4.4 Edit (Levenshtein) distance

Edit distance, also known as Levenshtein distance or evolutionary distance is a concept from information retrieval and it describes the number of edits (insertions, deletions and substitutions) that have to be made in order to change one string to another. It is the most common measure to expose the dissimilarity between two strings (Levenshtein 1966; Navarro & Raffinot 2002).

The edit distance *ed(x, y)* between strings $x=x_1 \ldots x_m$ and $y=y_1 \ldots y_n$, where $x, y \in \Sigma*$ is the minimum cost of a sequence of editing steps required to convert *x* into *y*. The alphabet Σ of possible characters *ch* gives Σ∗, the set of all possible sequences of $ch \in \Sigma$. Edit distance can be calculated using dynamic programming (Navarro & Raffinot 2002). Dynamic programming is a method of solving a large problem by regarding the problem as the sum of the solution to its recursively solved subproblems. Dynamic programming is different to recursion however. In order to avoid recalculating the solutions to subproblems, dynamic programming makes use of a technique called *memoisation*, whereby the solutions to subproblems are stored once calculated, to save recalculation.

To compute the edit distance *ed(x,y)* between strings *x* and *y*, a matrix $M_{1\ldots m+1,1\ldots n+1}$ is constructed where $M_{i,j}$ is the minimum number of edit operations needed to match $x_{1\ldots i}$ to $y_{1\ldots j}$. Each matrix element $M_{i,j}$ is calculated as per Equation 9, where $\delta(a,b) = 0$ if $a = b$ and 1 otherwise. The matrix element $M_{1,1}$ is the edit distance between two empty strings.

$$M_{1,1} \leftarrow 0$$

$$M_{i,j} \leftarrow \min \begin{cases} M_{i-1,j} + 1 \\ M_{i,j-1} + 1 \\ M_{i-1,j-1} + \delta(x_i, y_j) \end{cases}$$

Equation 9

The algorithm considers the last characters, x_i and y_j. If they are equal, then $x_{1..i}$ can be converted into $y_{1..j}$ at a cost of $M_{i-1,j-1}$. If they are not equal, x_i can be converted to y_j by substitution at a cost of $M_{i-1,j-1} + 1$, or x_i can be deleted at a cost of $M_{i-1,j} + 1$ or y_j can be appended to *x* at a cost of $M_{i,j-1} + 1$. The minimum edit distance between *x* and *y* is given by the matrix entry at position $M_{m+1,n+1}$.

Table 11 is an example of the matrix produced to calculate the edit distance between the strings "DFGDGBDEGGAB" and "DGGGDGBDEFGAB". The edit distance between these strings given as $M_{m+1,n+1}$ is 3.

		D	**G**	**G**	**G**	**D**	**G**	**B**	**D**	**E**	**F**	**G**	**A**	**B**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
D	1	0	1	2	3	4	5	6	7	8	9	10	11	12
F	2	1	1	2	3	4	5	6	7	8	8	9	10	11
G	3	2	1	1	2	3	4	5	6	7	8	8	9	10
D	4	3	2	2	2	2	3	4	5	6	7	8	9	10
G	5	4	3	2	2	3	2	3	4	5	6	7	8	9
B	6	5	4	3	3	3	3	2	3	4	5	6	7	8
D	7	6	5	4	4	3	4	3	2	3	4	5	6	7
E	8	7	6	5	5	4	4	4	3	2	3	4	5	6
G	9	8	7	6	5	5	4	5	4	3	3	3	4	5
G	10	9	8	7	6	6	5	5	5	4	4	3	4	5
A	11	10	9	8	7	7	6	6	6	5	5	4	3	4
B	12	11	10	9	8	8	7	6	7	6	6	5	4	3

Table 11: Edit distance matrix for the strings "DFGDGBDEGGAB" and "DGGGDGBDEFGAB" with the minimum edit distance position highlighted

An alternative expression of the edit distance equation which gives identical results is given in Equation 10, which is equivalent to Equation 9 because neighbouring cells in M differ by at most 1.

$$M_{i,1} \leftarrow i-1, M_{1,j} \leftarrow j-1$$

$$M_{i,j} \leftarrow \begin{cases} M_{i-1,j-1} & \text{if } x_i = y_i \\ 1 + \min\left(M_{i-1,j-1}, M_{i-1,j}, M_{i,j-1}\right) & \text{else} \end{cases}$$

Equation 10

The algorithm can be adapted to find the lowest edit distances for x in substrings of y. This is achieved by setting $M_{1,j} = 0$ for all $j \in 1...n+1$. In contrast to the edit distance algorithm described above, the last row $M_{m+i,j}$ is then used to give a *sliding window* edit distance for x in substrings of y as per Equation 11 (Navarro & Raffinot 2002).

$$eds(x, y) = \min_{1 \geq j \geq n+1} \left(M_{m+1,j} \right)$$

Equation 11

An example of this variation on the edit distance applied to search for the pattern "BDEE" in "DGGGDGBDEFGAB" is given in Table 12. The minimum edit distance positions are highlighted. Variations on the edit distance algorithm have been applied in domains such as DNA analysis and automated spell checking and are commonly used in MIR systems (Birmingham et al. 2001; Lemstrom & Perttu 2000; Rho & Hwang 2004; McPherson & Bainbridge 2001; Prechelt & Typke 2001).

