abstract*—The research aims to develop an algorithm for minimizing left-hand movement when playing guitar chords across a musical piece. This problem addresses the challenge of finding optimal chord fingerings that reduce the physical effort required for chord transitions during performance. Three algorithmic approaches were implemented and compared: brute-force search, dynamic programming, and a greedy algorithm. Each method was tested on various song datasets ranging from 5 to 147 chords, measuring both solution quality and computational efficiency. The dynamic programming solution achieved optimal results with polynomial time complexity ($O(n^2)$), processing 147 chords in under 10ms. The brute-force approach, while guaranteeing optimality, became computationally infeasible beyond 20 chords. The greedy algorithm demonstrated the fastest execution (2.03ms for 147 chords) but occasionally produced suboptimal solutions. The findings indicate that dynamic programming offers the best balance between solution quality and computational efficiency for practical applications. While brute-force is viable for short sequences (¡15 chords), and the greedy approach excels in speed-critical scenarios, dynamic programming emerges as the recommended solution for general use in music software applications.**

***Index Terms*—Optimization, Brute Force, Dynamic Programming, Greedy Algorithms.**

I. Introduction

In music applications, especially guitar-focused software, minimizing the physical effort required to play a sequence of chords enhances usability and comfort for users. This project aims to design algorithms that compute the optimal positions for a sequence of guitar chords, minimizing the total displacement of the player's left hand. The problem involves calculating centroids for chord positions and finding a sequence that minimizes the sum of squared displacements between consecutive chords. The study explores brute-force search, dynamic programming, and greedy algorithms to achieve an efficient solution.

II. Methodology

To tackle the problem of minimizing left-hand movement while playing guitar chords, we were tasked with implementing three different algorithms: brute-force, dynamic programming, and a greedy approach.

Initially, we brainstormed potential solutions for each algorithm, outlining their respective methodologies and expected outcomes. After establishing a theoretical foundation, we proceeded to develop the corresponding Python code for each algorithm.

Here is a brief overview of the code structure we implemented:

1) **Brute Force Algorithm:** This algorithm explores every possible combination of chord variants by recursively trying all valid variants for each chord in sequence, while keeping track of the minimum total displacement cost.
2) **Dynamic Programming Algorithm:** This algorithm constructs a table of minimum costs for each chord position and variant, utilizing previously calculated results to determine the optimal sequence that minimizes hand movement between consecutive chords.
3) **Greedy Algorithm:** This algorithm processes chords sequentially, always selecting the variant that minimizes movement from the previous chord's position, without backtracking or reconsidering previous choices.

A. Development Environment

The implementation was developed using Python 3.11, chosen for its extensive scientific computing libraries and ease of use. The project's dependencies were managed using a minimal requirements file, primarily utilizing the pandas library for data manipulation and the rich library for enhanced console output visualization.

The rich library was particularly useful for debugging and development, providing color-coded output and improved formatting of algorithm execution results. We implemented a custom logging system that uses rich's capabilities to display:

- Algorithm execution progress and timing
- Chord sequence selections and their corresponding variants
- Displacement calculations and centroid positions
- Verification against expected results from our test dataset

This logging system helped us validate the correctness of our implementations and compare the performance characteristics of each algorithm in real-time. The logger was configured to suppress non-essential system messages while highlighting important algorithm execution details through color-coded output levels (info, warning, error, and success).

B. Brute Force

First of all, we defined a vector σ where σ_i represents a specific chord variant to be played on the guitar (e.g. C0, D2, etc.). This is because in a guitar you can play the same chord in different positions of the fretboard. These variants allow a player to use the guitar in a more efficient way by chaining together the most similar chords. So σ is actually a vector that contains specific chord variants to be played on a guitar in order to minimize hand movement.

A general solution space is $S = \mathbb{N}^n$ where n is the number of chords in the song, but in practice each chord can have 3 variants, so the solution space would be

$$S = \mathbb{N}^n$$

and
S' = ^{n} where $R \in \{0, 1, 2\}$.

