A comparative analysis of A, Greedy, and Bruteforce algorithms for automatic chord progression generation*

Hakam Avicena Mustain - 13524075
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: hakammustain@gmail.com , 13524075@std.stei.itb.ac.id

***Abstract*—Beginner musicians often struggle to find chord progressions that fit a given melody, as this skill typically requires years of theoretical training. This paper presents an automatic chord progression generator that models harmonic accompaniment as a graph search problem over a state space of chord assignments, where each state represents a chord assigned to a melody phrase extracted from a MIDI file. Three search algorithms are implemented and compared using a cost function combining circle-of-fifths harmonic distance, duration-weighted melodic compatibility, and voice-leading smoothness, guided by an admissible heuristic based on tonal function. Experiments on four MIDI melodies of varying complexity show that A* consistently matches Brute Force's optimal solutions wherever the latter is computationally feasible, while exploring up to 16,000 times fewer states. Compared to Greedy, A* produces chord progressions with up to 19.5% lower harmonic cost, with the quality gap widening as melody complexity increases. These results demonstrate that informed search provides a practical balance between solution optimality and computational efficiency for automatic harmonization, offering beginner musicians a theory-grounded tool for generating playable chord progressions directly from melody.**

Keywords—chord progression generation; A* search; graph search; music theory; pathfinding

I. Introduction (*Heading 1*)

Finding the right chord progression is one of the most challenging aspects for beginner musicians. When listening to a song for the first time, a casual musician often struggles to identify which chords fit the melody. Chord progression serves as the harmonic backbone of a song, guiding the listener for the entire song from the opening phrase to a final resolution. Without a chord progression, even a simple melody can sound disorganized and unpleasant to hear.

Developing an ear for chord selection typically requires years of practice and a solid grounding in music theory. However, applying these rules manually requires deep theoretical knowledge that is not always accessible for beginners. There is a gap here, a musician may have a melody in mind but lack the theoretical foundation to harmonize in a structured way.

This paper proposes an automatic chord progression generator that takes a melody as an input and produces a chord progression guide as output, built by music theory principles encoded as cost functions. The system is built upon a graph search algorithm, treating chord selection as a pathfinding problem over a state space of possible harmonics. Three algorithms are implemented and compared, Brute Force (exhaustively explores all possible progressions), Greedy Best First Search (greedily selects the locally best chord at each step), and A* (balances cumulative harmonic cost with an estimated future cost to find the globally optimal progression). This paper aims to demonstrate not only the practical utility of algorithmic chord generation for beginner musicians but also trade-offs in solution quality and computational efficiency among different search strategies.

II. Theoretical framework

A. Pathfinding and search space

Pathfiding is the process of finding a path from a start state toward a goal state within a defined search space. Search space can take various forms, including graphs, grids, networks, or any other representation that models possible transitions from one state to another. Pathfinding aims to find a sequence of steps that connects the initial state to the goal state. This sequence is referred to as a path. This path is valid if the goal state is reached in sufficient way, also the path must satisfy additional criteria such as minimum cost, shortest distance, or fewer steps[1].

The chord progression generation problem is modeled as a pathfinding problem over a harmonic search space. The path navigates a space of harmonic choices where each state represents a chord assigned to a particular melody phrase and each transition represents the act of moving from one chord to the next consecutive phrase. The search space in this problem is defined by the following components:

- Initial state: A virtual start node before the first melody phrase, with no chord assigned.
- Goal state: A complete chord assignment for all n melody phrases, preferably ending on the tonic chord to achieve harmonic resolution.

- Actions: From state (i, c_i), the system may transition to any state (i+1, c□) where c□ is a harmonically compatible candidate chord for phrase p_{i+1}.
- Transition model: Each action produces a new state (i+1, c□) representing the selection of chord c□ for the next phrase.
- Path cost: The cumulative harmonic cost accumulated along the path, measured by the transition cost function defined in Chapter 3.

Since each transition between chords carries a cost reflecting harmonic distance and voice leading smoothness, this problem is modeled as a weighted graph search. The goal of pathfinding here is therefore not merely to reach any valid complete chord assignment, but to find the path of minimum total cost the chord progression that sounds most natural and musically satisfying.

B. Basic Music Theory

1) Major Scale and Diatonic Chords

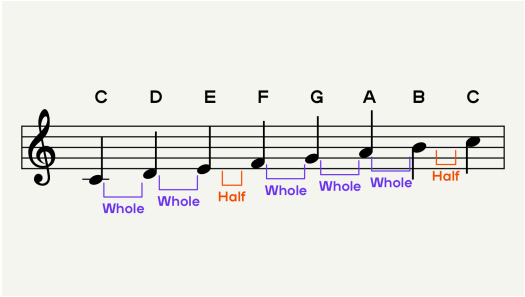

Fig. 1 Major scale in C

A musical state is an ordered sequence of pitches that serves as the tonal foundation for melody and harmony. Major scale is one of the most commonly used scales which consists of seven distinct pitches arranged to a fixed pattern of whole steps (W) and half steps (H): W-W-H-W-W-W-H.

