MATH2340 Essay 2

Denys Parsons' Directory of tunes is a classification of melodies. What problems arise when attempting to place melodies in a systematic order, and what mathematical ideas can be used to solve them?

1 Introduction

Denys Parsons created a directory of tunes, to look up these tunes you use a *Parsons Code*; you create this code by thinking about some notes of the tune and whether they go up (u), down (d) or remain the same (r) relative to the last. The code may look like $*uurdddduurdr$ (the theme from Beethoven's Symphony No. 9, Ode to Joy). I have written an example piece of code that takes midi data and produces a parsons code and contour plot (see appendices).

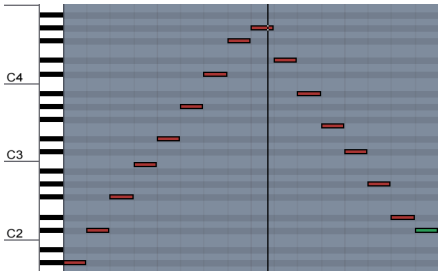


Figure 1: Piano roll representation of contour below

```
~/programming/uni/math2340 $ ./contour.py $(./miditoparsons.py test2.mid)
*uuuuuuuuddddddd
```

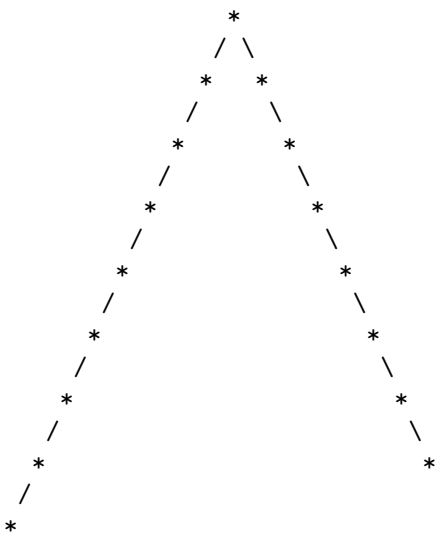


Figure 2: Script to convert single track monophonic midi to Parsons code and contour plot

This is an example of a way of classifying a tune in a way that can be looked up later, in this essay I will analyse Parsons code and other ways of achieving the goal of organising music in a systematic way for lookup later.

2 Discussion

The Parsons code is, in essence a human-usable hashing function for music (it's not very good notation [1]). A hashing function is defined as a function that maps data of any size to a fixed size. Essentially we're reducing the whole of the pitch range possible when written in music to the relation between each pitch when played in series. Hashes can be used to generate a data structure called a *hash table*. These are often used in computing for quick storage and retrieval of data. The Parsons book *The Directory of Tune and Musical Themes* is an example of a rudimentary hash table intended to be human-usable. One difference however is that hashing algorithms try to keep an even distribution of hash outputs; the Parsons code outputs are not at all even, in proceedings from Bridges Conference 2018, Andrew Crompton (Crompton, 2018) generated a 'chequerboard' that represented all places in which the codes in *The Directory* end up in *The Directory* showing that only one tune reached the -15 mark and only two the 15 mark (that is, 15 descending notes in a row).

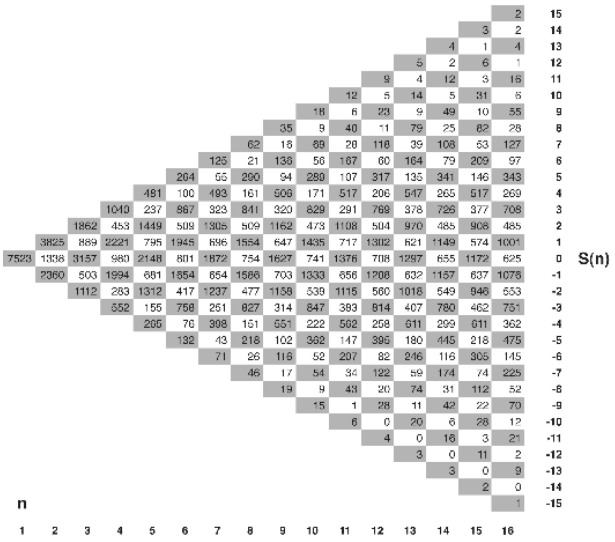
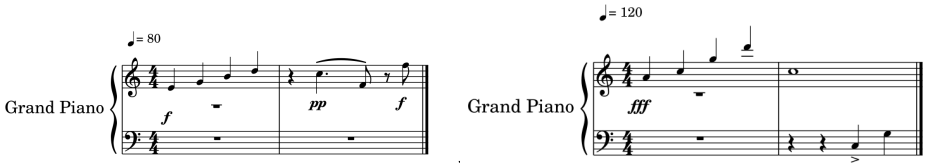


