CONSTRUCTING PROXIMITY GRAPHS TO EXPLORE SIMILARITIES IN LARGE-SCALE MELODIC DATASETS

Chris Walshaw

Department of Computing & Information Systems,
University of Greenwich, London SE10 9LS, UK
c.walshaw@gre.ac.uk

ABSTRACT

This paper investigates the construction of proximity graphs in order to allow users to explore similarities in melodic datasets. A key part of this investigation is the use of a multilevel framework for measuring similarity in symbolic musical representations. The basis of the framework is straightforward: initially each tune is normalised and then recursively coarsened, typically by removing weaker off-beats, until the tune is reduced to a skeleton representation with just one note per bar. Melodic matching can then take place at every level: the multilevel matching implemented here uses recursive variants of local alignment algorithms, but in principle a variety of similarity measures could be used. The multilevel framework is also exploited with the use of early termination heuristics at coarser levels, both to reduce computational complexity and, potentially, to enhance the matching qualitatively. The results of the matching algorithm are then used to construct proximity graphs which are displayed as part of an online interface for users to explore melodic similarities within a corpus of tunes.

1. INTRODUCTION

1.1 Background

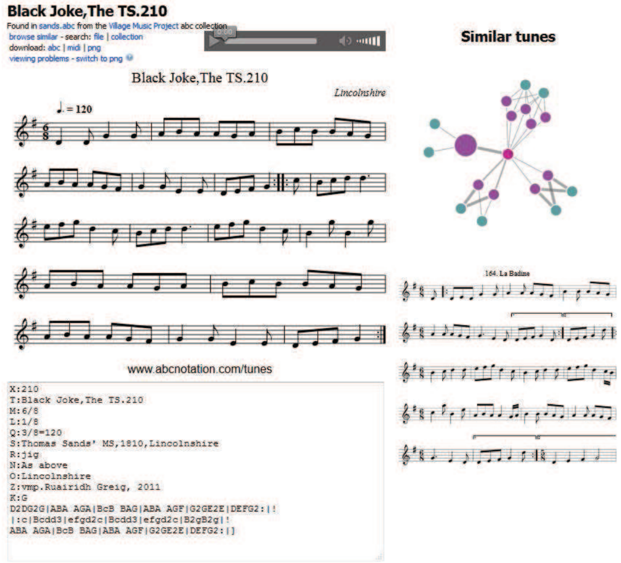
This paper presents an investigation into constructing proximity graphs using a multilevel melodic similarity metric. The resulting graphs are displayed as part of an online interface for users to identify related tunes, in particular, those found within the abc notation music corpus.

Abc notation is a text-based music notation system popular for transcribing, publishing and sharing music, particularly online. It was formalised and named by the author in 1993 and since its inception he has maintained a website, now at abcnotation.com, with links to resources such as tutorials, software and tune collections.

In 2009 the functionality of the site was significantly improved with an online tune search engine which currently indexes over 500,000 abc transcriptions, mostly folk and traditional music, from across the web. Users of the tune search are able to view, listen to and download the staff notation, MusicXML, MIDI representation and abc code for each tune, and the site currently attracts around half a million visitors a year.

In 2014 the search was enhanced with the introduction of TuneGraph, an online visual tool for exploring melodic similarity, [1]. TuneGraph uses a similarity measure to derive a proximity graph representing similarities within the abc notation corpus backing the search engine. From this a local graph is extracted for each vertex, aimed at indicating close variants of the underlying tune represented by the vertex. Finally an interactive user interface displays each local graph on that tune's webpage, allowing the user to explore melodic similarities.

A typical page display, is shown in Fig. 1, with the tune in standard notation, the MIDI player, the abc notation and the TuneGraph of close variants (top right). One of the close variants has been selected by the user (the vertex is enlarged) and is displayed below by the TuneGraph viewer (bottom right).



Black Joke,The TS.210
Found in sands.abc from the Village Music Project abc collection
browse similar - search: file | collection
download: abc | midi | png
viewing problems - switch to png

```
X:210
T:Black Joke,The TS.210
M:6/8
L:1/8
Q:3/8=120
S:Thomas Sands' MS,1810,Lincolnshire
R:jig
N:As above
O:Lincolnshire
Z:vmp.Ruairidh Greig, 2011
K:G
D2DG2G|ABA AGA|BcB BAG|ABA AGF|G2GE2E|DEFG2:|!
|:c|Bcdd3|efgd2c|Bcdd3|efgd2c|B2gB2g|!
ABA AGA|BcB BAG|ABA AGF|G2GE2E|DEFG2:|]
```

Figure 1. An example of a tune page.

A problem with the initial version of TuneGraph is that the similarity measure used to assess the proximity of variants is based on the incipit only (first three bars, neglecting any anacrusis). Of course not all closely related incipits result from closely related tunes, so this paper considers a different similarity measure which uses a multilevel representation of each tune in its entirety.

The introduction of this new representation has led to an investigation into the construction process for these graphs and a much better understanding of the parameters involved. That investigation is presented here.