From each degree of the major scale, a diatonic chord can be constructed by stacking notes in intervals of thirds using only the notes within that scale. Diatonic chords are chords built entirely from the notes of a specific scale, whether major or minor, and they form the foundation of harmony and chord progressions within a given key. In the key of C major, the seven diatonic chords are:

Table 2.1 — Constructed diatonic chord from C major

Scale Degree	Roman Numeral	Chord	Quality
1st	I	C major	major
2nd	ii	D minor	minor
3rd	iii	E minor	minor
4th	IV	F major	major
5th	V	G major	major
6th	vi	A minor	minor
7th	vii°	B diminished	diminished

The pattern of chord qualities (major-minor-minor-major-major-minor-diminished) is universal for all 12 major keys which are the same structural relationships held regardless of the root note.

2) Circle of Fifths and Harmonic Distance

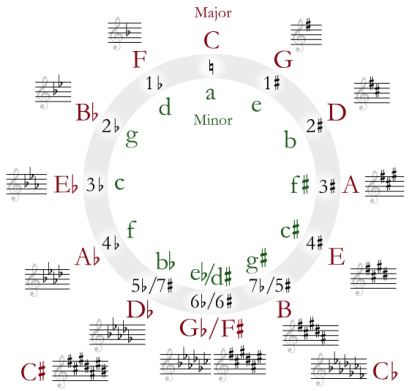

Fig. 2 Circle of Fifths

The circle of fifths is an arrangement of the 12 musical notes of the chromatic scale, each a fifth apart, organized in the shape of a circle. It reveals important relationships between pitches and organizes them in a way that is useful for understanding diatonic harmony.Chord progressions often move between chords whose roots are related by perfect fifth, making this concept useful in illustrating harmonic distance between chords. For example C and G or C and F are considered harmonically close to produce smooth, natural-sounding transitions.

This concept of harmonic distance is central to the cost function used in this paper. Given two consecutive chords in a progression, their distance on the circle of fifths provides a quantifiable measure of how "natural" the transition between them sounds. A distance of 1 step (e.g., C→G) represents a very smooth transition, while a distance of 5 or 6 steps represents a more jarring one. Formally, the harmonic distance between two chords with roots r_1 and r_2 is defined as:

$$d(r_1, r_2) = \min(|pos(r_1) - pos(r_2)|, 12 - |pos(r_1) - pos(r_2)|)$$

where pos(x) denotes the position of note x on the circle of fifths (0–11).

3) Harmonic Function (Tonal Function)

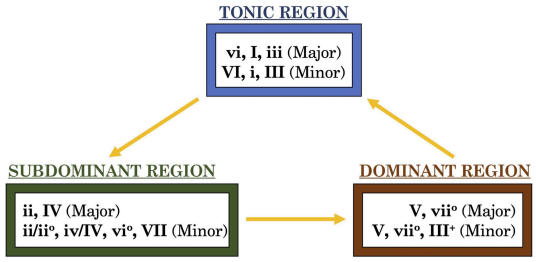

Fig. 3 Tonal Function

In music, harmonic function (or tonal function) denotes the relationship of a chord or scale degree to a tonal center. There are three abstract tonal functions; tonic, dominant, and subdominant; each of them could take on a more or less modified appearance in any chord of scale. The tonic is the central chord of a key, representing the tonal center and providing a sense of resolution and stability. The dominant is the fifth chord in the key, which creates tension and naturally resolves back to the tonic, making it a crucial element in creating movement and direction in music. The subdominant, the fourth chord in the key, serves as a complementary force to the dominant, often acting as a transitional or preparatory chord that leads smoothly into the dominant or back to the tonic. The progression of harmonic functions in music is cyclical in nature: from Tonic to Subdominant, then to Dominant, and back to Tonic. In the key of C major, the functional groupings are as follows:

Table 2.2 — Tonal Function with chord and each characteristic

Function	Chords	Characteristic
Tonic (T)	I (C), iii (Em), vi (Am)	Stable, sense of “home”
Subdominant (S)	IV (F), ii (Dm)	Restless, sense of “departure”
Dominant (D)	V (G), vii° (Bdim)	Tense, strong pull toward resolution

These functional categories inform the tonic resolution penalty applied at the goal state: a final chord closer to the tonic function receives a lower penalty, incentivizing the search toward harmonically complete progressions. The admissible heuristic used during A* search is described in Chapter 3; it is based on melodic compatibility lower bounds rather than harmonic function distance, which ensures admissibility while remaining musically informative.