Figure 3: Chequerboard showing where each tune in Parsons code ends up (Crompton, 2018)

A hash table is constructed as a series of buckets each produced from the full hash. A hash is generated and then through some function reduced into a set amount of 'buckets' (usually by modulo as this easily controls the number of buckets). This could be thought of as the truncation to 16 notes in *The Directory*. The script in *appendix A* can be used as a hashing function for an indeterminate length of music, implementing the limit allows for the construction of the buckets. From a potentially infinite length of a Parsons code we allows ourselves 3^{15} buckets which is suitable as each of the tunes in *The Directory* are organised 'alphabetically' by their code. Further, this allows us to look up any code that is less than 16 long as each entry is ordered such that we can consider each tune in the book past the first occurrence of this code as a substring of the full code.

[1] As seen here: `https://www.youtube.com/watch?v=wY-OIMkHhMs`

A particularly nefarious composer could also manage to compose many works that sound different but encode to the same code as all the other pieces (similar to generating hash collisions). For example the two pieces here have the same Parsons code despite being very different sounding:




```
~/programming/uni/math2340 $ ./miditoparsons.py First\ Piece.mid
*uuuddu
~/programming/uni/math2340 $ ./miditoparsons.py Second\ Piece.mid
*uuuddu
```

If this method of encoding became popular it wouldn't be hard for composers to write pieces that fit with the codes of more famous pieces of work to get noticed when searching for the piece you were thinking of. This creates a problem as the hashing algorithm is easy to create collisions for, of course the idea isn't to be cryptographically secure but in a scenario in which any person could write a piece that appears in the music database it wouldn't be hard to make the database unusable by flooding it with collisions. This problem exists in hash table too, however hashing algorithms are more suitable when they are producing an even spread of values on the output. We've shown that this is less likely with music both because creatively we tend towards a tonal centre as in the chequerboard and that it's very easy to construct a hash collision as in the two pieces above.

Parsons has solved the issue of creating a systematic order of tunes by reducing the problem space to being only three points of notation and then limiting the length, this in essence creates a hash table. This allows humans to be able to create the code in their heads and look it up in the book. A study on 36 users of the parsons code found that 0% of users with zero background could make a successful query however. (Uitdenbogerd & Yap, 2003) And that only when professional users used the system did the number arrive at over 50%. This perhaps suggests that it's not useful as a means of looking up tunes.

Further solutions to this problem have been created, incorporating more and more musical elements. For example, the 'Kolta code' that encodes rhythmic information as "shorter, longer, or the same". This has the same number of possible unique encodings as the Parsons code (as they both use a set of magnitude 3 as their basis. However in the patent for this code (Kolta, 2008) he combines both codes to create the Kolta-Parsons code which instead uses 9 possible characters. These can then be used with algorithms used for DNA-matching to find music in a database.

This means of course that both codes are still as 'human-usable' (within the bounds of musical ability as discussed above) but combining them both allows for 9^{15} combinations, allowing for each element in the directory to be much more unique than either of the codes. However looking up a song in a book full of 9^{15} combinations may be more difficult for the human (which, is why this patent was focused on music database software probably).

Whilst both have the same time complexity, that is, $O(n)$ to find each song from its code, the actual time for the Kolta-Parsons codes will be a lot longer. The time complexity $O(n)$ results from a pure linear search through the book. $O(\log n)$ could be achieved via a binary search in both cases, as the book is ordered. Through a binary search approach, the proportions of each divide could be altered cleverly to get a result that is still worst case $O(\log n)$ but the average number of comparisons may be smaller. Hash tables are $O(1)$ on average, $O(n)$ at worst, with no collisions, however the book has a time complexity of finding the entry in a sorted list associate with it, if there are collisions in the book there's the possibility of it being $O(n^2)$ even, but for now we will assume there are none, and even if there were some the number of songs with the same code is going to be at most 10 to 15 in such a book.

Further notation forms could be developed either reducing the music down more to become more usable. For example in the Kolta code, removing the 'same length' part to make it just 'short,

or long' this makes a 2^{15} length list possible, which is a bit small but I'd suggest that knowing the genre of the song and even if there were three or four songs with the same pattern a human can recognise which one it is. Easily you could make a book of 32768 songs with two or three in each 'slot' and cover most of the less-engaged listener's repertoire of songs that they listen to. The problem for humans is mainly that of ease of use of the system rather than that of saturation of each string of code. As of right now I have 34598 pieces of music saved on my hard disk,that's around 700GB of data, the average listener will have much less. This perhaps shows that the amount of music that the average listener isn't enough to fill the above 9^{15} strings purely with encoding of melody and timbre. Of course, music is more than that. Not accounting for other musical elements like texture, time signature, dynamics etc.

