libmusical 0.9

P. van Kranenburg

October 5, 2011

Contents

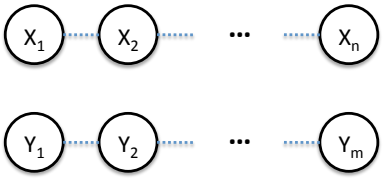
1 Introduction

libmusical is a c++-software library that provides classes and functions for alignment of sequences of symbols. The intended field of application is the alignment of musical sequences. However, the alignment algorithms are provided in their abstract forms. It is easy to apply the algorithms in other contexts as well.

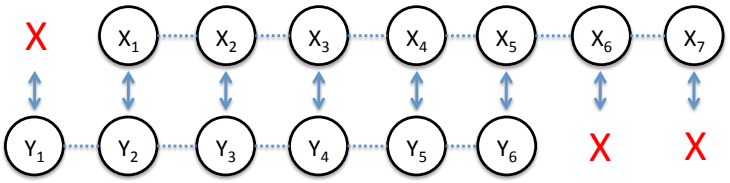
This library is a deliverable of the NWO WITCHCRAFTPLUS software-project.[1] One of the aims of this project is to provide robust implementations of software that was developed within the WITCHCRAFT research-project (Wiering et al., 2009).[2]

2 Alignment

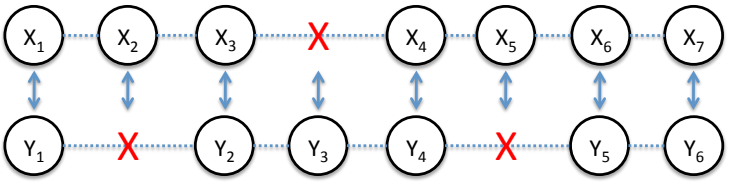
Cconsider two sequences of symbols $\mathbf{x} : x_1, \ldots, x_i, \ldots, x_n$, and $\mathbf{y} : y_1, \ldots, y_j, \ldots, y_m$.



A large number of alignments is possible. E.g., for $n = 7$ and $m = 6$,



or



etc.

In constructing an alignment of $\mathbf{x}$ and $\mathbf{y}$, symbol x_i can either be aligned with a symbol from sequence $\mathbf{y}$ or with a gap. Each alignment of two sequences

[1] http://www.catchplus.nl/diensten/deelprojecten/witchcraftplus/
[2] http://www.cs.uu.nl/research/projects/witchcraft/

gets a score, which is the sum of the scores of the alignments of the individual sybols. Alignment algorithms find the (or one of the) alignemnt with the highest score. Since the solution space is quite large, a dynamic programming approach is taken to find the optimal alignment efficiently. In the simplest form, the optimal alignment and its score are found by filling a matrix D recursively according to:

$$D(i,j) = \max \begin{cases} D(i-1,j-1) + S(x_i, y_j) \\ D(i-1,j) - \gamma \\ D(i,j-1) - \gamma \end{cases}, \tag{1}$$

in which $S(x_i, y_j)$ is a similarity measure for symbols, γ is the (fixed) gap penalty, $D(0,0) = 0$, $D(i,0) = -i\gamma$, and $D(0,j) = -j\gamma$. $D(i,j)$ contains the score of the optimal alignment up to x_i and y_j and therefore, $D(m,n)$ contains the score of the optimal alignment of the complete sequences. We can obtain the alignment itself by tracing back from $D(m,n)$ to $D(0,0)$; the algorithm has both time and space complexity $O(nm)$. This algorithm is known as the Needleman-Wunsch algorithm Needleman and Wunsch (1970).

3 Basic Architecture

3.1 Alignment

For an alignment, we need:

- two sequences of symbols;
- a similarity measure for symbols;
- a penalty function for insertion of gaps;
- the alignment algorithm itself.

For all four of these elements, abstract classes are present in the library. To implement a specialization for the alignment of a specific kind of symbols, specific classes should be derived from the provided abstract classes. The abstract classes are defined in header file `libmusical.h`. Everything is in namespace `musical`.

The most important abstract classes are:

Algorithm An alignment algorithm.

Symbol A symbol.