4) Voice Leading

Voice leading is the linear progression of individual melodic lines (voices or parts) and their interaction with one another to create harmonies, typically in accordance with the principles of common-practice harmony and counterpoint. These principles include voices sounding smooth and independent, generally minimizing movement to common tones as well as steps to the closest chord tone possible, therefore minimizing leaps where possible.

Good voice leading ensures that music flows naturally and is pleasing to the ear. The key principle to accomplish this, maintain a common tone. Common-tone retention is the single most powerful smoothing tool available; holding a pitch shared between two chords reduces the total interval displacement to zero for that voice.

In the context of chord progression generation, voice leading quality is quantified as the total number of shared notes (common tones) between two consecutive chords. The more common tones two chords share, the smoother their voice leading. For example, C major (C–E–G) and A minor (A–C–E) share two common tones (C and E), making their transition particularly smooth.

C. Brute Force Search

Brute Force is a straightforward problem-solving strategy that resolves a problem directly by exhaustively examining all possible candidate solutions and selecting the one that best satisfies the problem's criteria. This approach is based directly on the problem statement and the definitions involved, making it conceptually simple and easy to implement [3].

In the context of chord progression generation, Brute Force search operates by enumerating every possible combination of chord assignments across all melody phrases. Given n phrases and a candidate chord set of size k per phrase, the total number of combinations making Brute Force an exhaustive enumeration with time complexity $O(k^n)$. For each complete combination, the total harmonic cost is computed, and the combination with the minimum cost is returned as the result.

D. Greedy Best First Search

Greedy Best-First Search (GBFS) is a heuristic search algorithm used to find a path from an initial state to a goal state. It belongs to the category of informed search because it uses additional information in the form of a heuristic

function to guide the search process. GBFS operates by always expanding the state with the smallest heuristic value h(n), where h(n) estimates the cost from the current state to the goal. Its evaluation function is defined purely as:

$$f(n) = h(n)$$

GBFS is typically implemented using a priority queue ordered by h(n). At each step, the state with the smallest heuristic value is dequeued and expanded. Its neighbors are then evaluated using the heuristic function and inserted into the priority queue [1].

In the context of chord progression generation, the Greedy implementation selects at each phrase the chord that minimizes a combined local score consisting of the compatibility penalty, the transition cost from the previous chord, and the heuristic estimated remaining cost. This extends the classic GBFS evaluation beyond h(n) alone to account for local harmonic context, while retaining the greedy property of never backtracking. This allows GBFS to produce a chord progression quickly, as it explores far fewer states than Brute Force.

E. A* Algorithm

A* is a heuristic search algorithm that combines the strengths of Uniform Cost Search (UCS) and Greedy Best-First Search, using an evaluation function that accounts for both the accumulated path cost from the initial state and the estimated remaining cost to the goal [2].

The evaluation function of A* is defined as:

$$f(n) = g(n) + h(n)$$

where g(n) is the cumulative cost from the initial state to node n, and h(n) is the heuristic function estimating the minimum remaining cost from n to the goal [2].

A* is implemented using a priority queue ordered by f(n). The node with the smallest f(n) is always expanded first. When a node is expanded, the algorithm computes f values for all its neighbors and updates the queue if a better path is found [1]. The primary advantage of A* over GBFS is its guarantee of optimality solution.

In the context of chord progression generation, A* balances two forces at every phrase position: the accumulated harmonic cost of the progression so far (g(n)), which reflects how smooth and compatible previous chord choices have been, and the estimated harmonic distance remaining to a satisfying tonal resolution (h(n)). This dual consideration allows A* to avoid locally attractive but globally suboptimal chord choices.

III. METHODOLOGY

A. Problem Definition

Given a melody represented as a MIDI file, the goal of this system is to automatically generate a harmonically coherent chord progression that accompanies the melody. The melody is first segmented into phrases, and for each phrase, exactly one chord is assigned. The system then searches for the sequence of chords that minimizes the total harmonic cost across the entire progression, where cost is defined by the combination of harmonic distance and voice leading smoothness.

Formally, let a melody M be segmented into n phrases:

$$M = \{p_1, p_2, p_3, ..., p_n\}$$

The system must find a chord sequence:

$$C = \{c_1, c_2, c_3, ..., c_n\}$$

where each c_i is selected from the candidate chord set $\mathscr{C}$ consisting of all 24 major and minor triads across 12 keys, such that the total cost of the progression is minimized.