An obvious issue with this system of notation is music without a melody or any appreciable beat, examples exist in contemporary music, ambient music, pieces with lots of polyphony or polyrhythms, and extreme forms of music like noise. These systems are unable to encode these sorts of music, and thus another system would have to be used. Further these systems break down with more modern electronic styles that utilise techniques like sampling, or electronic dance music which often sticks rigidly to a particular rhythm across the genre.

More modern systems methods are kept secret but most use a method called *acoustic fingerprinting* in which the audio recorded is analysed to find the "peak intensities" and those are used as a key in a hash table to find each song (Wang, 2010).

3 Conclusion

Through this essay I've analysed the problems that arise when trying to encode music using a particular element of it and reducing it down to an easier notation for humans to use and look up in a book. Through the analogy to a hash table in computer science I've been able to find and demonstrate issues with Denys Parson's code and further, the Kolta-Parsons code patented in 2008.

Whilst both of these systems are usable for pieces that adhere to, essentially, a western-focused and historical view of music they are lacking to be able to encode more contemporary types of music. Studies (Uitdenbogerd & Yap, 2003) also suggest that they may be hard to use for their intended audience (people with little to no musical ability). These systems, however do allow for music to be written down and ordered in a systematic way where applicable and so achieve their goals of doing so. Often more up-to-date systems will use fingerprinting and actual audio analysis meaning that humming the tune into a microphone is possible.

References

Crompton, A. (2018). The checkerboard of tunes.

Kolta, M. J. (2008). *Method for locating information in a musical database using a fragment of a melody.* (US Patent 7,962,530 B1)

Uitdenbogerd, A., & Yap, Y. (2003, 01). Was parsons right? an experiment in usability of music representations for melody-based music retrieval.

Wang, A. (2010, Sep). *How shazam works.* (`https://laplacian.wordpress.com/2009/01/10/how-shazam-works/`, Accessed 19-12-2019)

A Code Listing for Midi to Parsons code

```
#!/usr/bin/env python
"""
    Take a midi file and convert it to parsons code,
    with a limit and offset
"""
import mido
import sys

def midi_to_parsons(midifile, limit=16, offset=0):
    """
        Input: midifile (string) = A midi file path
               limit    (int)    = How long is the parsons code
               offset   (int)    = How many notes in do we start

        Output: parsons (string) = A string containing a parsons code
                                   length limit at an offset from the
                                   start of the file
    """
    count = 0
    parsons = ""
    for message in mido.MidiFile(midifile):
        if "note_on" in str(message) and offset == 0:
            if parsons == "":
                # initialise list
                prev = message.note
                parsons += "*"

            # simple comparison
            if message.note > prev:
                parsons += "U"
                prev = message.note
            elif message.note < prev:
                parsons += "D"
                prev = message.note
            elif message.note == prev:
                parsons += "R"
                prev = message.note

            #increment count
            count += 1
            if count >= limit:
                break
        elif "note_on" in str(message):
            offset -= 1
```

```
    return parsons

if __name__ == "__main__":
    print(midi_to_parsons(sys.argv[1]))
```

B Code Listing for Parsons Code to Contour Plot

```
#!/usr/bin/env python
""" Parsons code to contour plot """

import sys

def contour(code):
    """
        Input: code (string) = Parsons code starting with
                               "*" and containing "u, d, r"
                               as each letter

        Output: Prints code and then contour plot for given code
    """
    print(code)
    if code[0] != "*":
        raise ValueError("Parsons Code must start with '*'")

    contour_dict = {}
    pitch = 0
    index = 0

    maxp = 0
    minp = 0

    contour_dict[(pitch, index)] = "*"

    for point in code:
        if point == "r":
            index += 1
            contour_dict[(pitch, index)] = "-"

            index += 1
            contour_dict[(pitch, index)] = "*"
        elif point == "u":
            index += 1
            pitch -= 1
            contour_dict[(pitch, index)] = "/"

            index += 1
            pitch -= 1
            contour_dict[(pitch, index)] = "*"

            if pitch < maxp:
                maxp = pitch
        elif point == "d":
            index += 1
            pitch += 1
            contour_dict[(pitch, index)] = "\\"
```

```
            index += 1
            pitch += 1
            contour_dict[(pitch, index)] = "*"

            if pitch > minp:
                minp = pitch

    for pitch in range(maxp, minp+1):
        line = [" " for _ in range(index + 1)]
        for pos in range(index + 1):
            if (pitch, pos) in contour_dict:
                line[pos] = contour_dict[(pitch, pos)]


        print("".join(line))

if __name__ == "__main__":
    contour(sys.argv[1])
```