Sequence A sequence of symbols.

Sequences Two sequences of symbols that should be aligned. We use a specific class for this because the similarity measure for symbols might need to use the two entire sequences for computation of certain variables.

SimilarityRater Returns the similarity of two symbols.

GapRater Returns scores for insertions of one or more gaps.

Reader This class creates an object of class Sequence, containing a sequence of Symbol objects.

An alignment class has a Sequences-object, a SimilarityRater and a GapRater. The last two should be instances of derived classes and the Sequence-object could be an instance of a derived class. The algorithms handle sequences, symbols and raters at an abstract level. The SimilarityRater is the only class that need to know what kind of Symbols the sequence consists of.

3.2 Data representation and input

In the current implementation classes are provided to read a sequence from a JSON string with specific layout. As an example:

```
{
  "Name␣of␣Sequence":
    {
      "symbols":
        [
          {"attribute␣1":value,"attribute␣2":value,"attribute␣3":value},
          {"attribute␣1":value,"attribute␣2":value,"attribute␣3":value},
          {"attribute␣1":value,"attribute␣2":value,"attribute␣3":value},
          {"attribute␣1":value,"attribute␣2":value,"attribute␣3":value},
          {"attribute␣1":value,"attribute␣2":value,"attribute␣3":value}
        ]
    }
}
```

The name of the top-level object is the name of the sequence. The symbols consist of attributes.

The following (abstract) classes are involved in reading the data:

JSONSource The task for this class is to get a JSON string from somewhere.

JSONReader Derived from Reader. This creates a Sequence from a JSON source.

3.3 Results

Objects of the class `AlignmentVisualizer` are able to produce textual or graphical representations of the resulting alignemnt. Currently, one method is provided that outputs a textual description of the alignment to the standard output stream.

3.4 Currently Available Alignment Algorithms

Currently, the following variants of the alignment algorithm are implemented:

- Needleman-Wunsch (Needleman and Wunsch, 1970). Global alignment. One symbol from **x** is aligned with at most one symbol from **y**, and vice versa. Only gaps of length 1 are taken into account.
- Smith-Waterman (1981). Local alignment. This algorithm finds one or more local alignments, i.e., parts of the sequences that approximately match. One symbol from **x** is aligned with at most one symbol from **y**, and vice versa. Only gaps of length 1 are taken into account.
- Needleman-Wunsch-Gotoh (Gotoh, 1982). Global alignment with optimization for affine gap cost function. One symbol from **x** is aligned with at most one symbol from **y**, and vice versa.

3.5 Future Available Alignment Algorithms

Planned for future implementation:

- General Alignment. One or more symbols from **x** can be aligned with zero or more symbols from **y**, and vice versa. A continuous gap score function is employed.
- Smith-Waterman-Gotoh. Local alignment with affine gap cost.
- Mongeau-Sankoff variant (Mongeau and Sankoff, 1990). One symbol from **x** can be aligned with zero or more symbols from **y**, and vice versa. Only gaps of length 1 are taken into account.

3.6 Currently Available Gap Raters

The following gap raters are currently available:

- `ConstantLinearGapRater`. This returns a fixed score for a gap of lenght 1. To be used with `NeedlemanWunsch` and `SmithWaterman`.
- `ConstantAffineGapRater`. This returns a fixed score for a gap opening and a fixed score for a gap extension. To be used with `NeedlemanWunschGotoh`.

4 Specializations

4.1 NLB

A specialization is provided under the name "nlb" (Nederlandse Liederenbank, i.e. Database of Dutch Songs), defined in header file `OptiAlignment.h`. This implements the configuration that proved best in Van Kranenburg (2010, Ch.6). Each symbol consists of three attributes: pitch in base 40 representation, prhase position and metric weight. These attributes are used by the similarity measure (`OptiSimilarityRater`) to compute the similarity of two symbols.

4.1.1 Classes

The following classes have been added for this specialization:

OptiSymbol Symbol with three data members: pitch in base 40 representation, prhase position and metric weight.

OptiSequence Sequence of OptiSymbols. As an extra data-member, this class has a pitch histogram describing the pitch distribution in the melody.

