CS517 Theory of Computation - Final Project
Constraint-Based Playlist Generation via SMT Solving

Alena Makarova
School of Electrical Engineering and Computer Science
Oregon State University
`makarova@oregonstate.edu`

Pratik Khanal
School of Electrical Engineering and Computer Science
Oregon State University
`khanalp@oregonstate.edu`

Abstract

We present a constraint-based method for automatically generating playlists using SMT solvers. Given a set of candidate tracks with metadata such as duration, valence (emotional tone), energy (intensity), genre, and popularity, our tool selects and orders songs to meet several global constraints. The resulting playlist must stay within a target duration range, ensure smooth transitions in valence and energy, and achieve a minimum average popularity. The challenge lies not just in filtering songs but in sequencing them to satisfy all constraints, making this a combinatorial optimization task. We formulate the problem using Z3 with both Boolean and arithmetic constraints. While the long-term goal is to minimize emotional roughness, our current focus is on finding feasible playlists that meet predefined bounds. Experiments on real Spotify-based data show that our approach generates valid and interpretable playlists efficiently for moderate input sizes.

1 Introduction

Playlist generation is central to music platforms like Spotify, Apple Music, and YouTube Music. Listeners often want playlists that fit a specific mood, genre, or time constraint, such as a relaxing morning mix or a high-energy workout set. Beyond casual use, playlist structure also matters for DJs, therapists, educators, and streaming platforms aiming to provide interpretable, dynamic, and user-centered recommendations.

Most existing methods, such as collaborative filtering and reinforcement learning, effectively capture user preferences but offer limited control over playlist structure. In particular, it's challenging to ensure smooth transitions in energy or mood or to enforce rules such as minimum popularity of songs in the public or balanced selection of different genres.

We propose a constraint-based approach to generate playlists that satisfy user-defined global constraints. Given metadata for a set of songs including duration, valence (pleasantness of the song), energy (intensity), genre, and popularity, our tool selects and orders tracks to:

- Fit a target duration range,
- Ensure smooth transitions in valence and energy,
- Achieve a minimum average popularity.

For instance, a valid playlist might exhibit a gradual mood shift with valence values such as 0.60, 0.65, 0.70, and 0.68, whereas abrupt jumps (e.g., 0.60 $\rightarrow$ 0.95 $\rightarrow$ 0.30) would violate constraints and disrupt the listening experience (Table 1).

Song Position	**Valence (Valid)**	**Valence (Invalid)**
1	0.60	0.60
2	0.65	0.95
3	0.70	0.30
4	0.68	0.70

Table 1: Gradual (valid) vs. abrupt (invalid) valence transitions.

We model the problem as an SMT optimization task using Z3. The formulation combines logical constraints (e.g., song ordering) with numerical ones (e.g., valence jumps), making it difficult for greedy or SAT-only methods but well-suited for SMT.

This paper describes our encoding, solver-based implementation, and results on real Spotify data. We show that constraint-based playlist generation is practical, interpretable, and useful for applications where structure and control matter.

2 Related work

Several approaches to playlist generation have been explored in the past. Many systems use machine learning [3] or heuristic methods, such as collaborative filtering or content-based similarity. For example, [1] provides a detailed survey of these techniques and explains how they are applied in real-world systems, such as Spotify and Pandora. These methods can be effective in capturing user preferences, but they often lack transparency and control. It's challenging to ensure that a playlist has a specific structure or adheres to rules such as total duration or smooth emotional transitions.

Reinforcement learning has also been applied to this task. [5] designed an interactive system that learns to recommend music through trial and error based on user feedback. This approach can adapt to individual users, but it doesn't guarantee that playlists follow specific rules or constraints.

Another challenge in playlist generation is that music similarity is subjective. [2] showed that even humans often disagree when judging whether two songs are similar. This makes it harder for machine learning models to make good decisions unless the user can clearly define what they want.

Constraint-based methods attempt to address this by allowing users to set specific rules. [4] were among the first to show how playlists can be generated using constraint satisfaction techniques. Their system allowed users to define preferences, such as tempo or artist diversity. However, earlier systems didn't support numeric features like valence or energy very well.