B. MIDI Parsing and Phrase Segmentation

The input to the system is a standard MIDI file. The system extracts the note sequence from the MIDI file using the music21 library, reading each note's pitch (as a MIDI number) and duration (in quarter beats). The key of the melody is automatically detected using music21's key analysis module, which returns both the tonic pitch class and the mode (major or minor).

Phrase boundaries are determined by a fixed-duration segmentation strategy. Given a user-specified phrase size (in quarter beats), the melody is divided into consecutive non-overlapping windows of that duration. Formally, phrase p_i spans beats $[(i-1) \times \text{phrase\_size}, i \times \text{phrase\_size})$, and all notes whose onset falls within that window are assigned to phrase p_i. The total number of phrases is:

$$n = \lfloor \text{total\_beats} / \text{phrase\_size} \rfloor$$

where total_beats is the offset of the last note plus its duration.

Each note within a phrase is stored as a (midi, duration) pair. This representation preserves duration information for use in the compatibility scoring described below.

For each phrase p_i, the set of valid candidate chords $\mathscr{C}_i$ is determined by filtering the full chord catalog to

retain only chords that share at least one pitch class with the notes present in the phrase:

$$\mathscr{C}_i = \{ c\square \in \mathscr{C} \mid \text{pitchClasses}(p_i) \cap \text{notes}(c\square) \neq \varnothing \}$$

If no chord satisfies this condition (an empty phrase), all 24 chords are considered as candidates.

C. State Space Representation

The chord progression problem is modeled as a graph search problem over a state space defined as follows:

- State: A state *s* is defined as a tuple *(i, c_i)*, where *i* is the current phrase index (1 to n) and c_i is the chord assigned to phrase *i*.
- Initial State: A virtual start node s_0 with no chord assigned, connected to all valid candidates for phrase p_1.
- Goal State: Any state *(n, c□)* where all n phrases have been assigned a chord — specifically, the goal prefers the final chord to be the tonic chord of the detected key, representing harmonic resolution.
- Actions: From state *(i, c_i)*, the system can transition to any state *(i+1, c□)* where c□ is a valid candidate chord for phrase p_{i+1}.
- Path: A complete path from s_0 to a goal state represents a full chord progression C = {c_1, c_2, ..., c□}.

The total number of states in the search space is at most n × 24, and the number of possible complete progressions is at most 24^n.

D. Cost Function Design

1) Edge Cost g(n)

The cumulative cost g(n) of reaching state *(i, c_i)* is the sum of transition costs along the path from the initial state. The transition cost between two consecutive chords c_i and c_{i+1} is defined as:

$$\text{cost}(c_i, c_{i+1}) = \alpha \cdot \text{d\_fifth}(c_i, c_{i+1}) + \beta \cdot (1 - \text{vl}(c_i, c_{i+1}))$$

where:

- $\text{d\_fifth}(c_i, c_{i+1})$ is the circle of fifths distance between the two chords, computed separately for major and minor chords using two distinct circle of fifths orderings:

  CIRCLE_MAJOR = [C, G, D, A, E, B, F#, C#, G#, D#, A#, F]
  CIRCLE_MINOR = [A, E, B, F#, C#, G#, D#, A#, F, C, G, D]

  The distance is normalized to [0, 1]:
  $\text{d\_fifth}(r_1, r_2) = \min(|\text{pos}(r_1) - \text{pos}(r_2)|, 12 - |\text{pos}(r_1) - \text{pos}(r_2)|) / 6$

- $\text{vl}(c_i, c_{i+1})$ is the voice leading score based on common tone count, normalized to [0, 1]:

$$\text{vl}(c_i, c_{i+1}) = |\text{notes}(c_i) \cap \text{notes}(c_{i+1})| / 3$$

- α and β are weighting coefficients satisfying α + β = 1, controlling the relative importance of harmonic distance vs. voice leading smoothness. In this paper, α = 0.6 and β = 0.4 are used as default weights, determined empirically.

A lower transition cost indicates a more natural, musically smooth chord change. For example, the transition C major → G major has d_fifth = 1/6 ≈ 0.167 (adjacent on a circle of fifths) and vl = 1/3 ≈ 0.333 (one common tone: G), yielding a low cost of approximately 0.233.

2) Additional Phrase Compatibility Penalty

To ensure the assigned chord is melodically compatible with the phrase it accompanies, a compatibility penalty is added to the path cost when entering state (i, c_i). Unlike a simple pitch class overlap count, this penalty is duration-weighted: pitch classes that are sustained longer within a phrase contribute more to the compatibility score, reflecting their greater perceptual prominence.

