Efficient Algorithms For Music Engraving, Focusing On Correctness

Jeron Lau, Scott Kerlin
Mathematics, Statistics and Computer Science Department
Augsburg University
2211 Riverside Ave, Minneapolis, MN 55454
lauj@augsburg.edu, kerlin@augsburg.edu

Abstract

Designing a software algorithm for music engraving is a difficult problem. For the written score to be readable for instrumentalists and singers, the music must be presented in a way that is easy to follow. Currently existing music notation software often makes a compromise between a correctly engraved score and efficient algorithms. This paper covers better algorithms to use to properly layout notes, and other symbols within a measure. The algorithms do not become slower with larger scores, and all run at an amortized $O(n)$, where n is the number of notes in that section of the score. These algorithms are developed with an open source software library for handling the music layout and animations for music software applications in mind. The techniques explored include using custom data structures that work well for this purpose.

1 Definitions

This section of the paper describes music notation terms.

1.1 Stave

The *stave* is made up of a series of horizontal lines, typically five. Notes are placed on the stave.

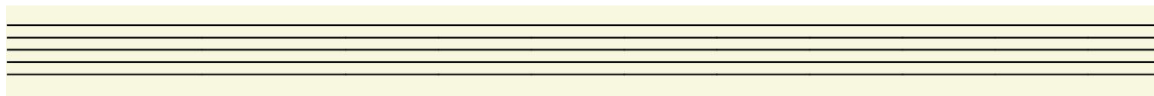

Figure 1: The common 5-line stave (Musical notation samples within this paper are generated by the engraver program developed during this research project)

1.2 Stave Space

A *stave space* is the space between two horizontal lines on the stave [1]. The stave space is a unit used in a similar manner to the em unit used in typography.

1.3 Note

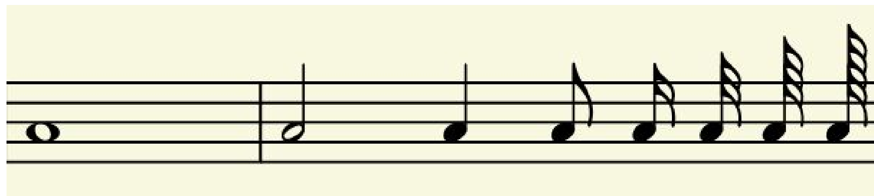

Figure 2: From left to right; Whole note, half note, quarter note, eighth note, 16th note, 32nd note, 64th note, 128th note

A *note* needs to convey three things: start time, duration and *pitch* (musical term for frequency of the sound) [5]. Start time is communicated by it’s horizontal position with respect to other notes and *rests* (silence). Duration is typically shown with either the *flag* or *notehead*, as shown in Figure 2, and pitch is typically shown with vertical position.

1.3.1 Flag

The *flag* of a note is used to divide the duration of a note in half. Multiple flags may be used to get shorter notes. Flags can only be used for notes shorter than a quarter note.

1.3.2 Beam

A *beam* is the joining of two flags as a single line.

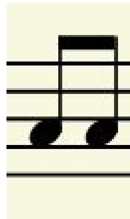

Figure 3 – Two eighth notes beamed

1.3.3 Notehead

Where the notehead is placed on the stave determines the pitch of the note. A hollow notehead is used for half notes and longer. Otherwise, a filled notehead is used.

1.3.4 Stem

The stem of a note is the vertical line that connects the flag (if there is one) to the notehead. Stems are not used in whole notes or breves (twice the length of a whole note).

1.4 Measures & Beats

A *measure* contains a fixed number of *beats* (commonly four). Beats are defined to be a certain duration (commonly a quarter note). Measures begin and end with *barlines*.

2 Stem Direction Decision

In order to make it easier for the person reading music notation, the stem direction is different when the note is placed above the middle stave line from when it's below the middle stave line. Notes on the middle stave line can have their stems point in any direction. The choice of which direction the stem oriented in can make it easier or harder to read the music. Shown below (Figure 4) is a flowchart for the "correct" orientation of the stem.