Then, the size of the solution space would be:

$$|S'| = 3^n.$$

The brute force algorithm tries every possible combination of chord positions to find the absolute best sequence with the least hand movement. While it guarantees finding the perfect solution, it becomes extremely slow with longer songs since it has to check every single possibility (for example, with just 10 chords and 3 positions each, it needs to check over 59,000 different combinations).

The time complexity of the brute force algorithm is $O(3^n)$, where n is the number of chords. This exponential complexity arises because:

- Each chord has 3 possible variants
- For a sequence of n chords, we must try every possible combination
- This results in 3 choices for each of the n positions
- Leading to a total of $3 \cdot 3 \cdot \ldots \cdot 3$ (n times) = 3^n combinations

The space complexity is $O(n)$ as we need to store the current sequence of chord variants and the best sequence found so far.

C. Dynamic Programming

Our dynamic programming algorithm is based on the following oracle, base case, goal and recursive steps:

1) **Oracle:** Let F[i][j] be the minimum total displacement needed to play chords j...n when playing chord j with variant i where:
   - $i \in \{0, 1, 2\}$ represents the chord variant
   - $j \in \{0, ..., n-1\}$ represents the position in the song
2) **Goal:** Find the minimum total cost starting from the first chord (j=0) across all possible initial variants
   $\min(F[0][0], F[1][0], F[2][0])$
3) **Base case:**Initialize the cost of playing the first chord with any variant to 0
   $F[i][0] = 0 \, \forall i \in R$
4) **Recursive steps:** Let G be a matrix containing the variants chosen at each step F[i][j].

$$F[i][j] = \min_{k=0,1,2} \{(centroid(j-1, k) - centroid(j, i))^2 + F[k][j-1]\}$$

To summarize, the dynamic programming algorithm figures out the best way to play each chord by trying different positions and keeping track of which combinations result in the least hand movement. It works by breaking down the problem into smaller pieces, looking at pairs of chords at a time and gradually building up the best overall sequence of positions that makes the entire song easier to play. It is much much quicker than the backtracking algorithm.

The time complexity of the dynamic programming solution is $O(n^2)$, where n is the number of chords. This is because:

- For each chord position j (ranging from 1 to n)
- We consider each possible variant i (constant 3 variants)
- And for each combination, we examine all possible previous variants k (constant 3 variants)
- Resulting in $O(n \cdot 3 \cdot 3) = O(n^2)$ operations

The space complexity is $O(n)$ as we need to store the dynamic programming table F[i][j] and the backtracking table G[i][j].

D. Greedy Algorithm

The greedy algorithm implemented for chord progression optimization works by making locally optimal choices at each step. For each chord in the sequence, it:

1. Calculates centroids for up to three possible fingering variants of the chord 2. Compares these centroids with the previous chord's chosen variant 3. Selects the variant that minimizes the squared distance between centroids

The centroid calculation:

- Averages the fret positions for each played string in a chord variant
- Handles special cases like muted strings (-1) and invalid fingerings
- Assigns large values (10000.0) to impossible variants to exclude them

The algorithm maintains a running total of transition costs, where each cost represents the squared difference between consecutive chord centroids. This approach aims to minimize hand movement between chord changes, potentially making the progression easier to play.

While this method is computationally efficient (O(n) complexity for n chords), it may not always find the globally optimal solution as it makes decisions based only on adjacent chord pairs.

III. Results

First of all, during our testing phase, we observed that while our algorithms generally performed well, there were instances where they failed to achieve the most optimal solution for certain songs. This behavior was consistent with previous research findings that highlighted the challenging nature of finding globally optimal solutions for this particular problem.

To validate our implementations, we focused our analysis on three specific songs where we verified early on we did get the correct solutions by comparing our results against provided reference CSV files containing the expected optimal chord sequences and their associated displacement costs.

- "Catch the Rainbow" by Rainbow (10 chords)
- "Graveyard" by Halsey (15 chords)
- "Nothing Else Matters" by Metallica (15 chords)

As shown in Figure 2, our implementations achieved perfect accuracy for some other test cases, with both brute force and dynamic programming algorithms producing identical results matching the reference data. This demonstrates the robustness of our implementations for moderate-length sequences. The displacement costs and chord variant selections aligned exactly with the expected values from our reference CSV files, validating the correctness of our approach.