For each phrase p_i, a weight map W is constructed by accumulating the total duration of each pitch class across all notes in the phrase:

$W(pc) = \Sigma\ \text{dur}(n)$ for all notes n in p_i where pitch_class(n) = pc

The compatibility penalty is then defined as:

$\text{penalty}(p_i, c_i) = 1 - (\Sigma_{pc \in \text{notes}(c_i)} W(pc)) / (\Sigma_{pc} W(pc))$

This is the proportion of total phrase duration covered by pitch classes that belong to the chord. A penalty of 0 means all note durations in the phrase are chord tones; a penalty approaching 1 means the chord

shares little durational weight with the phrase.

3) Tonic Resolution Penalty

At the goal state (n, c□), an additional tonic resolution penalty is applied to encourage the progression to end on a harmonically stable chord. This penalty is based on the harmonic function of the final chord relative to the detected tonic:

Table 3.1 — Tonic resolution penalty

Interval from Tonic	Diatonic Function	Expected Quality	Penalty
0 (I)	Tonic	major	0
7 (V)	Dominant	major	1
5 (IV)	Subdominant	major	2
9 (VI)	Submediant	minor	3
4 (III)	Mediant	minor	4
11 (VII)	Leading Tone	dim	5
other	Non-diatonic	—	5

If the final chord's quality does not match the expected quality for its diatonic function, an additional penalty of 1 is added. The tonic resolution penalty is scaled by 0.5 and added to the path cost before the final-phrase node is pushed onto the priority queue, ensuring that f-values remain consistent across all nodes during search.

E. Heuristic Function Design

The heuristic function h(n) estimates the remaining cost from state (i, c_i) to the goal and must be admissible to guarantee that A* returns an optimal solution.

The heuristic is defined as the sum of minimum possible compatibility penalties over all remaining phrases:

h(i, c_i) = Σ_{j=i+1}^{n} min_{c ∈ 𝒞□} penalty(p□, c)

For each remaining phrase p□, the minimum penalty achievable by any valid candidate chord is computed. Since compatibility penalty is always non-negative and the actual cost must include at least this penalty for each phrase (in addition to transition costs, which are also non-negative), this heuristic is a true lower bound and is therefore admissible.

The heuristic is also informative: it reflects the actual melodic content of upcoming phrases rather than a generic constant, allowing A* to effectively prune paths that cannot lead to a low-cost solution.

F. Algorithm Implementations

1) Brute Force

The Brute Force algorithm exhaustively enumerates all possible chord progressions by generating the complete Cartesian product of valid candidate chords across all phrases. For each complete progression, the total cost is computed and the minimum-cost progression is returned. Time complexity is $O(k^n)$ where k is the average number of valid candidates per phrase and n is the number of phrases. To prevent excessive runtime, a maximum combination threshold of 500,000 is enforced; songs exceeding this limit are skipped. This makes Brute Force feasible only for short melodies ($n \leq 5$ approximately).

2) Greedy Best-First Search

At each phrase position i, Greedy selects the chord c that minimizes the combined score:

score(c) = penalty(p_i, c) + trans(c_{i-1}, c) + h(i, c)

where trans is the transition cost from the previously selected chord and h is the heuristic estimated remaining cost. Once a chord is selected, the decision is final — Greedy never backtracks. The reported total cost is computed using actual path costs (penalty + transition) without the heuristic term, to ensure comparability with Brute Force and A* results.

3) A* Search

A* maintains a priority queue ordered by f(n) = g(n) + h(n), where g(n) is the accumulated actual path cost (compatibility penalties plus transition costs) and h(n) is the admissible heuristic described above. At each step, the state with the lowest f value is expanded. A visited dictionary tracks the best known g-cost for each state (i, c); a state is skipped if it is reached again with a higher or equal g-cost.

A critical implementation detail is that the tonic resolution penalty for final-phrase candidates is added to g before those nodes are pushed onto the priority queue, not after they are popped as goal states. This ensures that final-phrase nodes compete fairly with non-final nodes in the priority queue ordering, preventing a non-tonic chord from

appearing artificially cheaper and being selected prematurely as the terminal chord.

Because the heuristic is admissible, A* guarantees finding the globally optimal chord progression while exploring significantly fewer states than Brute Force.

G. Algorithm Example: Twinkle Twinkle Little Star

To easiness understanding for reader, this section traces the algorithm step by step on the melody "Twinkle Twinkle Little Star" in C major using a phrase size of 8 beats and A* algorithm.

1) MIDI Parsing and Key Detection

The input MIDI file contains 28 notes spanning 32 beats total. The system detects the key automatically as C major, giving a tonic root of pitch class 0. All subsequent cost and heuristic computations are relative to this tonic.