1.2 Organisation

The rest of the paper is organised as follows. The multilevel paradigm is not (yet!) accepted as a valuable tool in the symbolic music analysis toolkit so section 2 presents a rationale. In section 3 the multilevel matching implementation, and its use in the construction of the proximity graphs, is discussed: this includes two recursive variants of local alignment algorithms and a similarity measure adapted to handle their globalised nature. Experimentation and results follow in section 4 and finally, in section 5, conclusions are presented.

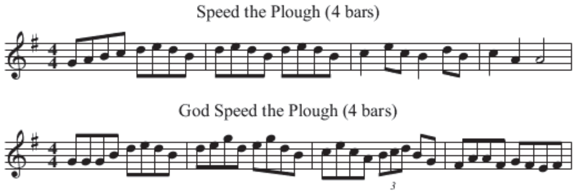


Figure 2. Two tune variants for Speed the Plough.

2. MULTILEVEL MATCHING: RATIONALE

Fig. 2 shows two versions of the first 4 bars of Speed the Plough, a tune well-known across the British Isles (at the time of writing the abcnotation.com tune search has 277 tunes with a title which includes the phrase "Speed the Plough", of which 157 are exact electronic duplicates. The first version in Fig. 1 is drawn from an English collection and the second, with the title "God Speed the Plough", from an Irish collection. Clearly these tunes are related but with distinct differences, particularly in the second and fourth bars.

It is typical in tunes like this that the emphasis is placed on the odd numbered notes, and in particular the first note of each beam. The strongest notes of the bar are thus 1 and 5, followed by 3 and 7.

To capture this emphasis when matching tune variants it might be possible to use some sort of similarity metric which weights stress (so that matching 1st notes carry more importance than, say, 2nd notes, e.g. [2]). However, in this paper the approach is to build a multilevel (hierarchical) representation of the tunes.

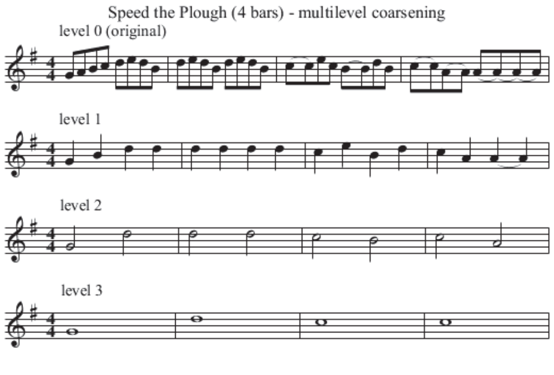


Figure 3. Multilevel coarsening of Speed the Plough

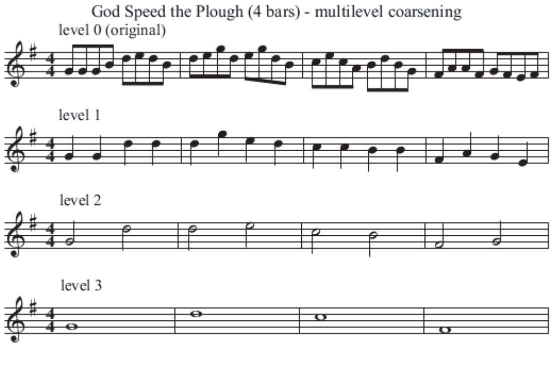


Figure 4. Multilevel coarsening of God Speed the Plough

Figs. 3 & 4 show multilevel coarsened versions of the original tunes, where the weakest notes are recursively replaced by removing them and extending the length of the previous note by doubling it.

At level 0, i.e. the original, the tunes are quantised to show every note as a sixteenth note, thus simplifying the coarsening process. In addition the triplet in bar 3 of "God Speed the Plough" is simplified by representing it as two eighth notes, the first and last notes of the triplet.

To generate level 1, the 2nd, 4th, 6th and 8th notes are removed from each bar; for level 2, the original 3rd and 7th notes (which are now the 2nd and 4th) are removed; for level 3, the original 5th note (now the 2nd) is removed. As can be seen, as the coarsening progresses the two versions become increasingly similar and thus provide a good scope for melodic comparisons which ignore the finer details of the tunes.

3. IMPLEMENTATION

This section discusses in detail the construction of the proximity graphs. The implementation is mostly straightforward. Each tune is initially normalised & quantised (section 3.1) and then recursively coarsened down to a skeleton representation with just one note per bar (section 3.2). Melodic matching can then take place at every level (section 3.3) using a melodic similarity measure. A proximity graph is induced by the similarity measure (section 3.4) which is then sparsified (section 3.5). Finally section 3.6 discusses how the multilevel framework is used.

3.1 Normalisation

As part of the normalisation process, each tune is cleaned of grace notes, chords and other ornaments. Generally most tunes under consideration from the abc corpus are single-voiced, [1], but if not, only the first voice is used for the matching.

Next, each tune is quantised so that longer notes are replaced with repeated notes (e.g. a half note is replaced with 4 eighth notes); more details can be found in [1].

3.2 Coarsening

The coarsening works by recursively removing "weaker" notes from each tune to give increasingly sparse representations of the melody. In the current implementation the coarsening strategy considers that the weaker notes are the off-beats or every other note and it is these which are removed (see Figs. 3 & 4). However, it should be stressed that the multilevel framework is not tied to a particular coarsening strategy and any algorithm that can be used (preferably recursively) to reduce the detail in the melody could be used in principle. For example, it should even be possible to use something as complex as a Schenkerian reduction, [3]; conversely many multilevel algorithms in other fields successfully use randomised coarsenings, [4].