Our approach builds on these ideas by using an SMT solver, which can handle both logical and numerical constraints. This enables the creation of playlists that not only respect user preferences but also adhere to strict structural rules.

3 Problem Definition

We aim to generate a music playlist of fixed duration (e.g., 30 minutes), where the selected and ordered songs satisfy user-defined constraints. Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be a set of songs, where each s_i is described by:

$$s_i = (d_i, v_i, e_i, p_i)$$

with d_i as duration, v_i as valence (emotional positivity), e_i as energy (intensity), and p_i as popularity. The goal is to construct an ordered sequence $P = [s_{i_1}, \ldots, s_{i_k}] \subset \mathcal{S}$ such that the total duration falls within a target range, valence, and energy transitions between adjacent songs are smooth, and the average popularity exceeds a user-defined minimum.

The task is to find such a sequence P or determine that none exists.

3.1 Problem Complexity

This problem is more than just filtering songs by individual features; it also requires sequencing them to meet global constraints, such as smooth emotional transitions and total duration. This turns it into a combinatorial optimization task that combines elements of the *Knapsack Problem* (selecting songs to meet a duration goal) and the *Traveling Salesman Problem* (ordering songs with minimal transition cost based on valence and energy). Both problems are NP-complete, and their combination renders the playlist generation problem NP-hard. By reducing playlist generation to a satisfiability problem, we leverage modern SMT solvers to handle this complexity efficiently.

4 Reducing the Playlist Problem to SMT

We reduce the constrained playlist generation problem to an SMT instance using the Z3 solver. The solver selects and arranges a subset of songs to satisfy all given constraints. The encoding uses a mix of Boolean and real-valued variables.

Let S be the set of available songs, and let k be the maximum number of songs allowed in the final playlist (i.e., the number of time slots). For each song i and slot j, we define a Boolean variable x_{ij}, which is true if and only if song i is placed in slot j.

We also define real-valued helper variables for valence and energy at each slot j, denoted by v_j and e_j, respectively. The main components of the encoding are:

- **Duration constraint:** Let d_i be the duration (in seconds) of song i. The total duration of selected songs must fall within a specified tolerance window around a target time T:

$$T - \delta \leq \sum_{i,j} d_i \cdot x_{ij} \leq T + \delta$$

- **Valence/Energy smoothness:** For each adjacent pair of slots $(j, j+1)$, the difference in valence and energy must not exceed pre-defined thresholds:

$$|v_{j+1} - v_j| \leq \epsilon_v \quad \text{and} \quad |e_{j+1} - e_j| \leq \epsilon_e$$

- **Popularity constraint:** Let p_i be the popularity of song i. The average popularity across the playlist must meet a given threshold $P_{\min}$:

$$\frac{1}{k} \sum_{i,j} p_i \cdot x_{ij} \geq P_{\min}$$

Objective: While the ideal goal is to minimize emotional "roughness" (i.e., the sum of valence and energy changes between consecutive tracks), our formulation treats this as a decision problem: can we construct a playlist whose roughness does not exceed a given threshold R?

$$\sum_{j=1}^{k-1} |v_{j+1} - v_j| + |e_{j+1} - e_j| \leq R$$

4.1 Toy Example: 3 Songs, 2 Slots

To show how the encoding works, suppose we have 3 songs with the following metadata:

Song	Duration	Valence	Energy	Popularity
s_1	130	0.6	0.7	60
s_2	200	0.9	0.8	50
s_3	150	0.3	0.6	70

Table 2: Metadata for 3-song example used in SMT encoding.

We define Boolean variables x_{ij} for each combination of song $i \in \{1, 2, 3\}$ and slot $j \in \{1, 2\}$, a total of 6 variables.