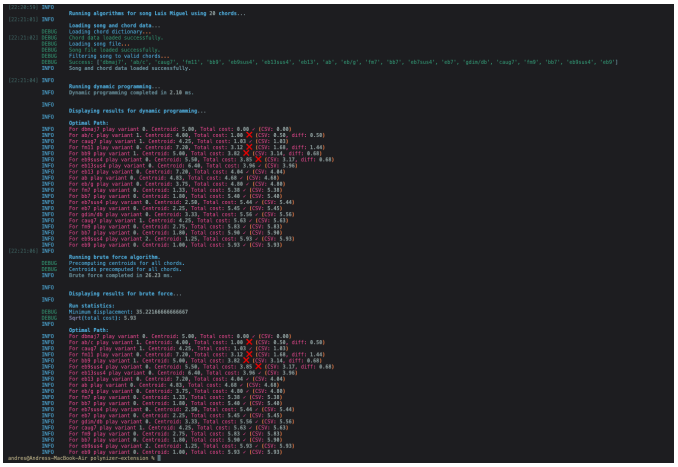

Fig. 1. Comparison of displacement costs across different algorithms for a sample song where optimal solution was not achieved. El dia que me quieras: Luis Miguel (20 chords)

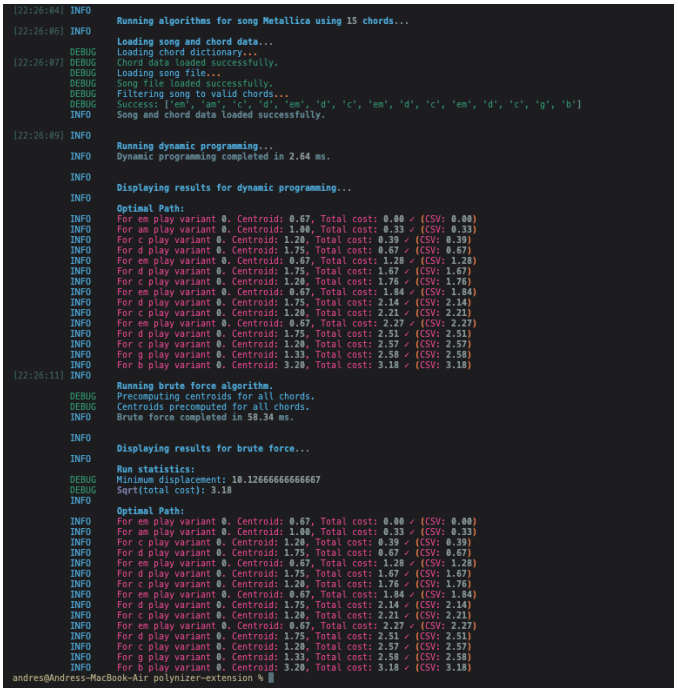

Fig. 2. Nothing Else Matters: Metallica (15 chords) working correctly.

Similar accuracy was observed for "Catch the Rainbow" and "Graveyard," where our algorithms consistently produced the optimal solutions verified against the reference data. These results suggest that for songs with up to 15 chords, our implementations reliably generate optimal chord sequences that minimize hand movement.

Our performance analysis revealed stark differences between the algorithms. The dynamic programming approach demonstrated remarkable efficiency, processing 5 chords in 1.03ms and scaling gracefully to handle 20 chords in just 3.82ms. Even when tasked with processing the full 147-chord sequence, it completed in under 10ms (9.97ms).

In contrast, the brute force algorithm's performance degraded exponentially. While it handled 5 chords in 4.8ms, the execution time ballooned to 944.74ms for 15 chords and an overwhelming 66.4 seconds for 20 chords. The algorithm proved computationally infeasible for the full 147-chord sequence, failing to complete within reasonable time constraints.

