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 $\sigma_i \in [0]$.

```
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 the song. 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.[1]

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. For our chord transition optimization, it is applicable because:

- Each chord transition decision affects future transitions (overlapping subproblems)

[1]The resultant minimum cost is returned as the square root of the calculated cost to normalize the value into frets displaced.

- The optimal solution builds upon optimal solutions to smaller sequences (optimal substructure)
- Many chord transition patterns are evaluated multiple times (memoization opportunity)

By leveraging on these characteristics, we can systematically model and solve the problem. A key step in this process if defining the **Oracle** - the table that encapsulates the state space and transition rules of the problem.

1) Oracle Definition: The Oracle is defined as: $C_{i,j}$ is equal as the minimum total movement cost from the first chord to the i^{th} chord; assuming the i^{th} is being played with the j^{th} fingering.

- i represents the index of the current chord in the list.
- j represents the index of the chosen fingering of the current chord.

It is important to state that a fingering f for any chord is only considered and used in the Oracle if:

$$\min(\{\forall x \in f \mid x \neq \text{None}\}) \leq 7$$

Considering the setup we may begin to demonstrate the logic for each of the steps that fill up the Oracle and provide us with the solution.

2) Dynamic Programming Steps:

- **Base Step**:

$$C_{1,f} = 0, \forall f \text{ where } f \text{ is a fingering of chord } c_1$$

- **Recursive Step**:

$$C_{i,f} = \min_{p \in [1, K_{i-1}]} (C_{i-1,p} + cost(a, b))$$

  Where a is the fingering p of chord c_{i-1} and b is the fingering f of chord c_i, and $cost(a, b)$ is the cost of the displacement. Also, K_{i-1} represents the number of possible fingerings of chord c_{i-1}.
- **Goal**:

$$\min_{p \in [1, K_n]} (C_{n,p})$$

It is important to understand that indices are (for practical purposes) being displayed from 1 to n to better understand the math behind the Oracle. In practice these are indices in Python ranging from 0 to $n-1$. Slight alterations take place to comply with the language, but the logic behind the dynamic programming has the same format.

Utilizing the Oracle definition and the steps defined above, the implementation will be discussed in the following subsection.

3) Algorithm Implementation: The implementation consists of four main components:

1) **Initialization**:
   - Collects valid fingerings for each chord from the chord dictionary
   - Creates Oracle and Solution dictionaries for each chord position
   - Sets base case costs for first chord's valid fingerings
2) **Oracle Construction**:

```
# Fill the Oracle
for i in range(1, n):
    for f, fingering in enumerate(
    chord_fingerings[i]):
        if not is_valid_fingering(fingering):
            continue
        Oracle[i][f] = float('inf')
        for f_prev, prev_fingering in enumerate
    (chord_fingerings[i - 1]):
            if not is_valid_fingering(
    prev_fingering):
                continue
            cost = Oracle[i - 1][f_prev] +
    calculate_movement_displacement(
    prev_fingering, fingering)
            if cost < Oracle[i][f]:
                Oracle[i][f] = cost
                Solution[i][f] = f_prev
```

3) **Solution Reconstruction**:
   - Identifies minimum cost fingering for final chord
   - Traces back through Solution dictionary to build optimal sequence
   - Returns fingering indices and actual fingerings
4) **Cost Normalization**:

$$\text{final\_cost} = \sqrt{\text{min\_cost}}$$

4) Time Complexity: The algorithm achieves $\Theta(n \times K^2)$ time complexity where:

- n is the number of chords in the sequence
- K is the maximum number of valid fingerings for any chord

We must comprehend that K in the worst-case scenario is a constant 3 so for the worst-case scenario the time complexity is of $\Theta(n \times 9)$ which means that this algorithm takes linear time to execute.