OptiSequences Pair of OptiSequences. This class has a method to compute the shift of the histogram of the second sequence such that the intersection of both histograms is maximal. This is the shift in pitch that is needed to make the alignment transposition invariant.

OptiSimilarityRater Implements the similarity measure of OptiSymbols as described in Van Kranenburg (2010, Ch.6).

OptiJSONReader Converts a JSON representation of the sequence to an OptiSequence-object. The JSON string can provide a pitch histogram, as shown below. If it does not, then the histogram is generated on the fly. This histogram will be used to make the alignment transposition invariant.

Example of a JSON representation of a OptiSequence (only the first 5 symbols are shown):


```
{
  "NLB074575_01":
    {
      "symbols":
        [
          {"pitch40":129,"phrasepos":0,"ima":0.415755},
          {"pitch40":146,"phrasepos":0.1,"ima":0.568928},
          {"pitch40":152,"phrasepos":0.25,"ima":0.086433},
          {"pitch40":146,"phrasepos":0.3,"ima":0.45186},
          {"pitch40":141,"phrasepos":0.35,"ima":0.102845}
        ],
      "pitch40histogram":
        {
          "":{"pitch40":129,"value":0.148148},
          "":{"pitch40":135,"value":0.055556},
          "":{"pitch40":141,"value":0.074074},
          "":{"pitch40":146,"value":0.277778},
          "":{"pitch40":152,"value":0.222222},
          "":{"pitch40":158,"value":0.148148},
          "":{"pitch40":163,"value":0.037037},
          "":{"pitch40":169,"value":0.037037}
        }
    }
}
```


4.1.2 Example

```
#include <iostream>
using namespace std;

#include "libmusical.h"
#include "OptiAlignment.h"

int main(int argc, char * argv[]) {

  // Get a JSON string for sequence 1 from a file
  // Create a Reader object for the JSON string
  musical::OptiJSONReader mr1(new musical::JSONFileSource("/path/to/melody1.json"));

  // Ask the Reader to generate the Sequence
  musical::OptiSequence * seq1 =
    static_cast<musical::OptiSequence*>(mr1.generateSequence());

  // Do the same for sequence 2
  musical::OptiJSONReader mr2(new musical::JSONFileSource("/path/to/melody2.json"));
  musical::OptiSequence * seq2 =
    static_cast<musical::OptiSequence*>(mr2.generateSequence());

  // Encapsulate the two sequences in a Sequences object
  musical::OptiSequences seqs = musical::OptiSequences(seq1,seq2);

  // Create an alignment algorithm
  musical::NeedlemanWunschGotoh nw = musical::NeedlemanWunschGotoh(&seqs);

  // Assign a similarity rater
  nw.setSimilarityRater( new musical::OptiSimilarityRater() );

  // Assign a gap rater
  nw.setGapRater( new musical::ConstantLinearGapRater(-0.8) );

  // Do the alignment
  nw.doAlign();

  // Print the score
  cout << "Score:" << nw.getScore() << endl;

  // Print the alignment to stdout
  musical::AlignmentVisualizer av(&nw);
  av.basicStdoutReport();

  return 0;
}
```

In a Unix environment this can be compiled with the following command:

```
g++ tst.cpp -I/usr/local/include/libmusical -lmusical
```

assuming that libmusical was installed in `/usr/local`.

4.2 Midi

Another specialization is provided for midi files. Classes are defined in `MidiAlignment.h`.

4.2.1 Classes

The following classes have been added for this specialization:

MidiSymbol Symbol with three data members: pitch in base 12 representation, onset and duration.

MidiExactPitchIntervalSimilarityRater Returns score 1.0 if the interval of the note from sequence 1 with its previous note is exactly the same as the interval of the note from sequence 2 with its previous note. Returns -1.0 otherwise.

MidiFileReader Converts a midi file into a Sequence-object, containing MidiSymbols.