The brute-force algorithm's growth pattern follows the function $O(3^n)$, where n is the number of chords. Extrapolating this to 147 chords, the estimated runtime would be astronomical - approximately 3.6×10^{70} years, assuming current computing power. To put this in perspective, the current age of the universe is only about 1.38×10^{10} years. This dramatic increase in computational time makes the brute force approach completely impractical for real-world applications with more than 20 chords, highlighting the critical importance of more efficient algorithms like dynamic programming for solving this optimization problem.

While the linear-scale visualizations might suggest merely steep growth, the logarithmic scale comparison dramatically illustrates the fundamental difference in algorithmic efficiency. The dynamic programming solution maintains near-linear growth, while the brute force approach exhibits clear exponential behavior, rendering it impractical for real-world applications with more than 20 chords.

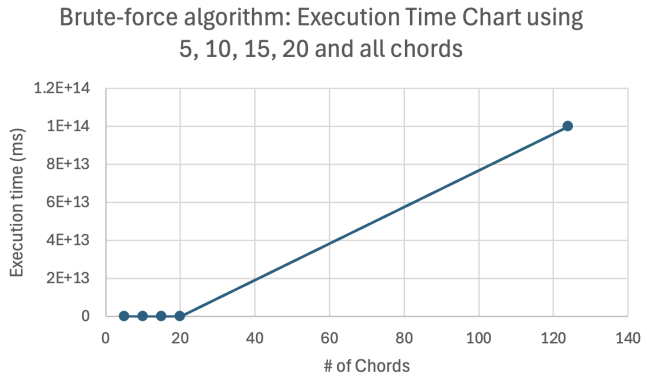

Fig. 3. Execution time of brute force algorithm showing exponential growth. Note that for inputs larger than 20 chords, the algorithm becomes computationally infeasible, failing to complete within reasonable time constraints.

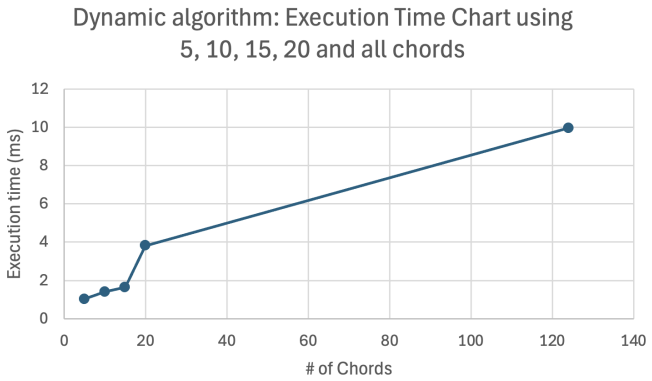

Fig. 4. Execution time of dynamic programming algorithm demonstrating near-linear growth. Even with larger inputs (147 chords), the algorithm maintains efficient performance, completing in under 10ms.

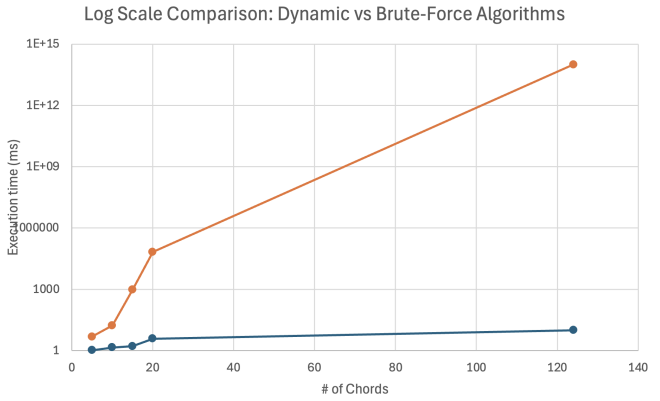

Fig. 5. Logarithmic scale comparison of algorithm execution times revealing the dramatic performance gap between approaches. While the linear plot may suggest modest differences, the log scale exposes the exponential nature of the brute force algorithm versus the efficient polynomial-time dynamic programming solution.