Coarsening progresses until there is one note remaining in each bar; it would be possible to take it further, coarsening down to one single note for a tune, but experimentation suggests that the bar is a good place to stop.

Exceptions to the "remove every other note" rule are handled with heuristics, typically for tunes in compound time. Thus for jigs in 6/8, 9/8 & 12/8, which are normally written in triplets of eighth notes, the weakest notes are generally the second of each triplet. The same applies for waltzes, mazurkas and polskas in 3/4, so that for 3 quarter notes in a bar, the weakest is generally the second. The heuristics for dealing with these, and other less common time signatures, are discussed in [1].

3.3 Similarity Measure

Once the multilevel representation is constructed a variety of methods could be used to compare tunes at each level. This is a strength of the multilevel paradigm which is not reliant on a particular local search strategy, [4].

In a recent comparison study Janssen *et al.*, [5], suggest that one of the best similarity measures for finding melodic segments in a corpus of folk songs is local alignment. Meanwhile in previous work the longest current substring (LCSS) was used successfully within a multilevel context for melodic search, [6] (in fact, LCSS is just a special case of local alignment – see section 3.3.2). Therefore, in this paper recursive versions of both local alignment and LCSS are compared (although unlike Janssen *et al.* local alignment is applied to intervals rather than pitches, making it transposition invariant).

3.3.1 Local alignment (LA)

Local alignment is a well-known technique originating from molecular biology. Given two strings it finds the optimal alignment for two sub-sequences of the originals. The algorithm does not require the aligned sub-sequences to match exactly and makes allowances for gaps and substitutions. For example the strings `***abcde**` and `*acfe****` (where the asterisks represent non-matching entries) could potentially be aligned between `a` and `e` with a gap at the `b` and the substitution of `d` for `f`. Gaps, otherwise known as insertions and deletions, and substitutions are penalised with weights.

The algorithm is known as local alignment (LA) because, unlike the global alignment algorithms which preceded it, mismatching sub-strings from either side of the alignment are not penalised (i.e. in the example the string of non-matching entries, indicated by asterisks, could be arbitrarily long without changing the alignment score).

To compute the optimal local alignment for two strings of length m & n, an (m+1) x (n+1) score matrix *A* is constructed with the top row and left hand column initialised to zero. The remainder of the matrix is then filled using

$$A(i,j) = \max \begin{cases} A(i-1,j-1) + s(X_i, Y_j) \\ A(i,j-1) + W_{\text{gap}} \\ A(i-1,j) + W_{\text{gap}} \\ 0 \end{cases}$$

$$s(X_i, Y_j) = \begin{cases} W_{\text{match}} & \text{if } X_i = Y_j \\ W_{\text{substitution}} & \text{if } X_i \neq Y_j \end{cases}$$

where W_{match}, $W_{\text{substitution}}$ and W_{gap} represent the weights for a matching or substituted entry or a gap in the aligned sequences. The implementation discussed here follows Janssen *et al.* and uses $W_{\text{match}} = 1$, $W_{\text{substitution}} = -1$ and $W_{\text{gap}} = -0.5$.

This algorithm was introduced by Smith & Waterman, [7]. In fact their original scheme is a little more computationally involved but the scheme above is widely used and is the variant tested by Janssen *et al.*

To calculate the alignment score, and hence the qualitative similarity, the above scheme suffices. However to determine the aligned sub-sequences a traceback procedure is required. The traceback is implemented by recording a matrix of DIAG, UP or LEFT pointers for every entry of the score matrix indicating where the maximum value originated. If the maximum value is zero an END pointer is stored.

The traceback starts at the pointer matrix entry corresponding to the maximum score found and then tracks back through the pointers, terminating when it reaches an END. Diagonal moves indicate contiguous values in the two aligned sub-sequences whilst left or up moves indicate gap in one of them.

3.3.2 Longest Common SubString (LCSS)

The longest common substring algorithm operates in a similar fashion to local alignment filling in an (m+1) x (n+1) matrix of alignment values. However, because there is no need to allow for gaps, no traceback is required: the position of the maximum score in the matrix indicates the end of the longest common substring and the value of this entry gives its length.

In fact it is easy to see that, if the local alignment weights $W_{\text{substitution}}$ and W_{gap} are sufficiently large, so that gaps and substitutions can never occur in an optimal alignment, then the LCSS algorithm is just a special case of local alignment.

From here on, therefore, both algorithms, LA and LCSS, will be referred to collectively as local alignment, the main distinction between the two being that LCSS produces exact matching aligned substrings, is faster to compute and requires less memory (there is no need to use a full matrix and a memory efficient version exists which just repeatedly swaps a pair of arrays, one containing the row under calculation and one containing the previous row). Conversely, LA is more computationally complex and more memory intensive (if the traceback is required to identify the sub-sequences), but will generally match longer sub-sequences. Using $W_{\text{match}} = 1$, the similarity measures or alignment scores that either algorithm produces represent the length of the sub-sequences aligned, although in the case of LA there may also be penalty weights for gaps and substitutions so that, for example, the matching of `abcde` with `acfe` has a score of 1 – ½ + 1 – 1 + 1 = 1½.

