# repytah

*Release 0.1.2*

**Chenhui Jia,Lizette Carpenter, Thu Tran, Amanda Y. Liu, Sasha Ye**

**May 15, 2023**

# GETTING STARTED

repytah is a Python package that builds aligned hierarchies, a representation for sequential data streams.

CHAPTER

ONE

# CITING REPYTAH

Please cite `repytah` using the following:

C. Jia et al., repytah: A Python package that builds aligned hierarchies for sequential data streams. Python package version 0.1.1, 2023. [Online]. Available: [https://github.com/smith-tinkerlab/repytah{]}(https://github.com/smith-tinkerlab/repytah).

## 1.1 Why repytah

Sequential data streams often have repeated elements that build on each other, creating hierarchies. Therefore, the goal of the Python package `repytah` is to extract these repetitions and their relationships to each other in order to form aligned hierarchies, a low-dimensional representation for sequential data that synthesizes and aligns all possible hierarchies of the meaningful repetitive structure in a sequential data stream along a common time axis.

An example of sequential data streams is music-based data streams, such as songs. In addition to visualizing the structure of a song, aligned hierarchies can be embedded into a classification space with a natural notion of distance and post-processed to narrow the exploration of a music-based data stream to certain lengths of structure, or to address numerous MIR tasks, including the cover song task, the segmentation task, and the chorus detection task.

For future work, based on the aligned hierarchies, we can build aligned sub-hierarchies. Aligned sub-hierarchies are a collection of individual aligned hierarchies for each repeated pattern in a given sequential data. With aligned sub-hierarchies, we can better deal with tasks that require a degree of flexibility. For example, we can implement the aligned sub-hierarchies to find all versions of the same piece of music based on a given version of the recording, which might involve different expressions including the number of repeats.

## 1.2 Installing from *conda* package-manager or from *pip*

### 1.2.1 Anaconda

The safest way to install repytah is through `conda`. In terminal, navigate to the directory that contains environment.yml and execute:

```
conda env create -f environment.yml
conda activate repytahenv
conda install -c conda-forge repytah
```

### 1.2.2 PyPI

In terminal, navigate to the directory that contains *requirements.txt* and execute the following command:

```
pip install -r requirements.txt
pip install repytah
```

Warning: Might encounter problems if Python version is not >= 3.7, <3.11.

## 1.3 Install from *pip* or *conda*

### 1.3.1 PyPI

The latest stable release is available on PyPI, and you can install it by running:

```
pip install repytah
```

### 1.3.2 Anaconda

If you use Anaconda, you can install the package using `conda-forge`:

```
conda install -c conda-forge repytah
```

### 1.3.3 Source

To build repytah from source, you need to first clone the repo using `git clone git@github.com:smith-tinkerlab/repytah.git`. Then build with `python setup.py build`. Then, to install repytah, say `python setup.py install`.

Alternatively, you can download or clone the repository and use `pip` to handle dependencies:

```
unzip repytah.zip
pip install -e repytah
```

or:

```
git clone https://github.com/smith-tinkerlab/repytah.git
pip install -e repytah-main
```

By calling `pip list` you should see `repytah` now as an installed package:

```
repytah (0.x.x, /path/to/repytah)
```

## 1.4 Utilities

utilities.py

This module, when imported, allows search.py, transform.py and assemble.py in the repytah package to run smoothly.

The module contains the following functions:

- **create_sdm**
  Creates a self-dissimilarity matrix; this matrix is found by creating audio shingles from feature vectors, and finding cosine distance between shingles.
- **find_initial_repeats**
  Finds all diagonals present in thresh_mat, removing each diagonal as it is found.
- **stretch_diags**
  Fills out diagonals in binary self-dissimilarity matrix from diagonal starts and lengths.
- **add_annotations**
  Adds annotations to each pair of repeated structures according to their length and order of occurrence.
- **__find_song_pattern**
  Stitches information about repeat locations from thresh_diags matrix into a single row.
- **reconstruct_full_block**
  Creates a record of when pairs of repeated structures occur, from the first beat in the song to the last beat of the song. Pairs of repeated structures are marked with 1's.
- **get_annotation_lst**
  Gets one annotation marker vector, given vector of lengths key_lst.
- **get_y_labels**
  Generates the labels for visualization.
- **reformat [Only used for creating test examples]**
  Transforms a binary matrix representation of when repeats occur in a song into a list of repeated structures detailing the length and occurrence of each repeat.

repytah.utilities.**create_sdm**(*fv_mat*, *num_fv_per_shingle*)

Creates self-dissimilarity matrix; this matrix is found by creating audio shingles from feature vectors, and finding the cosine distance between shingles.

**Parameters**

- **fv_mat** (*np.ndarray*) – Matrix of feature vectors where each column is a time step and each row includes feature information i.e. an array of 144 columns/beats and 12 rows corresponding to chroma values.
- **num_fv_per_shingle** (*int*) – Number of feature vectors per audio shingle.