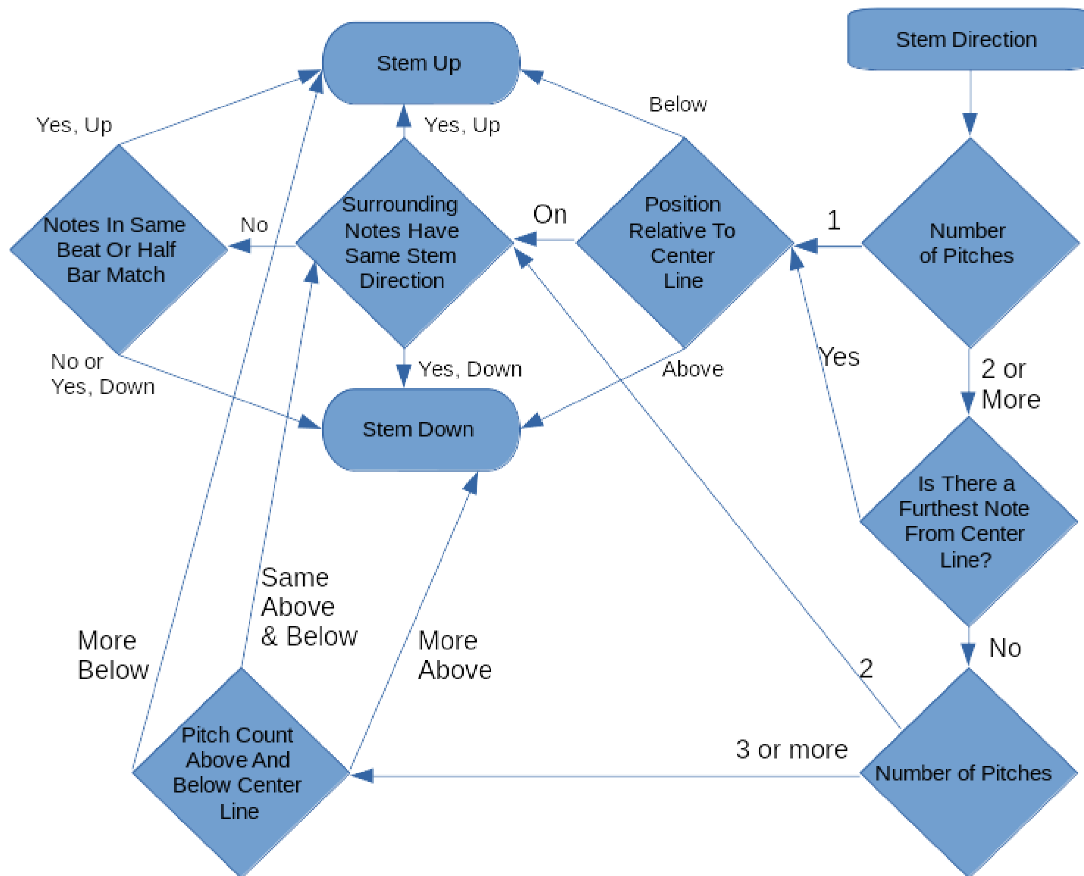

Figure 4: Stem direction decision algorithm flowchart compiled from a description in Elaine Gould's Behind Bars [1]. Note that this flowchart does not take into account beaming and rules for other unusual notations.

For the most part, the flowchart is straightforward to implement as an algorithm. The checking of the surrounding notes' stem directions could add complexity, though. The

algorithm may have to seek ahead or behind multiple times to determine direction. Fortunately, the algorithm won't have to go through the entire file in the worst case. It only needs to go until the end of the measure or the end of the beat, which usually doesn't contain many notes (almost always between 1 and 32). An amortized analysis of the algorithm would result in O(1). Discovering which note is furthest from the center line will grow in complexity with the number of pitches in the chord. Fortunately this is also very limited, as one player or piano hand reaches their limit before 8 pitches in a chord. Therefore, this algorithm can also be amortized to O(1).

3 Rhythmic Spacing Calculations

Spacing between notes is changed depending on the duration of the notes. Longer notes take up more space than shorter notes. This is done to improve readability. Usually, this is done as a condensed version of space being linearly related to duration. A whole note is given 7 arbitrary units in Elaine Gould's Behind Bars, but rather than a half note having 3.5, it's given 5 to reduce the amount of wasted space. The reduced space also makes it easier to read for musicians [1]. When laying out the music onto a page, the arbitrary units can change width depending on how to best fill the page, since the music on a page is expected to fill the entire page. The algorithm described in paper Algorithms and Data Structures for a Music Notation System based on GUIDO Music Notation [2] can be used to do the layout of measures on a page. Below is a description of the algorithm for calculating layout within a measure.

The eight durations with their respective arbitrary unit values don't cover every possible case (there are an infinite number of possible note durations). After many experiments, I found this kind of spacing unfortunately doesn't fit nicely into a quadratic, cubic or other curves. The error at many points ends up being at least ±1 no matter how you fit it. The solution is based on different ranges of durations; Do a linear interpolation on the normalized duration. We'll call this algorithm `get_spacing(duration)` that returns the arbitrary units as a floating point.