Each slot must contain exactly one song:

$$x_{11} + x_{21} + x_{31} = 1, \quad x_{12} + x_{22} + x_{32} = 1$$

Each song appears at most once:

$$x_{i1} + x_{i2} \leq 1 \quad \text{for } i = 1, 2, 3$$

We compute the valence and energy in each slot as weighted sums. For example:

$$v_1 = 0.6x_{11} + 0.9x_{21} + 0.3x_{31}, \quad v_2 = 0.6x_{12} + 0.9x_{22} + 0.3x_{32}$$

The solver then checks if it can assign truth values to x_{ij} variables such that all constraints (duration, valence/energy difference, and average popularity) are satisfied. If so, it returns a playlist, e.g., s_3 in slot 1 and s_1 in slot 2 that meets all the criteria.

4.2 Implementation Details

We implement the above encoding using the Z3 SMT solver. Each Boolean variable x_{ij} is represented using Z3's `Bool`, while helper variables like v_j, e_j are real-valued (`Real`). Constraints for the duration, valence/energy jumps, and popularity are added directly using Z3's arithmetic expressions. Absolute differences (for smoothness) are encoded using auxiliary variables with linear constraints. We first frame the problem as a decision task by bounding roughness and optionally use Z3's Optimize module to minimize roughness once a feasible playlist is found. If the constraints are satisfiable, the resulting model is parsed into an ordered playlist and exported as structured JSON output.

5 Scalability and Optimization Techniques

We encode playlist generation as an SMT problem using Z3. The solver selects and orders k tracks from n candidates to meet user-defined constraints on duration, smoothness, and popularity.

5.1 Encoding Overview

We define Boolean variables x_{ij} to indicate whether track i is placed in slot j. Each slot also has real-valued helper variables v_j and e_j for valence and energy. To capture emotional variation, we compute absolute differences between adjacent slots:

$$\texttt{roughness} = \sum_j |v_{j+1} - v_j| + |e_{j+1} - e_j|$$

Instead of minimizing roughness directly, we constrain it to be below a global roughness bound R and solve the instance. We can tighten R iteratively to find near-optimal playlists.

5.2 Model Size and Complexity

The number of Boolean variables grows as $O(n \cdot k)$. Real-valued constraints for valence, energy, duration, and popularity add further overhead. The encoding is tractable for moderate sizes ($n < 150$, $k < 10$), but solving may slow down with larger or highly constrained instances.

5.3 Efficiency Tricks

To improve scalability, we apply several techniques:

- Use helper variables for valence and energy to express constraints linearly.
- Apply genre constraints only to frequently occurring genres.
- Allow optional random sampling of tracks to reduce input size.
- Enforce a 180-second timeout per instance; unsolved cases can be retried with relaxed constraints.

While we experimented with ideas for soft constraints and dynamic playlist lengths, our current model does not fully support them, so we kept k fixed.

5.4 Solver Workflow

The playlist generation process follows these steps:

Algorithm 1 Constraint-Based Playlist Generation

```
Require: Track list $\mathcal{S}$, playlist length $k$, duration target $T$, valence/energy thresholds $\epsilon_v, \epsilon_e$, popularity threshold $P_{\min}$
 1: Optionally sample a subset of $\mathcal{S}$
 2: Create variables $x_{ij}, v_j, e_j$ for each track and slot
 3: Add constraints:
        • One track per slot, each track used at most once
        • Duration within $[T \pm \delta]$
        • Bounded valence/energy transitions
        • Average popularity $\geq P_{\min}$
 4: Constrain roughness $\leq R$
 5: Solve with timeout (e.g., 60s)
 6: if solution found then
 7:     return valid playlist
 8: else
 9:     return failure message
10: end if
```

If Z3 times out or returns `unsat`, the instance is logged with an error code. This helps identify hard cases often due to tight transition bounds or high popularity cutoffs. While we don't use a debug mode, logging failures across parameter settings helps guide constraint tuning.