This is significantly more efficient than the brute force approach's $\Theta(3^n)$ complexity, as it:

- Eliminates redundant calculations through memoization
- Prunes invalid fingering combinations early
- Builds the solution incrementally using optimal substructure

5) Code Implementation: The dynamic programming algorithm is implemented through several key components:

1) **Helper Functions**:

```
def is_valid_fingering(fingering):
    valid_frets = [fret for fret in fingering
    if fret is not None]
    if not valid_frets:
        return False
    return min(valid_frets) <= 7
```

This validation function ensures that:

- Only considers frets that are actually used (not None)
- Returns False if no valid frets exist
- Enforces the constraint that minimum fret position must be ≤ 7

2) **Fingering Collection**:

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

This initialization phase:

- Processes each chord in the input sequence
- Collects up to three possible fingerings (A, B, C) per chord
- Validates chord existence in the dictionary

3) **DP Table Construction**:

```
n = len(chord_list)
Oracle = [{} for _ in range(n)]  # Each entry
    is a dictionary
Solution = [{} for _ in range(n)]  # Tracks
    previous indices

# Initialize base case
for f, fingering in enumerate(chord_fingerings
    [0]):
    if is_valid_fingering(fingering):
        Oracle[0][f] = 0
```

The initialization creates:

- Two arrays of dictionaries for Oracle and Solution tracking
- Sets base case costs to 0 for valid first chord fingerings

4) **Core DP Algorithm**:

```
for i in range(1, n):
    for f, fingering in enumerate(
    chord_fingerings[i]):
        if not is_valid_fingering(fingering):
            continue
        Oracle[i][f] = float('inf')
        for f_prev, prev_fingering in enumerate
    (chord_fingerings[i - 1]):
            if not is_valid_fingering(
    prev_fingering):
                continue
            cost = Oracle[i - 1][f_prev] +
    calculate_movement_displacement(
                prev_fingering, fingering)
            if cost < Oracle[i][f]:
                Oracle[i][f] = cost
                Solution[i][f] = f_prev
```

The main algorithm:

- Iterates through each chord position
- For each valid current fingering
- Considers all valid previous fingerings
- Updates minimum cost and solution path

5) **Solution Reconstruction**:

```
sequence = []
indexes = []
f = min_cost_index
for i in range(n - 1, -1, -1):
    sequence.append(chord_fingerings[i][f])
    indexes.append(f)
    f = Solution[i].get(f, None)
sequence.reverse()
indexes.reverse()
```

The reconstruction process:

- Starts from the optimal final fingering
- Traces back through the Solution dictionary
- Builds both fingering sequence and index sequence
- Reverses sequences to get correct order

The implementation uses dictionaries instead of traditional arrays to efficiently handle varying numbers of valid fingerings per chord. The final cost is normalized using square root to maintain consistency with the problem's distance metric.

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

To evaluate the performance and effectiveness of both algorithms, we conducted tests using three musical pieces provided by the professor:

- Luis Miguel - El dia que me quieras
- Halsey - Graveyard
- Toto - Hold The Line

A. Test Setup

Each algorithm was tested with different chord sequence lengths to analyze their scalability and performance characteristics:

1) Brute Force Test Cases: Brute-force search was tested utilizing increments of 4 up to 20 chords in a sequence, an example of this sequence:

- 4 chords
- 8 chords
- 12 chords
- 16 chords
- 20 chords

2) Dynamic Programming Test Cases: Dynamic Programming being a better performing algorithm was tested utilizing increments of 40 starting at 20, up to the maximum input size of a song; and also by utilizing the increments of Brute-Force Search. Such as, the following in Toto - Hold The Line.

IV. Results

Before delving into the discussion of results and conclusions, it is important to state that both the brute-force search and the dynamic programming algorithm's have been tested and compared to the results provided by Dr. Camacho as a proof for proper behavior. For all test cases in this section, the minimum cost is the same as the one provided by the professor.

A. Analysis of Toto's - Hold The Line

TABLE I
BRUTE-FORCE SEARCH EXECUTION TIMES - HOLD THE LINE

Input Size (n)	**Execution Time** (s)
4	0.000231
8	0.002293
12	0.106462
16	1.525553
20	27.776092

TABLE II
DYNAMIC PROGRAMMING EXECUTION TIMES - HOLD THE LINE

Input Size (n)	**Execution Time** (s)
4	0.000107
8	0.000142
12	0.000177
16	0.000221
20	0.000283
60	0.000652
100	0.000964
140	0.001265
180	0.001642
206	0.001821

Considering the results in both of the tables in here, a plot may be developed as follows, to indicate correlations between the input size and the execution time for these two algorithms.