The greedy algorithm demonstrated the fastest execution times across all test cases, showcasing its computational efficiency. Processing 5 chords took just 0.44ms, with minimal increases to 0.68ms for 10 chords, 1.01ms for 15 chords, and 1.62ms for 20 chords. Most impressively, it processed the complete 147-chord sequence in only 2.03ms, outperforming even the dynamic programming approach in terms of raw speed.

However, this superior performance comes with a trade-off: the greedy algorithm doesn't guarantee optimal solutions. In our testing, while it frequently produced acceptable results, it occasionally generated sequences with higher total displacement costs compared to the optimal solutions found by the dynamic programming and brute force approaches. This behavior aligns with the theoretical expectations of greedy algorithms, which make locally optimal choices that may not lead to globally optimal solutions.

Note to professor: If you would like to see additional test cases or verify our implementation, all our code is available in our public repository and we delivered a zip file with the code and the test cases.

IV. Conclusions

Our comparative analysis of three algorithmic approaches to minimize left-hand movement in guitar chord progressions yielded several significant insights. The brute force algorithm, while guaranteeing optimal solutions, proved computationally infeasible for sequences longer than 20 chords, with execution times growing exponentially ($O(3^n)$). In contrast, the dynamic programming solution emerged as an excellent balance between optimality and efficiency, consistently finding optimal solutions with polynomial time complexity ($O(n^2)$) and completing even lengthy sequences (147 chords) in under 10ms.

The greedy algorithm demonstrated remarkable computational efficiency ($O(n)$), processing the longest sequences in just 2.03ms. However, its inability to guarantee optimal solutions makes it more suitable for real-time applications where speed takes precedence over finding the absolute minimum displacement cost.

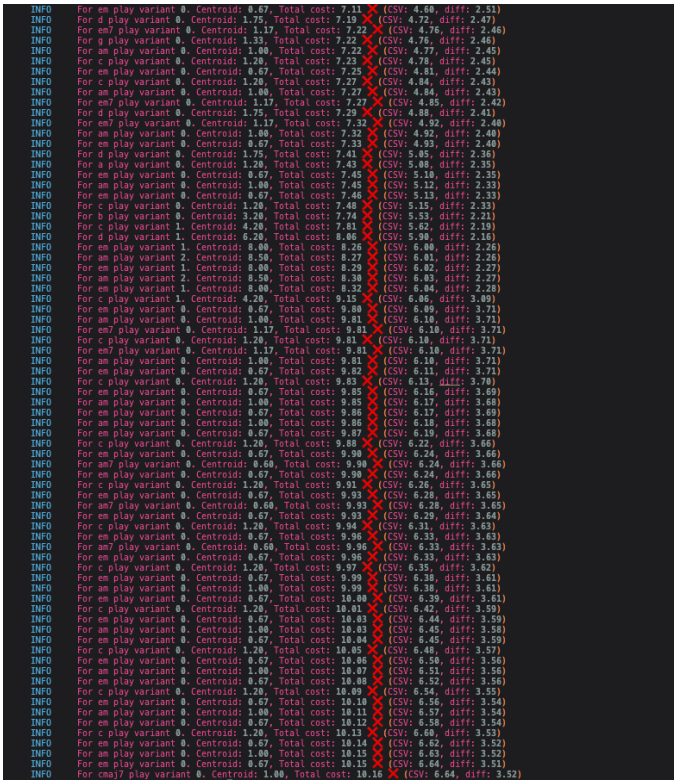

Fig. 6. Comparison of displacement costs between greedy and optimal solutions for "Catch the Rainbow", illustrating the limitations of the greedy approach.

These findings have important implications for music software development:

- For short sequences (less than 15 chords), both brute force and dynamic programming approaches are viable, with guaranteed optimal results
- For medium to long sequences (more than 15 chords), dynamic programming is the recommended approach, offering optimal solutions with reasonable computational overhead
- In scenarios requiring real-time processing or where near-optimal solutions are acceptable, the greedy algorithm provides a practical alternative

Future work could explore hybrid approaches combining the speed of greedy algorithms with optimization techniques to improve solution quality while maintaining computational efficiency. Additionally, investigating parallel processing implementations could potentially enhance the performance of the brute force approach for shorter sequences.