6 Experimental Evaluation

We evaluated the performance of our SMT-based playlist generation tool on a real-world filtered subset of the Spotify Million Playlist Dataset[1], which includes metadata such as track ID, genre, valence, energy, duration, and popularity. We preprocess the data into a JSONL format, with one object per song, which is suitable for SMT encoding. Each track was annotated with metadata, including duration, valence, energy, popularity, and genre. In our dataset, emotion scores in the form of valence and energy are between 0 and 1 and popularity scores range from 0 to 100. We first filter the data consisting more than a million Spotify tracks to obtain tracks with a duration between two and five minutes and popularity score above 40. This resulted in a total data of over 115,000 tracks consisting only the required features. We measured solve times under varying input sizes and constraint settings to assess both scalability and constraint satisfiability.

6.1 Setup

All experiments were conducted on a laptop equipped with an Intel i7 processor (2.6 GHz) and 16 GB of RAM. The SMT solver used was Z3. For each test, we varied the number of available tracks (n), maximum playlist length (k), and constraint tightness (valence/energy delta thresholds).

6.2 Input, Output, and Example Result

Our tool takes as input a JSONL file containing one JSON object per track, derived from a filtered CSV of Spotify metadata. Each entry includes fields such as `track_id`, `duration_sec`, `valence`, `energy`, `popularity`, and `genre`. Command-line arguments define constraints such as target duration, allowed valence and energy jumps, playlist length, and minimum average popularity.

The output is a structured JSON file that contains:

- `playlist`: An ordered list of selected tracks with metadata and start times;
- `total_duration_sec`: The overall playlist duration;

[1] `https://www.kaggle.com/datasets/amitanshjoshi/spotify-1million-tracks`

- `objective_val`: The total emotional roughness (sum of valence and energy differences);
- Or a failure message if no feasible playlist is found within the timeout.

Below is an example playlist generated with the following settings: $n = 60$, $k = 4$, duration target 600s with ± 100 tolerance, valence delta 0.1, energy delta 0.3, and average popularity greater or equal to 50. The result had a total duration of 699 seconds and an objective value of 0.749.

Slot	Track Name	Artist	Genre	Duration (s)
1	Go Solo (feat. Tom Rosenthal)	Zwette	german	192
2	Denkmal	Stereoact	party	175
3	Burn Dem Bridges	Skin On Skin	techno	175
4	Alone	HIMALAYAS	alt-rock	157

Start Time (s)	Valence	Energy	Popularity
0	0.368	0.557	60
192	0.308	0.689	42
367	0.351	0.837	60
542	0.252	0.877	52

Table 3: Track metadata split across two subtables for readability

6.3 Performance and Practical Limits

We evaluated how solver performance scales with different parameters, focusing on valence delta limits, playlist length, and track universe size. Thus, we can summarize the key trends observed across the experiments.

Effect of Valence Jump Limit (Δ): Figure 1 shows that solve time is very sensitive to how much we allow the valence to change between songs. We keep other constraints fixed and only change the valence jump. When the limit is minimal (e.g., $\Delta = 0.05$), the solver takes much longer to find a playlist because the options are more restricted. But once the limit is relaxed past 0.15, the solver finds solutions much faster. This happens because it's easier to find valid transitions when songs are allowed to differ more in valence.

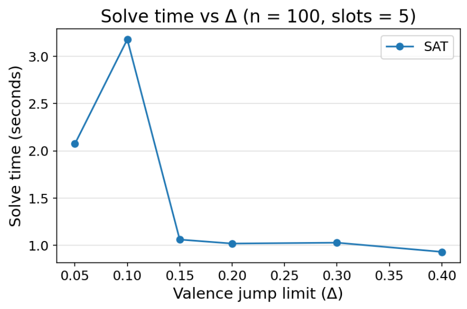

Figure 1: Solve time vs. valence jump limit Δ (n = 100, slots = 5).

Effect of Track Universe Size (n): As shown in Figure 2, the tool scales reasonably well up to $n = 1000$ tracks. However, when $n = 10,000$, the instance becomes unsatisfiable or exceeds the timeout limit. This is expected, as the SMT encoding introduces $O(n \cdot k)$ Boolean variables and additional constraints, resulting in combinatorial growth.