3.3.3 Recursive local alignment = global alignment

A problem with using LCSS, and to a lesser extent LA, is that they are local. For example, using LCSS, `ab**ba` has exactly the same alignment score (of 2) when matched with `**ab` and with `ab**ba`, even though the latter seems a far better match. This is because the second match (`ba`) is not accounted for.

This was less of an issue in the predecessor to this paper, [1], where LCSS was used in a multilevel melodic search algorithm, since search algorithms are typically trying to find the best matches of a short phrase in a dataset of complete melodies. However for matching it is crucial to distinguish between tunes which match well across their entire length and those which perhaps only match for a short segment.

Interestingly Smith & Waterman touch on this in their original paper where they say "the pair of segments with the next best similarity is found by applying the traceback procedure to the second largest element of [the matrix] not associated with the first traceback", [7]

Unfortunately, just working from the existing matrix may lead to overlapping local alignments, but instead local alignment may be applied recursively as follows: when applied to two strings, S1 and S2, local alignment splits both into three substrings S1 = L1 + A1 + R1 and S2 = L2 + A2 + R2, where A1 and A2 are the aligned substrings (exact matches for LCSS or potentially with gaps and substitutions for LA), L1 and L2 are the left hand side unmatched substrings and R1 and R2 are the right hand side unmatched substrings (where any of the these unmatched substrings may be of length 0). Thus, having found A1 & A2 and split S1 & S2, local alignment can then be applied to L1 & L2 and to R1 & R2.

This procedure continues recursively, terminating when no alignment is found, or one or both lengths of the substrings being aligned are 0. For example, if the start of S1 is aligned with the end of S2 no further recursion is possible as the lengths of L1 and R2 are 0.

This recursion effectively turns the local alignment algorithms LCSS or LA into a globalised similarity measure, giving an alignment score along the length of both strings being compared. Henceforth these **R**ecursive algorithms will be referred to as RLCSS and RLA.

3.3.4 Biased recursive local alignment

An issue that became apparent when using recursive alignment, is that the algorithm makes no distinction between one long aligned sequence and several shorter ones. For example (using RLCSS) `abcd****` has the same alignment score (of 4) when compared with `abcd****` and with `**a**b**c**d**`, even though the former seems a good match and the matching with the latter is essentially noise.

To address this, the similarity measure is biased towards longer aligned sub-sequences by taking the 2-norm (square root of the sum of squares) of the alignment scores found by the recursive local alignment. In the above example this means that the **biased recursive local alignment** score is $\sqrt{4^2} = 4$ when matching `abcd****` with `abcd****`, whereas when matching with `**a**b**c**d**` it is $\sqrt{1^2 + 1^2 + 1^2 + 1^2} = 2$. Space precludes detailed empirical evidence of the effect of this biasing but it made a huge difference to the accuracy of the matching in terms of removing false positives from the results (see also section 3.4 for typical impact).

This biased recursive local alignment thus gives a measure, S_{XY}, expressing the similarity two arrays of intervals X and Y, each representing a tune.

3.4 Constructing the fundamental proximity graph

Neglecting the multilevel framework for now, this similarity measure, S_{XY}, induces a complete weighted graph on the dataset, where the edge weight between each pair of melodies is given by the similarity. Subsequently, when the graphs are displayed, edge thickness is shown in proportion to the weight with similar vertices joined by thick edges and dissimilar ones by thin edges.

However, most edges in the graph will have very small weights as most melodies in the dataset are only similar to a few others. At this point, therefore, it makes sense to restrict the graph to include only edges for tunes which are reasonably close matches. This graph is referred to henceforth as the **fundamental proximity graph** (FPG). (The FPG has an analogue in search: rather than presenting the whole dataset, ordered by increasing distance, typically search results will be restricted to a subset of "reasonably similar" results with some cut-off after which more dissimilar results are not shown.)

This restriction could be achieved in a variety of ways but here it is assessed by a **fundamental matching threshold**, T, and edges between melodies are only included in the FPG if they match across at least some proportion T of their length. More specifically an edge between vertices V_x and V_y is excluded if

$$S_{XY} < \max(\text{length}(X), \text{length}(Y)) * T.$$

As an aside, when calculating using this threshold it is also possible to use the minimum length but this results in very short tunes (such as fragments, included in the dataset as examples) matching with many other tunes and their corresponding vertices having very high degree.

Typical values for T in the experiments are 1/2 (very restrictive, excludes almost all edges), 1/3, 1/4, 1/6 and 1/8 (fairly inclusive, allows a lot of false positives). Note that there is no reason for this to be a simple fraction and T could just as easily be set to, say, 0.40 or 0.317; fractions are simply used as they tend to be more expressive.

Note it is not the intention in this paper to determine a definitive value for T (even if such a value exists). In an ideal world this would be a user chosen parameter and in principle it should be possible to set some range of values, e.g. T in the interval [0.125, 0.5], which the user could adjust according to their needs (provided that the lower value is not too small to make the calculation intractable – if set to 0, every edge is included and the fundamental proximity graph is a complete graph).