2) Phrase Segmentation

With phrase_size = 8 beats, the melody is divided into 4 phrases. For each note, its pitch class is computed as MIDI number mod 12 (e.g., MIDI 60 = C4 → 60 mod 12 = 0). The resulting phrases are:

Table 3.2 — Phrase Segmentation

Phrase	Beats	Notes	Pitch Class Set
1	0-8	C C G G A A G	{C=0, G=7, A=9}
2	8-16	F F E E D D C	{C=0, D=2, E=4, F=5}
3	17-24	G G F F E E D	{D=2, E=4, F=5, G=7}
4	24-32	G G F F E E D	{D=2, E=4, F=5, G=7}

3) Candidate Chord Filtering

For each phrase, only chords sharing at least one pitch class with the phrase are retained as candidates. Phrase 1 with pitch classes {C, G, A} yields 14 compatible chords, including Cmaj, Cmin, Fmaj, and Amin.

4) Compatibility Penalty Computation

The compatibility penalty between Phrase 1 and Cmaj is computed using the duration-weighted formula. Phrase 1 contains 7 notes: C appears twice, G appears three times, and A appears twice, giving total_weight = 7. The chord tones of Cmaj are {C, E, G}:

chord_weight = W(C) + W(G) = 2 + 3 = 5

penalty(Phrase 1, Cmaj) = 1 − (5/7) = 0.2857

By the same method, penalty(Phrase 1, Amin) = 1 − (4/7) = 0.4286, and penalty(Phrase 1, Gmaj) = 1 − (3/7) = 0.5714.

5) A* Search Execution

Initialization. All 14 candidate chords for Phrase 1 are inserted into the priority queue with f = g + h, where g is the compatibility penalty and h is the sum of minimum compatibility penalties over remaining phrases:

Table 3.3 — Initial Priority Queue (selected entries)

Chord	g(compact)	h(future est.)	f = g + h
Cmaj	0.2857	1.2857	1.5714
Amin	0.4286	1.2857	1.7143
Fmaj	0.4286	1.2857	1.7143
Gmaj	0.5714	1.2857	1.8571

Cmaj has the lowest f = 1.5714 and is expanded first. Search trace. The table below shows each node expansion in order:

Table 3.4 — A* Node Expansion Trace

Step	State(phrase, chord)	g	f	Action
1	(1, Cmaj)	0.2857	1.5714	Expand → push Phrase 2 candidates
2	(2, Cmaj)	0.7143	~1.5714	Expand → push Phrase 3 candidates
3	(3, Cmaj)	1.1429	~1.5714	Expand → push Phrase 4 candidates
4	(4, Cmaj)	1.750	1.750	GOAL — tonic penalty = 0 → final cost = **1.7500**

Path reconstruction. The parent dictionary is traced backward from the goal state:
(4, Cmaj) ← (3, Cmaj) ← (2, Cmaj) ← (1, Cmaj)
Final output: Cmaj → Cmaj → Cmaj → Cmaj with total cost 1.7500.

IV. Experiment and Analysis

A. Experimental Setup

Experiments were conducted on four test songs with varying complexity to evaluate the three algorithms across different search space sizes. The songs were selected to represent a range of melodic complexity from simple kid western rhymes to a moderately complex Indonesian popular song.

Table 4.1 — Comparison of test songs

Song	Complexity	Phrases (n)	Key	Phrase_size
Twinkle Twinkle Little star	Simple	4	C major	8 beats
Happy Birthday to You	Simple	3	F major	8 beats
Pelangi-Pelangi	Medium	6	C major	8 beats
I Will Fly	Complex	104	C major	4 beats

Each song was loaded as a MIDI file and parsed into phrases using the fixed-duration segmentation method described in Chapter 3. Phrase pitches were stored as weighted (midi, duration) tuples to enable duration-weighted compatibility scoring. Three algorithms were applied to each song: Brute Force, Greedy Best-First Search, and A*. Metrics collected for each run were: (1) the generated chord progression, (2) total harmonic cost, (3) number of states explored, and (4) execution time in milliseconds.

B. Results

For the Brute Force algorithm, some songs are skipped because they consume too much time and space (not efficient).