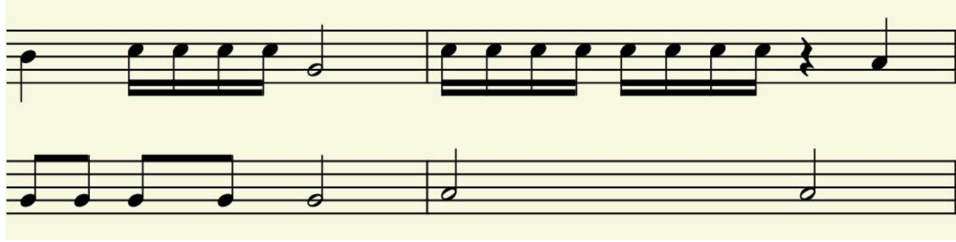

Figure 5: Output image using the described rhythmic spacing algorithm

The problem gets more complicated when the number of staves exceeds one. The algorithm engraves one measure at a time, all staves at once, because the spacing of a measure in one stave affects the spacing on others. The algorithm uses a priority queue to get the next note in time throughout all of the staves. Due to the relatively small number of staves in any given piece of music and the nature of what's being inserted into the queue, implementing the priority queue as a double ended queue using an array layout is actually faster than using a heap. This makes pops O(1), and inserts O(n). But, not all notes are going to be in the queue (unless it's all whole notes, which in that case the algorithm does not need to insert). The maximum number of notes in the priority queue is the number of staves. The only way the algorithm will go through that many items in the priority queue to insert is if all the next notes are at a different point in time (which is rare in writing, and splitting notes over beats and half measures is required for legibility, consistently adding notes that happen at the same point in time). Therefore, the amortized time of each insert operation is effectively O(1).

The algorithm as a whole goes through each note using the order specified by the priority queue. Using the priority queue's time analysis, the amortized time of the entire algorithm ends up being O(n) where n is the number of notes. Since each note needs to be iterated over to render the score, there is no added complexity to calculate the spacing of the notes in a multi-stave score, using the amortized analysis.

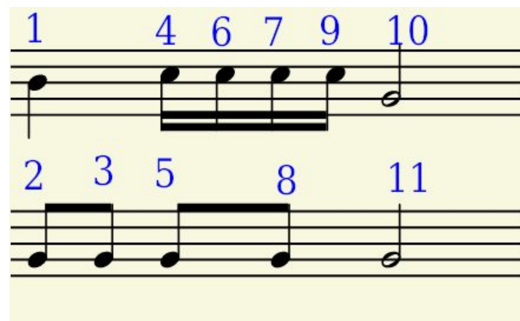

Figure 6: Rendering order for one measure based on priority queue

The priority values in the queue are the fraction of measure remaining to be engraved. In figure 6, the first two rendered notes have priority 1 (since nothing has been engraved yet), stave 1 is added back to the priority queue with priority 0.75 and stave 2 with priority 0.875. This causes two notes to be rendered on stave 2 before the renderer returns to stave 1. Between the 3rd and 4th rendered notes, the priority changes by 0.125 which is passed to `get_spacing()` to calculate the advance of the x coordinate. Between the 5th and 8th notes, there are two separate changes of priority, which makes the advance of the x coordinate between notes 5 and 8 on stave 2 the sum of `get_spacing()` with 0.625 for the first sixteenth on stave 1, and `get_spacing()` with 0.625 for the second sixteenth on stave 1, visually stretching the beat.

3.1 Pseudo-Code For Rhythmic Spacing Calculations

The following is pseudo-code for the *engrave_measure* algorithm:

```
# Start rendering horizontally at beginning of measure
let x := 0.0

# Remaining duration for all measures
let all := FULL_MEASURE_DURATION

# Add All staves to the priority queue
let priority_queue := new Deque
for each stave in score {
    priority_queue.push_back((FULL_MEASURE_DURATION, stave))
}

# Empty the queue
while !priority_queue.is_empty() {
    # Get next note to render
    let (time, stave) := priority_queue.pop()
    let note := stave.next()

    # Increment x, if it needs to be incremented
    if time < all {
        x += get_spacing(all - time)
        all = time
    }
    time -= note.duration

    # Render note
    render_note(x, note, stave)
```

```
        # Add back to queue if there's time remaining
        time = time - note.duration
        if time ≠ 0 {
            insert_into_priority_queue(priority_queue, (time, stave))
        }
    } # End while loop

    # Add the rest of the space for the measure.
    x += get_spacing(all)

    # Render barline (end of measure)
    render_barline(x)
```

4 Dynamic, Articulation, And Lyric Placement

Dynamics and *articulations* are markings used to convey the style of the music. Dynamics are used to show how strong or soft a note should be played. Articulations are used to convey how separated or connected the notes are. Dynamics, articulations and lyrics are placed in relation to the stave and notes, and do not conversely affect stave and note placement or measure width (their algorithms do not affect the rhythmic spacing calculations described above).