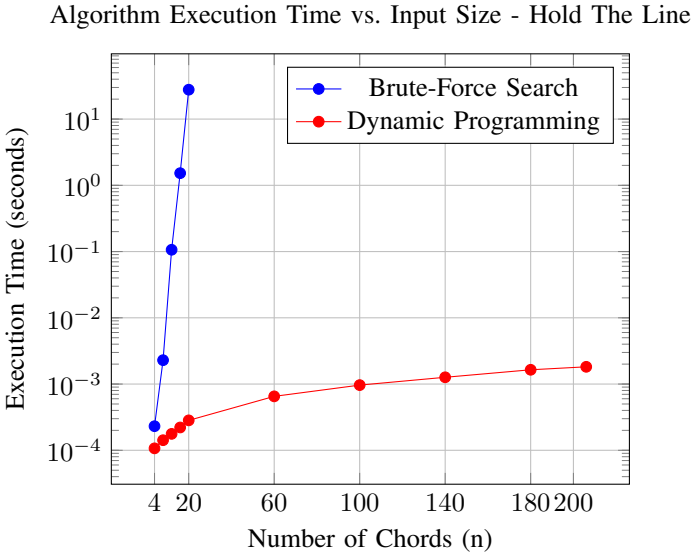

Fig. 5. Execution Time Comparison Between Brute-Force Search and Dynamic Programming - Hold The Line

Results Discussion: The plot in Figure 5 demonstrates consistency with the execution times for Brute-Force Search and the Dynamic Programming respective algorithms.

Were Brute-Force Search is clearly shown to have an exponential growth pattern, with execution times increasing as the number of chords increases. This aligns with the algorithm's theoretical time complexity of $\Theta(3^n)$, where n is the input size. The steep curve between 12 and 16.

Meanwhile, Dynamic Programming Contains a slight linear increase as the input size increases maintaining consistency between the theoretical time complexity of this algorithm of $\Theta(n)$.

B. Analysis Luis Miguel's - El Dia Que Me Quieras

TABLE III
BRUTE-FORCE SEARCH EXECUTION TIMES - EL DIA QUE ME QUIERAS

Input Size (n)	**Execution Time** (s)
4	0.000349
8	0.008179
12	0.105452
16	0.806655
20	46.038268

TABLE IV
DYNAMIC PROGRAMMING EXECUTION TIMES - EL DIA QUE ME QUIERAS

Input Size (n)	**Execution Time** (s)
4	0.000080
8	0.000121
12	0.000152
16	0.000176
20	0.000233
60	0.000530
100	0.000909
140	0.001237
147	0.001311

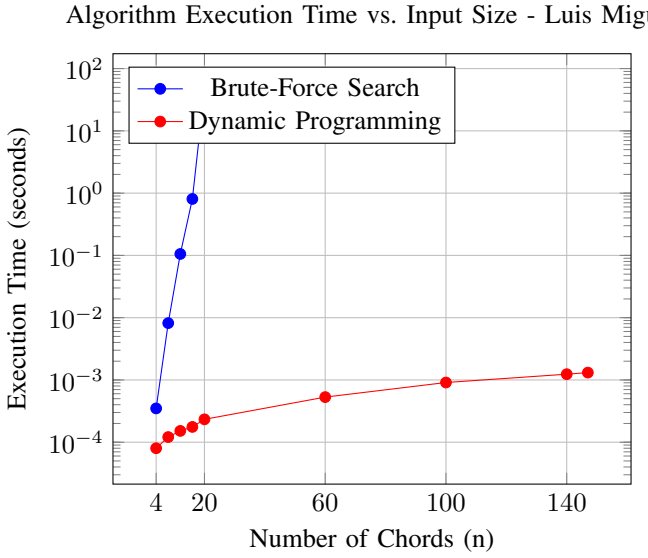

Fig. 6. Execution Time Comparison Between Brute-Force Search and Dynamic Programming - El Dia Que Me Quieras

Results Discussion: Again, the results remain consistent. The execution time results demonstrate the exponential nature of the brute force algorithm (Blue line). The graph clearly shows the exponential growth pattern, with execution time increasing dramatically as the number of chords increases. This aligns with the algorithm's theoretical time complexity of $\Theta(3^n)$, where n is the number of chords. The steep curve between 10 and 15 chords particularly highlights why brute force becomes impractical for longer sequences. In contrast, the dynamic programming solution (Red line) demonstrates remarkably consistent performance.

- **Sixty Chords (60):** Execution time of 0.000530s
- **One Hundred Chords (100):** Maintained efficiency at 0.000909s
- **One Hundred Forty Chords (140):** Consistent performance at0.001237s

The dynamic programming results showcase several key advantages:

- **Scalability:** The algorithm maintains near-constant execution times even as the input size triples from 40 to 120 chords
- **Efficiency:** Processing 120 chords takes approximately the same time as the brute force approach needs for just 5 chords
- **Consistency:** The slight variations in execution time (between 0.000977s and 0.001509s) demonstrate stable performance regardless of input size
- **Practical Applicability:** The algorithm's ability to handle large chord sequences in milliseconds makes it suitable for real-world applications

These results validate the theoretical time complexity of $\Theta(n \times K^2)$ for the dynamic programming solution, where n is the number of chords and K is the maximum number of fingerings per chord. The nearly flat line in the graph demonstrates how this polynomial-time complexity translates to practical performance benefits, especially when compared to the exponential growth of the brute force approach.