Table 4.2 — Computational Performance

Song	Algorithm	Total Cost	States Explored	Time (ms)
Twinkle	Brute Force	1.7500	81648	2071.886
	Greedy	1.7500	68	7.356
	A*	1.7500	5	7.895
Happy Birthday	Brute Force	1.2500	6426	131.935
	Greedy	1.4917	56	6.513
	A*	1.2500	23	49.401
Pelangi-Pelangi	Brute Force	SKIPPED	SKIPPED	SKIPPED
	Greedy	2.00	113	23.808
	A*	2.00	10	55.649
I will fly	Brute Force	SKIPPED	SKIPPED	SKIPPED
	Greedy	49.1145	1959	13775.324
	A*	39.5281	21533	179340.468

Table 4.3 — Generated Chord Progressions

Song	Algorithm	Progression
Twinkle	All	Cmaj → Cmaj → Cmaj → Cmaj
Happy Birthday	Brute Force / A*	Fmaj → Fmaj → Fmaj
Happy Birthday	Greedy	Cmaj → Fmaj → Fmaj
Pelangi-Pelangi	Greedy / A*	Cmaj → Cmaj → Cmaj → Cmaj → Cmaj → Cmaj
I Will Fly	Greedy	Cmaj×9 → Gmaj → Cmaj×2 → Amin×2 → Emin×6 → Cmaj×4 →

		Fmaj×2 → Gmaj×3 → Dmin → Emin×4 → Cmaj×9 → Amin×2 → Emin×3 → Gmaj×2 → Emin×3 → Cmaj×9 → Fmaj×2 → Cmaj×10 → Amin×7 → Cmaj → Dmin×13 → Amin → Dmin×4 → Amin → Dmin×4 → Amaj
I Will Fly	A*	Emin×2 → Gmaj×8 → Cmaj×2 → Amin×2 → Cmaj×46 → Fmaj×2 → Cmaj×10 → Dmaj×28 → Amaj×4

C. Analysis

1) Optimality: A* vs Brute Force

For songs where Brute Force was feasible (Twinkle, n=4 and Happy Birthday, n=3), A* produced an identical total cost to Brute Force in both cases. For Twinkle Twinkle, all three algorithms converged on Cmaj → Cmaj → Cmaj → Cmaj with cost 1.7500, consistent with C major as the detected key. For Happy Birthday, both Brute Force and A* converged on Fmaj → Fmaj → Fmaj with cost 1.2500, confirming that A*'s heuristic is admissible and the implementation produces optimal results.

It is worth noting that this optimality guarantee was made possible by a correction to the tonic resolution penalty: the penalty term get_harmonic_function(chord, tonic) × 0.5 is now added to the path cost before a final-phrase node is pushed onto the priority queue, rather than after it is popped as a goal. This ensures that f-values are consistent across all nodes during comparison, preventing a non-tonic chord from appearing artificially cheaper than it is and being selected prematurely as the terminal chord.

2) Greedy Suboptimality

The most instructive result appears in Happy Birthday: Greedy produced Cmaj → Fmaj → Fmaj with cost 1.4917, compared to the optimal 1.2500 found by Brute Force and A*. Although Greedy's progression is musically reasonable — it includes Fmaj (the tonic of F major) in the second and third phrases — its locally greedy selection at phrase 1 chose Cmaj over Fmaj, incurring a suboptimal transition. This illustrates the core limitation of greedy search: the chord that appears cheapest in isolation at an early phrase may not lead to the globally cheapest path.

For I Will Fly (n=104), this gap widens considerably: Greedy produced cost 49.1145 versus A*'s 39.5281, representing a 19.5% improvement in solution quality by A*. The Greedy progression exhibits long monotonous stretches of repeated chords (e.g., Dmin repeated 13 consecutive times near the end), whereas A* produced a more varied and structured progression. This demonstrates that the quality gap between A* and Greedy grows with melody length and complexity.

3) State Exploration Efficiency

A* consistently explored far fewer states than Brute Force. For Twinkle (n=4), Brute Force explored 81,648 states while A* explored only 5 — a reduction of more than 16,000×. For Happy Birthday (n=3), the ratio was 6,426 to 23 — a 279× reduction. This dramatic pruning arises from the admissible heuristic, which computes a lower bound on remaining compatibility cost across all future phrases and eliminates suboptimal paths early.

Greedy explored the fewest states in all cases (68 for Twinkle, 56 for Happy Birthday, 113 for Pelangi-Pelangi, 1,959 for I Will Fly) because it never backtracks and commits to exactly one chord per phrase. However, as demonstrated above, this efficiency comes at the direct expense of solution quality.

For Pelangi-Pelangi (n=6), both Greedy and A* converged on the same progression (all Cmaj, cost 2.0000), with A* doing so in only 10 state expansions versus Greedy's 113. This shows that even when the two algorithms agree on the solution, A*'s structured search is more directed.

4) Execution Time Trade-offs
For simple and medium songs ($n \leq 6$), execution times for Greedy and A* are in the single-digit to double-digit millisecond range, well within practical limits. Brute Force, despite being feasible for small n, already required 2,071ms for Twinkle (n=4) and 131ms for Happy Birthday (n=3) orders of magnitude slower than the informed algorithms.