4.1 Dynamics

Dynamics are generally placed centered with respect to the notehead, directly below stave. They should be as close to the notehead as possible, but all other markings take precedence on nearness to the notehead. Dynamics may move to the left if other markings / notes are in the way [1] (such as the middle part of three on a stave, or ledger lines from the next note overlapping where it would go).

The music font may define a balancing point for the dynamics. This makes the dynamic appear more centered, as dynamics tend not to be symmetrical. According to the SMuFL (Standard Music Font Layout) specification, dynamics may define an `opticalCenter` attribute in the JSON font metadata file [4]. The `center_below_note_markings()` function uses this data if it is defined.

4.1.1 Dynamics Placement Algorithm

The algorithm (assuming the bounding box includes the margin):

```
dynamic->bbox := center_below_note_markings(next_note->bbox)

# If dynamic collides to the right, move left
if dynamic->bbox.collides(next_note->bbox) {
    let overlap := dynamic->bbox->x + dynamic->bbox->width
       - next_code->bbox->x
    dynamic->bbox->x := dynamic->bbox->x - overlap

    # If dynamic collides to the left, move down from original, and
    # from new position (and choose position that’s closest to note)
    ...
}
```

4.2 Articulation

Placement of articulation is simple. It’s always centered on the notehead, positioned above or below the stave, whichever is closest to the notehead. The exception is the strong accent marking, which is always placed above the stave [1].

4.3 Lyric Placement

Lyrics are always placed beneath the lowest marking on the stave. Longer words extend the width of the arbitrary units used for rhythmic spacing, and shorter words bring notes closer together to make it easier to read the text [1].

The following pseudo-code shows the algorithm used to determine if the notes need to be closer together, after calculating the new x coordinate.

```
if new_x > old_x + AVG_WORD_LENGTH_WIDTH {
    # Too much space between text
    new_x := old_x + AVG_WORD_LENGTH_WIDTH
}
```

5 Accidental Placement

Accidentals are modifiers on the pitch of a note (more granular than vertical position on the stave). They need extra space, so their correct placement needs a modification to the rhythmic spacing algorithm. For durations longer than a 16th note the full width of the measure can be altered to accommodate for space for the accidental. Shorter durations, and multiple accidentals require that the spacing becomes distorted to make room for the accidentals [1].

In order to accomplish this, the rhythmic spacing algorithm must do 2 passes to render the measure for all staves. The first pass calculates the number of arbitrary units, finding collisions of accidentals with the notes, and the second pass uses the amount of overlap to adjust the total width of the measure (stretching the original, so everything fits without colliding).

6 Beaming Rules

For multiple notes within the same beat, it is not acceptable to use flags. Rather, the notes should be beamed with each other. The angle of the beam is determined by the difference between outer notes. For instance, the beam is angled down, if the last note in the group is lower than the first note. For 2-3 notes the beam should not be angled more than one stave space vertically. For 4 or more notes the beam can be angled up to two stave spaces, but not more. The notes closest to the beam decide the angle for chords. The shortest stem is used to calculate both the y coordinate for the beginning and end of the beam [1].

The angle of the beaming depends on first choosing a stem direction (see 6.3), which makes this a multi-pass algorithm. Below is pseudo-code for a simplified version of the entire algorithm, putting the pieces together. Each pass runs at O(n) where n is the number of notes in the group, and there's a fixed number of passes, so the whole algorithm runs at O(n).

```
# Go through each note in the beam group, and find stem direction
let direction = beam_group.get_stem_direction()
```

```
# Create a beam as close as possible to noteheads
let beam = Beam(beam_group.first()->y + MIN_STEM,
    beam_group.last()->y + MIN_STEM)

# Move one side farther away, so that y doesn't differ too much
beam.limit_y_difference()

# Go through each note and calculate how much change is needed.  Get
# the maximum change and shift by it.
let shift := beam_group.amount_to_shift()
beam.shift(shift)
```

6.1 Beaming Rules For Simple, Compound And Complex Meters

In simple and compound meters, notes that are in different beats may not be beamed with each other. The exception is 4/4 and 4/8 time signatures where the first two beats can be beamed with each other, as well as the last two in each measure. The only difference to this with complex meters, is that the beats don't have a consistent duration.