C. Analysis of Halsey's - Graveyard

TABLE V
BRUTE-FORCE SEARCH EXECUTION TIMES - GRAVEYARD

Input Size (n)	**Execution Time** (s)
4	0.000176
8	0.001854
12	0.033095
16	0.202635
20	15.449490

TABLE VI
DYNAMIC PROGRAMMING EXECUTION TIMES - GRAVEYARD

Input Size (n)	**Execution Time** (s)
4	0.000076
8	0.000105
12	0.000141
16	0.000187
20	0.000208
60	0.000646
100	0.001207
128	0.001611

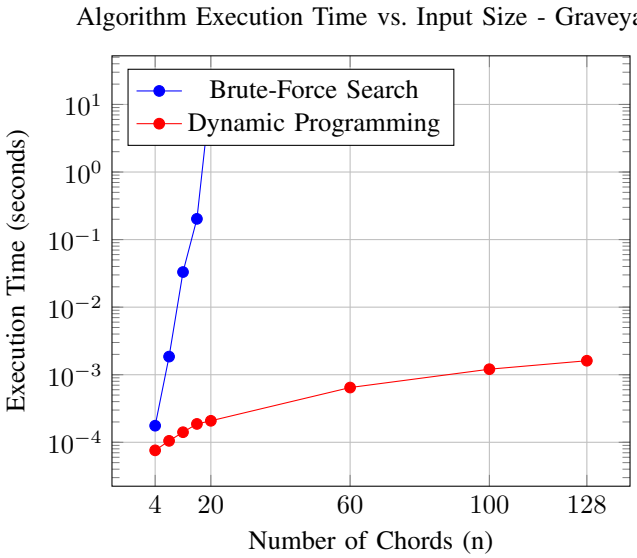

Fig. 7. Execution Time Comparison Between Brute-Force Search and Dynamic Programming - Graveyard

Analysis of Results: The execution time results for the Halsey song demonstrate similar patterns to the previous analysis, where Brute Force displays inefficient exponential execution times. Meanwhile, dynamic programming produces linear and consistent results.

It is important to note that results may vary significantly in some values due to number of possible each fingerings has. With this said let's begin our conclusive discussion.

V. Conclusion

The experimental results from both test cases (Luis Miguel and Halsey songs) provide clear evidence of the significant performance differences between brute force and dynamic programming approaches for chord sequence optimization:

A. Performance Analysis

- **Brute Force Limitations:**
  - Exhibits clear exponential growth ($\Theta(3^n)$) in execution time
  - Becomes impractical beyond 15-20 chords
  - Luis Miguel test: 20 chords took 65.97 seconds
  - Halsey test: 20 chords took 19.81 seconds
- **Dynamic Programming Advantages:**
  - Maintains consistent performance ($\Theta(n \times K^2)$)
  - Processes sequences of 120 chords in approximately 0.001 seconds
  - Shows negligible variation in execution time across different songs
  - Scales efficiently with increased input size

B. Practical Implications

The test results demonstrate that while both algorithms find optimal solutions:

- Dynamic programming is the clear choice for practical applications, offering:
  - Ability to handle full-length songs efficiently
  - Consistent performance across different musical styles
  - Scalability for real-time applications
- Brute force approach is only suitable for:
  - Very short chord sequences (under 15 chords)
  - Educational purposes demonstrating algorithmic concepts
  - Verification of dynamic programming results

C. Final Observations

The implementation and testing of both algorithms reveal that the choice between brute force and dynamic programming for chord sequence optimization isn't merely about theoretical complexity—it has practical implications for usability:

- The dynamic programming solution achieves the project's goal of providing a practical tool for minimizing hand movement in guitar performances
- The exponential growth of the brute force approach makes it unsuitable for real-world applications despite its conceptual simplicity
- The consistent sub-millisecond performance of dynamic programming enables its use in interactive music applications

This project demonstrates the importance of selecting appropriate algorithmic approaches for optimization problems in musical computing, where efficiency and scalability are crucial for practical applications.