For I Will Fly (n=104), a significant trade-off emerges: A*'s execution time was 179,340ms (approximately 179 seconds), compared to Greedy's 13,775ms (approximately 14 seconds). This overhead stems from the heuristic computation itself: for each node expansion, the admissible heuristic iterates over all remaining phrases to compute per-phrase minimum compatibility penalties, an $O(n \times k)$ operation where k is the average number of chord candidates per phrase. Over 21,533 state expansions with up to 104 remaining phrases each, this cost accumulates substantially. This reveals a fundamental trade-off: the more informed the heuristic, the better the solution quality, but the higher the per-node computational cost.

5) Qualitative Assessment of I Will Fly
The original chord progression of "I Will Fly" by Ten2Five is Am – F – C – G (in A minor). The system detected C major as the key, which is the relative major of A minor. Both share the same key signature, and music21's key detection algorithm defaults to the major interpretation in such cases.
The Greedy output is dominated by Cmaj in the first half and Dmin in the latter half, with limited harmonic movement. Its final chord, Amaj, is non-diatonic to C major, reflecting the greedy algorithm's inability to plan ahead for a harmonically appropriate resolution.

The A* output opens with Emin and Gmaj both diatonic to C major and consistent with the original song's minor-inflected character before settling into a long Cmaj stretch, transitioning through Fmaj, and concluding with an extended Dmaj run ending on Amaj. While neither output reproduces the original Am–F–C–G pattern exactly, the A* result is structurally more coherent, its chord changes occur at meaningful phrase boundaries, and its lower total cost (39.5281 vs 49.1145) reflects a globally better alignment between the melody's pitch content and the assigned chords. This supports the conclusion that optimizing global harmonic cost produces qualitatively more musically appropriate progressions than locally greedy selection.

V. Conclusion

This paper presented an automatic chord progression generator that models harmonic accompaniment as a graph search over a state space of chord assignments, comparing Brute Force, Greedy Best-First Search, and A*. Musical accuracy was improved through duration-weighted compatibility scoring, separate circle-of-fifths representations for major and minor chords, quality-aware harmonic function evaluation, and voice-leading-optimized MIDI export.

A* consistently matched Brute Force wherever feasible and outperformed Greedy in solution quality across all songs where a difference emerged. On Happy Birthday, both A* and Brute Force converged on Fmaj → Fmaj → Fmaj with cost 1.2500, confirming heuristic admissibility — contingent on incorporating the tonic resolution penalty into node cost at enqueue time rather than after a final-phrase candidate is dequeued, which preserves consistent f-value ordering throughout the search.

Future work could extend the fixed-duration phrase segmentation toward boundaries informed by rests, melodic contour, or beat-level salience; broaden the chord catalog with seventh chords, borrowed chords, and secondary dominants; and refine key detection to distinguish relative major/minor keys — relevant to songs like "I Will Fly," where an A minor interpretation fits better than C major. A perceptual listener study would also complement the cost function with an objective measure of musical quality.

Video Link at Youtube and GitHub

Youtube: https://youtu.be/D7kTttiNNnc

GitHub: https://github.com/hakamavicena/makalah-stima-chord-generation-by-algorithm

Acknowledgment

The author thanks Rinaldi Munir, lecturer of IF2211 Strategi Algoritma, for the theoretical foundation on pathfinding and search algorithms that made this work possible. The author also thanks the class assistants of IF2211 for their guidance throughout the course. Finally, the author thanks his family and friends for their continuous support and encouragement during the completion of this paper.

References

[1] N. U. Maulidevi, "Penentuan Rute (Route/Path Planning) Bagian 1: BFS, DFS, UCS, Greedy Best First Search," Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2026.

[2] N. U. Maulidevi and Rinaldi M, "Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*," Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2026.

[3] A. Levitin, Introduction to The Design and Analysis of Algorithms, 3rd ed. Boston: Pearson, 2012.

[4] MusicRadar, "The ultimate guide to the circle of fifths," musicradar.com, Jun. 2025.

[5] I. Tonality, "Harmonic Direction I: Tonal Functions," rwu.pressbooks.pub, 2023.

[6] Iowa State University, "6.1 Functional Harmony: Tutorial," iastate.pressbooks.pub, 2023.

[7] Music Theory Authority, "Voice Leading Principles," musictheoryauthority.com.

[8] Berklee Online, "Voice Leading Paradigms for Harmony in Music Composition," online.berklee.edu, Oct. 2025.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Hakam Avicena Mustain