6.2 Beaming Rules (Subdivisions)

For durations less than an eighth note, multiple beams may be used. Additional beams are placed further away from the notehead. Oftentimes, durations are mixed within a beamed group. In these cases, inner beams that do not connect (are not surrounded by any others of the same length) have the width of one notehead. A series of 32nd notes is beamed in groups of 4, attached only by the eighth note beam.

6.3 Beaming Rules (Stem Direction)

The stem direction of a beamed group is decided using a similar pattern to ordinary stem directions. The stem direction is based on the majority of the notes being above or below the middle stave line. If there is no majority, then the furthest note from the middle decides the stem direction. If notes are equidistant from the middle stave line, then it should match surrounding groups. If surrounding groups are equidistant as well, default to down stems [1]. There's an exception, though; If there's a minority of notes that are a great distance from the middle stave line, the stem direction should follow the minority.

This entire algorithm can follow a flowchart, and be implemented using the same kind of simple algorithm used for stem direction for non-beamed notes.

7 Slur, Bend, And Glissando Markings

Slurs are long curves drawn over multiple notes used to denote the phrasing (grouping of notes) in a part. Glissandos and bends are markings to continuously change the pitch of a note. This section covers engraving algorithms for markings that go through multiple notes.

7.1 Slur Line Generation

Lilypond, a music engraving program, uses a slur line generation algorithm that generates multiple slur lines and uses an “ugliness” rating algorithm to figure out which one is the least ugly (no overlaps, slur line rendered near notes), then uses that slur marking [3].

A simpler approach may be used. Once the tie has either been classified as going above or below the notes, the amount of clearance at each note is calculated. From there, a cubic curve is generated. The difference in the y coordinate between the first and second control points has a maximum. When the maximum is exceeded, the lower control point is moved up, as well as the begin and end coordinates of the slur in order to flatten it out.

7.2 Glissandos and Bends

Glissandos start after the notehead where they begin and stop before the notehead where they end, drawn at an angle. The vertical position on either end matches the vertical position of the notes. The exception is that when there are accidentals, the glissando should stop before the accidental, and not cover the entire vertical distance [1]. This algorithm can be simply drawing a single line, and if there is an accidental, the ending coordinates can be modified using linear interpolation to calculate the y coordinate from the x coordinate (which is calculated as the notehead’s x coordinate minus the width of the accidental). Bends are rendered the same as short glissandos that either don’t have a written starting position or a written stopping position.

8 Results And Future Work

The algorithms in music engraving appear to be able to be reduced to an amortized O(n) execution time. This means that music engraving software does not have to become slower with bigger scores. The resulting generated notation should be “beautiful” to enhance the reading experience of the players [3]. But, this research shows that that can be optimized to the point where you can see the final product as you are working as opposed to having a preview lag behind, as it does in GNU Denemo (using LilyPond).

8.1 Future Work

The number of allocations in the implementation of the algorithms discussed in this paper, could be reduced. This would reduce the constant in the execution-time. There are also many places where results could be cached, that currently are not.

The current implementation also does not include animations yet. Quick animations are essential for a natural-feeling user experience. Additionally, an actual user interface has yet to be designed for a music notation software depending on this code.

Not all musical elements that may appear in a score have been covered in this research, particularly notation in modern and experimental music. There’s also a fine-tuning of the kerning between musical symbols that needs to be done.

In the future, it would be useful to look into the different algorithms that LilyPond uses to make very “elegant”-looking scores, and see if those algorithms can be optimized, or if similar results can be achieved using faster algorithms. Modifying and adopting algorithms from LilyPond could lead to a more complete engraving library.

8.2 Conclusion

It appears that it is possible to make a music notation software with an amortized O(n) execution time (with n being the number of notes) for all rendering algorithms and still

have a correctly engraved and formatted score. A better, more responsive, user experience should be possible from existing music notation software for scores with a large number of parts and measures.

References

[1] Elaine Gould. *Behind Bars: the definitive guide to music notation.* Faber Music Ltd, 2011.

[2] Kai Renz. *Algorithms and data structures for a music notation system based on GUIDO music notation.* https://tuprints.ulb.tu-darmstadt.de/265/1/kai_renz_diss.pdf, 2002.

[3] The LilyPond development team. *Essay on automated music engraving.* http://lilypond.org/doc/v2.20/Documentation/essay.pdf, 2002.

[4] W3C. *Standard Music Font Layout (SMuFL) Version 1.3*. https://w3c.github.io/smufl/gitbook/, 2019.

[5] Dominique Vandenneucker. *Music Data Structures*. Arpege Music. http://www.music-software-development.com/music-data-structures.html, 2012.