Note it is not the intention in this paper to determine a definitive value for T (even if such a value exists). In an ideal world this would be a user chosen parameter and in principle it should be possible to set some range of values, e.g. T in the interval [0.125, 0.5], which the user could adjust according to their needs (provided that the lower value is not too small to make the calculation intractable – if set to 0, every edge is included and the fundamental proximity graph is a complete graph).

The use of *biased* recursive local alignment does obscure what these fractions imply exactly, as it is no longer a case of adding up all the recursively aligned scores. To analyse this further consider that a large proportion of melodies in the dataset are 32 bar tunes in an AABB format. This is very typical in western European folk music and usually means that the tune is written as 16 bars, AB, with repeat markers at the end of each section. For a reel in common time this would be quantised as 8 eighth notes per bar or a total of 16 x 8 = 128 notes (strictly speaking 127 intervals).

So if T is set to 0.5 then, when using RLCSS, to be included two tunes would need to match exactly across at least half the tune (8 bars or 64 notes).

If T is set to 0.25 then they would need to match exactly across one a quarter of the tune (4 bars or 32 notes). Alternatively, again with T set to 0.25, they could match across four segments, each two bars (16 notes) long (in this case $S_{XY} = \sqrt{16^2 + 16^2 + 16^2 + 16^2} = \sqrt{1{,}024} = 32$); in other words a total of 64 notes or half the tune.

A similar analysis for T = 0.125 shows that the edge can be included if the tunes match exactly over at least:

- a single 2 bar segment (16 notes or an eighth of the tune); or
- four segments, each 1 bar long (so a total of 32 notes or a quarter of the tune); or
- sixteen segments, each ½ a bar long (so a total of 64 notes, or half the tune).

and obviously many other combinations are possible.

This gives a sense of the impact of the biased recursive local alignment: the matching can occur over a single long phrase or several shorter phrases, but for the latter the total length of the matching substrings will be longer.

Using RLA the picture is more difficult to analyse: for any pair of tunes, the aligned sub-sequences will typically be longer than RLCSS (because of the inclusion of gaps and substitutions) but similarity scores will be lower, because of the penalties. In practice, it seems possible to use higher values of T (e.g. 1/2, 1/3 and 1/4) to generate the fundamental proximity graph (see section 4.1.1).

3.5 Constructing proximity graphs for users

In fact the fundamental proximity graph is never actually constructed, although a sparsified version is. Ultimately the aim is to create a local proximity graph for each tune showing the closest matching variants. There are practical restrictions on the sizes of graphs that can be easily displayed by the website and assimilated by the user, leading the earlier work on TuneGraph to focus on the size/density of the local graphs and to favour those with no more than 40 vertices, [1].

The use of the FPG does help a great deal towards that end but, as will be seen (later, in Table 1), for some settings of T, it can still result in some vertices with a large number of neighbours (vertex degree) and consequently some very large local graphs.

To reduce some of these (and simplify the construction algorithm as compared with the previous TuneGraph paper which uses iterative bisection), each vertex is compared with every other vertex and only a fixed number of the closest neighbours which also pass the matching threshold are used to create edges in the **sparsified proximity graph** (SPG). The parameter controlling this is D, the **maximum included degree**, so that each vertex adds a maximum of D edges into the graph.

For many vertices there will be no neighbours which pass the matching threshold (i.e. no sufficiently similar tunes) but some will end up with significantly higher degree than D (since, although a vertex V may only match with a maximum of D neighbours, many other vertices could match with V). Therefore a further sparsification step takes place (as described in [1]) traversing the list of SPG edges (sorted in decreasing order by combined degree of the incident vertices) and removing any edge if both of its incident vertices have degree greater than a pre-specified **minimum sparsification degree**, *S*.

The previous TuneGraph paper focussed heavily on the choice of D and S putting the emphasis on the size/density of the local graphs probably at the expense of the data that they contain: potentially the local graphs can be made very rich in structure by matching tunes that are not very similar. Here, instead, by ensuring that the edges of the sparsified proximity graph are a subset of those from the fundamental proximity graph, the aim is to create local graphs that are both visually manageable (by sparsifying those which are not) and which do not contain a lot of spurious edges representing dissimilar tunes. Therefore, although considerable experimentation has been carried out with D and S (especially since the introduction of the simplified sparsification algorithm), none of that experimentation is presented here and for all the results they are set to D = 6 and S = 4.

Finally note that the construction of the SPG is essentially a post-processing cleanup operation which aims to eliminate any vertices of high degree so that the graphs are easy for users to assimilate and understand. In fact, experimentation in section 4.1.1 shows that for the more restrictive settings of T the FPG could be used in place of the SPG with no cleanup necessary (for example for RLA with T = 1/2 the maximum degree of vertices in the SPG is 37 and for RLCSS with T = 1/4 it is just 16).

3.6 Using the multilevel framework

It should be clear by now that constructing the sparsified / fundamental proximity graph is a vast computation. Even for the small test dataset used in the experiments with N = ~5,000 tunes, it potentially involves ~12,500,000 pairwise comparisons, i.e. ½ N(N-1) and, if every tune were 16 bars long (128 eighth notes), each comparison involves filling in a 128 x 128 matrix (16,384). So in total 3,200,000,000 calculations and that is without using recursion for the local alignment, which could easily double the total. For the full dataset, which currently has N = ~187,000 tunes, the complexity is astronomical.