Effect of Playlist Length (k): Figure 3 shows that the time it takes to solve the problem grows quickly as the playlist length increases. When $k \leq 10$, the solver usually finds a solution within a few seconds. But for longer playlists, especially when $k > 10$, the solver often runs out of time. This is because each additional slot introduces more decision variables and constraints between songs, making the problem significantly more challenging to solve. Playlist length turns out to be one of the most significant factors affecting performance.

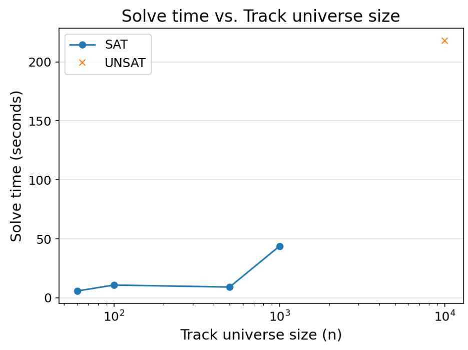

Figure 2: Solve time vs. track universe size (n).

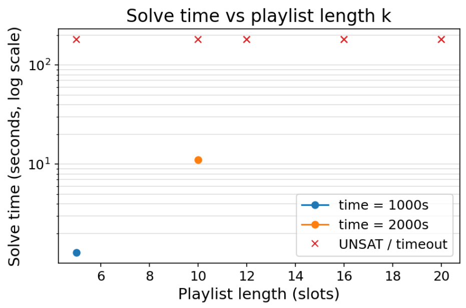

Figure 3: Solve time vs. playlist length (k), on log scale.

Since our playlist generation tool samples a subset of tracks before solving, we wanted to understand how this randomness affects results. Specifically, we aimed to determine whether changing the random seed (i.e., the track sample) affects playlist quality and solver performance.

To check this, we fixed all other parameters and ran multiple trials with different seeds to generate 100 random tracks. We then recorded the resulting objective values and solve times (Figure 4).

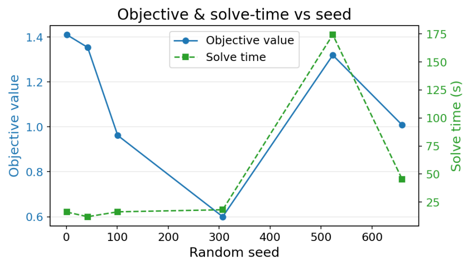

Figure 4: Variation in objective value and solve time across random seeds.

The blue line (left axis) tracks the emotional roughness of the playlist. Some seeds, like 307, lead to smooth playlists (objective ≈ 0.6), while others, like 1 and 42, result in rougher transitions (objective > 1.3). The green dashed line (right axis) shows solve time. Most runs stay under 60 seconds, but seed 523 takes nearly 180 seconds. This indicates that some samples are easier for the solver and yield better results, while others increase complexity.

To better understand how solver performance scales with input size, we benchmarked our SMT-based playlist generator across varying playlist lengths $k \in \{4, 8, 10\}$ and dataset sizes $n \in \{60, 80, 100, 500\}$, using consistent constraint settings (target duration = 1000s, valence and en-

k n	4	8	10
60	**103.8**	180.6	180.6
80	180.5	180.7	180.8
100	**56.1**	180.7	180.7
500	181.2	181.9	182.4

Table 4: Median solve times (in seconds) for different values of n and k

ergy delta = 0.2, popularity threshold = 50). For each configuration, we ran multiple trials and recorded the median solution time.

Table 4 summarizes the results. As expected, solve time increases with both n and k. While smaller configurations (e.g., $n = 60$, $k = 4$) solve in just over 100 seconds, most other settings approach or exceed the 180-second timeout. The results suggest that the solver struggles to scale beyond moderate playlist lengths or large track pools under strict constraints.

To gauge how the solver copes with a different target duration (*time* $= 2000$ s), we ran the same grid of candidate sets as above. For every (n, k) pair, we report the median solving time; entries marked † reached the 180 s time-out with no feasible playlist.

k n	4	8	10
60	180.5†	69.8	71.2
80	180.8†	23.0	**1.5**
100	180.7†	**1.6**	14.2
500	180.3†	25.5	12.8

Table 5: Median solve times (seconds) for *time* $= 2000$ s under varying dataset size n and slot count k. † = timeout/UNSAT.