5 Applications

5.1 nlbdistmat

`nlbdistmat` creates a distance matrix for melodies in NLB JSON-encoding, as shown in section 4.1. It accepts two files as input. These files should list the filenames of the melodies, one filename on each line. The first file contains the rows, the second the columns. Example:

```
nlbdistmat fullcollection.flist queries.flist
```

5.2 alignmidi

`alignmidi` aligns two monophonic midi files and writes the alignment to the standard output stream.

```
alignmidi file1.mid file2.mid
```

6 Implement a specific alignment task

This section contains the steps for adding another specialization with name `XX`:

1. Derive `XXSymbol` from `Symbol`. `XXSymbol` is supposed to have data memebers that correspond to the attributes of the symbol.

2. *Optional:* Derive `XXSequence` from `Sequence`. Only do this if you want to add data memebers or functions to the sequence.

3. *Optional:* Derive `XXSequences` from `Sequences`. Only do this if you want to add data memebers or functions to the pair of sequences.

4. Derive a reader class `XXReader` from `Reader`. Override the member function getSequence(). It should convert an input encoding to a `XXSequence` object.

5. Derive `XXSimilarityRater` from `SimilarityRater`. Override member function `getScore(...)`. It should return the similarity of a subset of symbols from sequence **x** with a subset of symbols from sequence **y**. For the current alogirhtms only the scores for alignment of one symbol from **x** and one symbol from **y** is required (see section 3.4).

6. *Optional:* Derive `XXGapRater` from `GapRater`.

7. *Optional:* Derive `XXAlignmentVisualizer` from `AlignmentVisualizer`.

After performing all these steps, the Sequences, and the raters can be assigned to an alignment algorithm.

7 Roadmap

The current version, 0.9, implements three alignment algorithms and two applications for musical sequences. There are, however, remaining issues that have to be sovled. Furthermore, future versions will provide new features and new alignment algorithms. For version 1.0 the following improvements are planned:

- Improvements in exception handling. This will make the software more robust.
- Adding Smith-Waterman-Gotoh algorithm.

8 Misc

8.1 Compilation

The latest version is available at: `http://sourceforge.net/projects/libmusical/`

To install, issue the following commands in a unix shell:

```
tar xfvz libmusical-0.9-Source.tar.gz
cd libmusical-0.9-Source/build
cmake ..
make
sudo make install
```

This will install the headers, library and two applications (`nlbdistmat` and `alignmidi`) into `/usr/local`.

The example application alignmidi aligns two midi files:

```
alignmidi file1.mid file2.mid
```

8.2 Levenshtein distance

The Levenshtein edit distance (Levenshtein, 1966) can be obtained by using the Needleman-Wunsch algorithm with fixed gap score of -1 and a similarityrater that returns 0 for a perfect match and -1 otherwhise. The score of the algorithm is the negation of the edit distance because it counts the edits (substitutions and insertions/deletions). Since the `operator==` should be overloaded for the specific kind of symbol at hand, such a similarity rater should be implemented for every kind of symbol.

Code example, assuming that the sequences consist of symbols that are accepted by a `XXLevenshteinSimilarityRater`:

```
// Assume we already have:
// Sequences * seqs, which contains:
// Sequence * seq1
// Sequence * seq2

// Create the similarity rater
musical::XXLevenshteinSimilarityRater * lr =
  new musical::XXLevenshteinSimilarityRater();

// Create the gap rater with gap score -1.0
musical::ConstantLinearGapRater * gr =
  new ConstantLinearGapRater(-1.0);

// Create the aligner
musical::NeedlemanWunsch nw = musical::NeedlemanWunsch(seqs);
nw.setSimilarityRater(lr);
mw.setGapRater(gr);

// Do the alignment
nw.doAlign();

// Now the edit distance is available as the score of the alignment
cout << "Edit␣distance:" << nw.getScore() << endl;
```

References

Gotoh, O. (1982). An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710.

Mongeau, M. and Sankoff, D. (1990). Comparison of musical sequences. *Computers and the Humanities*, 24:161–175.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453.

Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197.

Van Kranenburg, P. (2010). *A Computational Approach to Content-Based Retrieval of Folk Song Melodies*. PhD thesis, Utrecht University, Utrecht.

Wiering, F., Veltkamp, R. C., Garbers, J., Volk, A., and Van Kranenburg, P. (2009). Modelling folksong melodies. *Interdisciplinary Science Reviews*, 34(2–3):154–171.