As previously, [1], a straightforward way to cut this down pragmatically is to segment the dataset according to meter, so that tunes are only compared with others in the same meter. In the small test dataset the largest group (which dominates the calculation) then contains ~1,500 tunes in 6/8 resulting in 1,125,000 pairwise comparisons. For the full dataset the largest group contains ~56,000 tunes in 4/4 which is close to being intractable, but fortunately the multilevel framework can assist here by computing similarity scores at all levels of the multilevel representation, coarse to fine.

At first sight this might seem to increase the computational complexity but the interval arrays are much smaller at the coarsest level than the original. For a typical 16 bar score of a 32 bar tune the arrays will be 16 entries long at the coarsest level rather than the 128 in the original. If the coarse level matching can detect that a pair of tunes does not match, that edge can be excluded from the SPG at the cost of filling in a 16 x 16 matrix (256 entries) as opposed to the 128 x 128 matrix (16,384 entries), a 64-fold saving.

To that end the multilevel similarity calculation uses **level matching threshold**, T^l, and the multilevel matching is terminated *at any level* if

$$S^l_{XY} < \max(\text{length}(X^l), \text{length}(Y^l)) * T^l$$

where X^l / Y^l are the interval arrays for tunes X and Y at level l of the multilevel representation and S^l_{XY} is the biased recursive local alignment measured between them.

Obviously some matches which should actually be included in the FPG may be filtered out at a coarse level (i.e. those comparisons which fail the level matching threshold at one or more levels but pass the fundamental matching threshold). Therefore the level matching threshold, T^l, needs to be used with caution and should be more conservative than T (obviously there is no point making T^l larger than T as it would then take precedence at the finest level). Section 4.1.2 conducts some experiments into how these parameters interact.

This approach is referred to as **multilevel filtering (MLF)**: the multilevel similarity scores, S^l_{XY}, are computed and (as timings show in section 4.1.2) are used extensively to filter out dissimilar matches. However, the S^l_{XY} are discarded for $l > 0$ (i.e. all but the finest level) and the similarity between a pair of tunes is just the score, S_{XY} (= S^0_{XY}), from the original representation.

Another way to use the multilevel framework, alongside the filtering, is to sum the similarity scores, S^l_{XY}, at each level to give a multilevel similarity score, $\Sigma_l S^l_{XY}$, and to use this when weighting edges. This approach was used successfully for searching the dataset, [6], and is referred to here as **multilevel weighting (MLW)**. No empirical evidence is presented here that this approach is successful – it is rather a matter of opinion as to whether the multilevel representation is a meaningful reduction of the tune (although the effective use of the technique in search results, [6], and the success of the multilevel filtering in section 4.1.2 suggest that it may be).

Finally, if the multilevel representations are not used the matching framework is referred to as **single level (SL)**.

4. EXPERIMENTATION

4.1 Results – Test Dataset

The initial experimentation uses a small subset of the full abc corpus consisting of the 5,638 abc transcriptions taken from the Village Music Project[1], a collection of English social dance music mostly transcribed from handwritten manuscript books in museums and library archives. Of these 30 are removed due to implementation limitations (see [1]) leaving 5,608.

4.1.1 Fundamental Proximity Graph

The first experiments are to determine the characteristics of the fundamental proximity graph (FPG). Recall from section 3.4 that the FPG only includes edges between two vertices (tunes), V_X and V_Y, if the similarity score for the interval arrays which represent them, X and Y, is greater than some fraction, T, of the length the larger array.

Local alignment	Matching Threshold, T	Non-isolated vertices	Degree	
			Avg.	Max.
RLA	1/4	3,907	63.89	738
	1/3	3,206	18.49	441
	1/2	1,923	1.06	37
RLCSS	1/8	4,436	17.26	253
	1/6	2,812	1.8	23
	1/4	1,800	0.86	16

Table 1. Characteristics of the fundamental proximity graph for the test dataset.

Table 1 shows the results for different values of T and both local alignment algorithms, RLA and RLCSS, in terms of the number of non-isolated vertices (those with at least one edge), and the average and maximum degree. Obviously the smaller the value of T, the more edges are included and so the more dense the graph (i.e. the higher the average degree). As mentioned in section 3.4, ideally the user would be allowed to control the value of T to determine dynamically the restrictiveness of matching and consequently the size/shape of the local graphs.

No direct comparison between RLA and RLCSS is possible but one feature that is immediately apparent from the table is that they induce somewhat different structures on the dataset. Compare, for example, RLA with T = 1/3 against RLCSS with T = 1/8: both have similar average degree values (18.49 versus 17.26) and hence a similar number of edges but RLA has fewer non-isolated vertices (3,206 versus 4,436) and consequently a much higher maximum degree (441 versus 253). The same features can be observed for RLA with T = 1/2 as compared with RLCSS with T = 1/4 (both have an average degree close to 1).

This proves nothing but does suggest that at a specific graph density, RLCSS connects up more of the vertices.