		D	**G**	**G**	**G**	**D**	**G**	**B**	**D**	**E**	**F**	**G**	**A**	**B**
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	1	1	1	1	1	1	1	0	1	1	1	1	1	0
D	2	1	2	2	2	1	2	1	0	1	2	2	2	1
E	3	2	2	3	3	2	2	2	1	0	1	2	3	2
E	4	3	3	3	4	3	3	3	2	1	1	2	3	3

Table 12: Edit distance for the string "BDEE" in "DGGGDGBDEFGAB". This string represents the first 13 notes from the tune "Jim Coleman's" in normalised ABC format

Instead of using dynamic programming, bit-parallelism can be employed to compare strings with at most n errors. Bit-parallel algorithms simulate classical string matching algorithms, but use bit-masks to store the number of errors allowed. In this way, the algorithms are limited by the word size of the masks used (Navarro & Raffinot 2002). Bit-parallel algorithms have the advantage of having faster execution times, but the disadvantage of limiting of the maximum number of allowable errors to the word size of the bit-masks used (Lemstrom & Perttu 2000; Navarro & Raffinot 2002).

It is understood from experiments with human listeners that humans perceive transposed melodies to be similar. Interestingly studies in animals have demonstrated that this ability is unique to humans (Kenneally 2008). Hence there have been several attempts to adapt the edit distance algorithm for melodic dissimilarity to provide *transposition invariant* melodic dissimilarity. Mongeau & Sankoff (1990) for example use intervals between successive pitches to represent a melody for a dissimilarity comparison instead of the absolute values of pitches. Their algorithms can be understood by first considering the note alphabet Σ to be = $\mathbb{Z}$ or $\mathbb{R}$, the integer or real alphabet. The string x' represents the transposed copy of x, transposed by t, if $x' = (x_1 + t)\ (x_2 + t)...(x_m + t)$. For example if a melody was represented by the string x = {3, 7, 5, 5, 8, 7, 7,5 ,3} it could be relatively encoded as x' = {4, -2, 0, 3, -1, 0, -2, -2}. Using this scheme, there is naturally one less element in interval representation of the melody then in the original melody. The crucial property of this representation is that it is transposition invariant. In other words, if x and y are transpositions of each other, then $x' = y'$. The limitation of this approach becomes apparent if the case of an insertion or a deletion is considered. Consider the two strings x = {1, 2, 3, 4, 5} and y = {1, 3, 4, 5}. The edit distance between these strings $ed(x, y) = 1$. When converted to an interval representation these strings become x'={1, 1, 1, 1} and y' = {2,1, 1}. The edit distance between these strings $ed(x', y') = 2$. Hence each insertion and deletion has a double weighting on the calculation of the transposition invariant edit distance of two melodic strings. Lemstrom & Ukkonen (2000) state that using interval encodings; when intervals are calculated on the fly from absolute sequences, a deletion or insertion transposes the rest of the melody and so as an alternative, they propose instead adapting a cost function for local transformations (insert, delete, replace) which is transposition invariant. A "standard" edit distance cost function considers the insertion, deletion and replacement of each pair of elements in x and y. In Lemstrom & Ukkonen's (2000) proposed transposition invariant edit distance calculation, the cost function is adapted to consider in addition, the previous and current characters in x and y. Equation 12 provides a transposition invariant method of calculating edit distance which is equivalent to Equation 9 for calculating transposition invariant edit distances.

$$M_{1,1} \leftarrow 0$$

$$M_{i,j} \leftarrow \min \begin{cases} M_{i-1,j} + 1 \\ M_{i,j-1} + 1 \\ M_{i-1,j-1} + \left(If\ x_i - x_{i-1} = y_j - y_{j-1}\ then\ 0\ else\ 1\right) \end{cases}$$

Equation 12

Algorithms for calculating transposition invariant distances (Hamming distance, longest common subsequence, edit distance) between strings are also given in (Makinen et al. 2003). Their transposition invariant minimum edit distance *edt* between x' and *y*, where *x* and *y* are melodic strings and x' is *x* transposed by *t*, is given in Equation 13.

$$edt(x', y) = \min_{t \in T}(\quad ed(x + t, y))$$

Equation 13

$T = \{x_i - y_j \mid 1 \leq i \leq m,\ 1 \leq j \leq n\}$, in other words, the set of all possible values for *t* which would result in an alignment between *x* and *y*. In order to calculate *edt*(*x'*,*y*), they propose using a brute force approach by calculating *ed*(*x* + *t*,*y*) for all *T*. This is similar to the approach described in section 4.3. This obviously increases the computational complexity of the algorithm over a straightforward edit distance calculation between the two strings. In order to speed up the calculation, they propose using sparse dynamic programming. Sparse dynamic programming was introduced in (Eppstein et al. 1992a; Eppstein et al. 1992b). The main idea behind these techniques is that only elements in a string associated with a match are visited. In order to achieve this, the authors propose calculating an ordered set of matching elements in x' and *y* for every value of *t* such that $M_t = \{(i, j) \mid x_i + t = y_j\}$. Using sparse dynamic programming, the computational complexity of the transposition invariant edit distance algorithm is $O(mn \log n)$ compared to $O(mn)$ for standard edit distance. They also present a measure called "Longest Common Hidden Melody", which is a transposition invariant version of the longest common subsequence measure.