Table 5 shows a clear shift compared with the 1000 s benchmark. Too few slots ($k = 4$) make the duration window impossible. Increasing to $k = 8$ dramatically lowers solve time (23–70 s) once $n \leq 500$, as the extra flexibility lets Z3 satisfy the length constraint early. With $k = 10$, the model is often easiest: for $n = 80$ the solver finishes in ≈ 1.5 s, yet the same k is slower at $n = 60$ because the smaller pool offers fewer feasible combinations. The largest pool ($n = 500$) is still tractable when $k \geq 8$, but times climb above 20 s, showing the quadratic blow-up in Boolean variables ($n \times k$).

In short, doubling the required play-time proved better for longer playlists and at least eight tracks are now necessary for the solver to avoid timeouts under the given smoothness and popularity constraints.

6.4 Unsatisfiable Configurations

Some parameter combinations make it extremely difficult or impossible to satisfy all constraints. For example, with a small dataset ($n = 40$), a long target duration (3000 s), a short playlist ($k = 10$), and very strict smoothness thresholds ($\Delta_{\text{valence}} = \Delta_{\text{energy}} = 0.05$) and popularity requirement ($P_{\text{avg}} = 70$), the solver returned `unsat` after 31 seconds:

```
solving...
Z3 returned unsat in 31.13s
No feasible playlist found under the given parameters.
```

This confirms that excessive constraint tightness, especially with limited input diversity can quickly lead to infeasible instances. We log such cases to help users identify and relax conflicting parameters.

7 Conclusion

In this project, we developed a constraint-based tool for playlist generation using SMT solvers. The goal was to create emotionally smooth and popular playlists that respect user-defined constraints, such as duration, valence, and energy transitions, as well as average popularity. Our tool takes real Spotify-derived metadata as input and uses Z3 to find feasible playlists that satisfy all constraints.

We demonstrated that for moderate sizes, approximately 100-150 tracks and playlist lengths of up to 10, our tool performs well, often finding solutions in under a minute. We also ran a series of benchmarks that confirmed how sensitive solve time is to playlist length (k), transition thresholds, and the choice of track sample. For example, tight valence constraints and longer playlists tend to slow down the solver significantly or even lead to unsatisfiable results. On the other hand, relaxing thresholds, such as valence delta to 0.2, made most instances solvable within seconds.

We added several tricks to help with scaling, like pre-sampling tracks, simplifying constraints, and skipping rarely used genre constraints. These made a noticeable difference in reducing solve time and memory load.

That said, there is still a room for improvement. Beyond 150 input tracks or $k > 10$, the problem becomes significantly more challenging, and Z3 may time out. Also, our model doesn't currently support soft constraints for dynamic playlist lenghts. These would be interesting to explore in future work, along with integrating some user preference data or using ML embeddings to guide the search.

References

[1] Geoffray Bonnin and Dietmar Jannach. Automated generation of music playlists: Survey and experiments. *ACM Computing Surveys (CSUR)*, 47(2):1–35, 2014.

[2] Arthur Flexer and Thomas Grill. The problem of limited inter-rater agreement in modeling music similarity. *Journal of New Music Research*, 45(3):239–251, 2016.

[3] Shaurya Gaur and Patrick J. Donnelly. Generating smooth mood-dynamic playlists with audio features and knn. In *Artificial Intelligence in Music, Sound, Art and Design*, page 162–178. Springer, Cham, 2024.

[4] François Pachet and Pierre Roy. Automatic generation of music programs. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming — CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 331–345, Berlin, Heidelberg, 1999. Springer.

[5] Xinxi Wang, Yi Wang, David Hsu, and Ye Wang. Exploration in interactive personalized music recommendation: A reinforcement learning approach. *arXiv preprint arXiv:1311.6355*, 2013.