Finally the previous work on TuneGraph, [1], suggested that, subjectively, the ideal size for the local graphs displayed to users is a maximum of ~40 vertices with a preferred size of ~20. Local graphs typically include two levels of separation so if the average degree of vertices is 20, say, there could potentially be 20 x 20 = 400 vertices in the average local graph. On the other hand, in reality many vertices in the local graphs are connected (for example, if a vertex of degree 20 is part of a clique then its 20 neighbours will all be connected to each other and so its local graph will only contain 21 vertices). However, this does suggest that the minimum values for the matching threshold should be no less that T = 1/3 for RLA and no less than T = 1/8 for RLCSS, so that the average degree does not rise above 20.

[1] See http://village-music-project.org.uk/

At the opposite end of the scale, the maximum values for T should not be so large that the FPG contains no edges. If the average degree is around 1 and there are around 2,000 non-isolated vertices then the average degree of non-isolated vertices is ~5,000 x 1 / 2,000 = ~2.5 (more accurately 2.79 for RLA with T = 1/2 and 2.42 for RLCSS with T = 1/4), leading to average local graphs with 5 – 10 vertices.

In summary, this suggests that a reasonable range of values of T for the user to control is [0.333, 0.5] for RLA and [0.125, 0.25] for RLCSS.

4.1.2 Multilevel filtering

For small or medium sized datasets, such as the test dataset, computational complexity is not a major issue. However, for the entire corpus it is not practical to run the graph construction process in full, hence the development of the multilevel filtering scheme which aims to filter out dissimilar tunes at coarse representations (when the interval arrays are much shorter and the local alignment much faster). The downside is that the multilevel scheme may mistakenly filter out similar tunes.

Tables 2 and 3 explore this with filtering results for the RLA and RLCSS algorithms and for various combinations of T and T^l. For the single level (SL) variants no filtering takes place but, as discussed in section 3.6, for the multilevel filtering variants (MLF), the larger the value of T^l the more edges will be filtered at coarse levels. Most of these edges would not be included in the fundamental proximity graph (FPG) as the underlying tunes are too dissimilar and so the multilevel filtering speeds up the matching. However, as T^l increases towards T the tendency is for it to filter out more FPG edges in error. The aim therefore is to find a suitable value of T^l which minimises both the runtime and the percentage of FPG edges filtered (although the filtered FPG edges are likely to arise from the weakest matches and might subsequently be removed anyway during sparsification).

	T	T^l	#edges in FPG	#edges in FPG filtered	%age filtered	runtime (s)
SL	1/3	n/a	51,854	n/a		1,188
MLF		1/16		1,734	3.34%	1,415
MLF		1/12		13,451	25.94%	714
MLF		1/8		35,293	68.06%	235
MLF		1/6		47,790	92.16%	84
SL	1/2	n/a	2,970	n/a		1,001
MLF		1/8		294	9.90%	229
MLF		1/6		597	20.10%	75
MLF		1/4		687	23.13%	52
MLF		1/2		1,347	45.35%	50

Table 2. Filtering results for the RLA algorithm.

	T	T^l	#edges in FPG	#edges in FPG filtered	%age filtered	runtime (s)
SL	1/8	n/a	48,405	n/a		900
MLF		1/16		913	1.89%	740
MLF		1/12		7,119	14.71%	316
MLF		1/8		26,593	54.94%	94
SL	1/6	n/a	5,039	n/a		880
MLF		1/12		153	3.04%	328
MLF		1/8		269	5.34%	96
MLF		1/6		1,304	25.88%	35
SL	1/4	n/a	2,410	n/a		842
MLF		1/8		4	0.17%	90
MLF		1/6		8	0.33%	33
MLF		1/4		90	3.73%	25

Table 3. Filtering results for the RLCSS algorithm.

Taking the data as a whole first of all, it can be seen that when the FPG is sparse the filtering is more successful. For example, for RLA with T = T^l = 1/2, the maximum filtration is 45.35% as compared with 92.16% when T = T^l = 1/3. Similarly for RLCSS with T = T^l = 1/4 the maximum filtration is just 3.73% as compared with 54.94% when T = T^l = 1/8.

Comparing RLA with RLCSS, however, it is clear that RLCSS is much more successful at not filtering out FPG edges although it may still filter a lot (say more than 10%) if the FPG is not particularly sparse and T^l is close to T (for example when T = T^l = 1/8 or T = T^l = 1/4).

It is possible to reduce filtering for RLA down to less than 10% but only for the smallest values of T^l, specifically T^l = 1/16 for T = 1/3 and T^l = 1/8 for T = 1/2. This is not so useful as the multilevel filtering doesn't improve the runtime so much: for example MLF actually increases the runtime from 1,188 seconds to 1,415 for T^l = 1/16 and T = 1/3. The runtime results are better for T^l = 1/8 for T = 1/2 and MLF is over 4 times faster than SL (229 seconds as compared with 1,001) with 9.90% filtering – however, this is at the upper end of the range suggested above for T.

Conversely for RLCSS there are combinations of T and T^1 which achieve significantly less than 10% filtering and where T^l is large enough to dramatically improve runtime. The best example is T = 1/6 and T^l = 1/8 where the MLF runtime is 96 seconds as compared with 880 for SL at the expense of only 5.34% filtering. Fortunately, this is in the middle of the range of values of T that might be appropriate for a user to control (i.e. [0.125, 0.25] – see above). Even at the bottom end of the range, T = 1/8 = 0.125, it is possible to use T^l = 1/12 and achieve substantial time savings (316 seconds for MLF as compared with 900 for SL) with only 14.71% filtering. At the top end of the range, where the FPG is very sparse it is possible to use T = T^l = 1/4 and see a huge time saving (25 seconds for MLF as compared with 842 for SL) at the expense of only 3.73% filtering.

It is not totally clear why multilevel filtering does not combine so well with RLA as it does with RLCSS but the likelihood is that the sub-sequences found by RLA at the coarse levels do not necessarily match those found at finer levels. Conversely, provided the coarsening algorithm removes the same entries in both strings, then a longest common substring at a finer level will result in corresponding longest common substrings at coarser levels (for example, if `****abcdefgh****` is coarsened to `**aceg**` and subsequently to `*ae*`).

Note also that this is not an unknown occurrence when using the multilevel paradigm in other fields, [4]. Sometimes the more sophisticated local refinement algorithms interact less well with multilevel coarsening and in fact the best combination is often a smart coarsening algorithm with a relatively simple local refinement scheme.

4.1.3 Sample local graph results

Table 4 shows the characteristics of the local graphs produced for three T / T^l configurations using the three different frameworks (SL, MLF & MLW) and RLCSS as the similarity measure. The characteristics are given in terms of the number of local graphs produced (essentially the number of non-isolated vertices for that value of T, potentially reduced by filtering and sparsification) plus average and maximum values for the number of vertices and edges in each local graph.

There are not many conclusions that can be drawn from this table but it does indicate that for each value of T the characteristics are similar for all three frameworks (provided a suitable value of T^l is chosen).

				#vertices		#edges	
	T	T^l	#graphs	avg.	max.	avg.	max.
SL	1/8	n/a	4,436	13.5	32	13.9	36
MLF		1/12	4,381	13.2	29	13.5	32
MLW		1/12	4,381	12.6	26	12.9	32
SL	1/6	n/a	2,812	6.0	20	6.4	26
MLF		1/8	2,745	5.8	22	6.1	28
MLW		1/8	2,745	5.8	22	6.2	28
SL	1/4	n/a	1,800	4.0	15	4.1	26
MLF		1/4	1,742	4.0	15	4.1	26
MLW		1/4	1,742	4.0	13	4.0	24

Table 4. Local graph results for the RLCSS algorithm.

4.2 Results – entire abc corpus

The second data set is the entire abc corpus which currently consists of around 509,000 tunes from across the web. Of these 273,000 are exact electronic duplicates which are excluded and another 41,500 are potentially copyright and also ignored. A further 7,500 (3.8% of the remainder) are excluded because of implementation limitations (see [1]), leaving a total of 186,847.

Taking into account the various observations above, it seems that a good configuration is RLCSS as the local matching scheme with T = 1/6 and T^l = 1/8.

Table 5 shows local graph characteristics for MLF and MLW both of which took around 24 hours to run. In contrast the runtime prediction for SL was 2 years! (Indeed if sparser local graphs are acceptable, the multilevel frameworks take only around 8 hours for T = T^l = 1/4.)

		#vertices		#edges	
	#graphs	avg.	max.	avg.	max.
MLF	160,157	9.6	44	12.0	120
MLW	160,157	9.3	40	11.6	116

Table 5. Local graph results for the entire corpus.

Again, not many conclusions can be draw from this table other than the similar characteristics of MLF and MLW. However, the resulting local graphs for MLF can be explored at abcnotation.com.

5. CONCLUSIONS

This has paper presented an investigation into constructing proximity graphs using a multilevel melodic similarity metric. It also discussed the use of two recursive variants of local alignment algorithms (RLA & RLCSS) and a similarity measure adapted to handle their global nature.

The results suggest that multilevel filtering, coupled with RLCSS, works well at building proximity graphs from a corpus of tunes significantly speeding up the runtime without filtering out too many matches.

Although further work remains to eliminate some of the minor limitations in the multilevel matching, the results can be explored at abcnotation.com.

6. REFERENCES

[1] C. Walshaw, "TuneGraph: an online visual tool for exploring melodic similarity," in *Proc. Digital Research in the Humanities and Arts*, 2015, pp. 55–64.

[2] R. Typke, *Music Retrieval based on Melodic Similarity*, 2007.

[3] A. Marsden, "Schenkerian Analysis by Computer: A Proof of Concept," *J. New Music Res.*, vol. 39, no. 3, pp. 269–289, 2010.

[4] C. Walshaw, "Multilevel Refinement for Combinatorial Optimisation: Boosting Metaheuristic Performance," in *Hybrid Metaheuristics - An emergent approach for optimization*, C. Blum, Ed. Springer, Berlin, 2008, pp. 261–289.

[5] B. Janssen, P. van Kranenburg, and A. Volk, "A Comparison of Symbolic Similarity Measures for Finding Occurrences of Melodic Segments," in *Proc. ISMIR*, 2015, pp. 659–665.

[6] C. Walshaw, "Multilevel Melodic Matching," in *5th Intl. Workshop on Folk Music Analysis*, 2015, pp. 130–137.

[7] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Mol. Biol.*, vol. 147, pp. 195–197, 1981.