**Returns**
Self-dissimilarity matrix with paired cosine distances between shingles.

**Return type**
self_dissim_mat (np.ndarray)

repytah.utilities.**find_initial_repeats**(*thresh_mat*, *bandwidth_vec*, *thresh_bw*)

Looks for the largest repeated structures in thresh_mat. Finds all repeated structures, represented as diagonals present in thresh_mat, and then stores them with their start/end indices and lengths in a list. As each diagonal is found, they are removed to avoid identifying repeated sub-structures.

**Parameters**

- **thresh_mat** (*np.ndarray[int]*) – Thresholded matrix that we extract diagonals from.
- **bandwidth_vec** (*np.ndarray[1D,int]*) – Array of lengths of diagonals to be found. Should be 1, 2, 3,..., n where n is the number of timesteps.
- **thresh_bw** (*int*) – One less than smallest allowed repeat length.

**Returns**
List of pairs of repeats that correspond to diagonals in thresh_mat.

**Return type**
all_lst (np.ndarray[int])

repytah.utilities.**stretch_diags**(*thresh_diags*, *band_width*)

Creates a binary matrix with full length diagonals from a binary matrix of diagonal starts and length of diagonals.

**Parameters**

- **thresh_diags** (*np.ndarray*) – Binary matrix where entries equal to 1 signals the existence of a diagonal.
- **band_width** (*int*) – Length of encoded diagonals.

**Returns**
Logical matrix with diagonals of length band_width starting at each entry prescribed in thresh_diag.

**Return type**
stretch_diag_mat (np.ndarray[bool])

repytah.utilities.**add_annotations**(*input_mat*, *song_length*)

Adds annotations to the pairs of repeats in input_mat.

**Parameters**

- **input_mat** (*np.ndarray*) – List of pairs of repeats. The first two columns refer to the first repeat of the pair. The third and fourth columns refer to the second repeat of the pair. The fifth column refers to the repeat lengths. The sixth column contains any previous annotations, which will be removed.
- **song_length** (*int*) – Number of audio shingles in the song.

**Returns**
List of pairs of repeats with annotations marked.

**Return type**
anno_list (np.ndarray)

repytah.utilities.**__find_song_pattern**(*thresh_diags*)

Stitches information from thresh_diags matrix into a single row, song_pattern, that shows the time steps containing repeats; From the full matrix that decodes repeat beginnings (thresh_diags), the locations, or beats, where these repeats start are found and encoded into the song_pattern array.

**Parameters**
**thresh_diags** (*np.ndarray*) – Binary matrix with 1 at the start of each repeat pair (SI,SJ) and 0 elsewhere. WARNING: Must be symmetric.

**Returns**
Row where each entry represents a time step and the group that time step is a member of.

**Return type**
song_pattern (np.ndarray)

repytah.utilities.**reconstruct_full_block**(*pattern_mat*, *pattern_key*)

Creates a record of when pairs of repeated structures occur, from the first beat in the song to the end. This record is a binary matrix with a block of 1's for each repeat encoded in pattern_mat whose length is encoded in pattern_key.

**Parameters**

- **pattern_mat** (*np.ndarray*) – Binary matrix with 1's where repeats begin and 0's otherwise.
- **pattern_key** (*np.ndarray*) – Vector containing the lengths of the repeats encoded in each row of pattern_mat.

**Returns**

Binary matrix representation for pattern_mat with blocks of 1's equal to the length's prescribed in pattern_key.

**Return type**

pattern_block (np.ndarray)

repytah.utilities.**get_annotation_lst**(*key_lst*)

Creates one annotation marker vector, given vector of lengths key_lst.

**Parameters**

**key_lst** (*np.ndarray[int]*) – Array of lengths in ascending order.

**Returns**

Array of one possible set of annotation markers for key_lst.

**Return type**

anno_lst_out (np.ndarray[int])

repytah.utilities.**get_y_labels**(*width_vec*, *anno_vec*)

Generates the labels for visualization with width_vec and anno_vec.

**Parameters**

- **width_vec** (*np.ndarray[int]*) – Vector of widths for a visualization.
- **anno_vec** (*np.ndarray[int]*) – Array of annotations for a visualization.

**Returns**

Labels for the y-axis of a visualization. Each label contains the width and annotation number of an essential structure component.

**Return type**

y_labels (np.ndarray[str])

repytah.utilities.**reformat**(*pattern_mat*, *pattern_key*)

Transforms a binary array with 1's where repeats start and 0's otherwise into a list of repeated structures. This list consists of information about the repeats including length, when they occur and when they end.

Every row has a pair of repeated structure. The first two columns are the time steps of when the first repeat of a repeated structure start and end. Similarly, the second two columns are the time steps of when the second repeat of a repeated structure start and end. The fifth column is the length of the repeated structure.

Reformat is not used in the main process for creating the aligned hierarchies. It is helpful when writing example inputs for the tests.

**Parameters**

- **pattern_mat** (*np.ndarray*) – Binary array with 1's where repeats start and 0's otherwise.

- **pattern_key** (*np.ndarray*) – Array with the lengths of each repeated structure in pattern_mat.

**Returns**
Array with the time steps of when the pairs of repeated structures start and end organized.

**Return type**
info_mat (np.ndarray)

## 1.5 Transform

transform.py

This module contains functions that transform matrix inputs into different forms that are of use in bigger functions where they are called. These functions focus mainly on overlapping repeated structures and annotation markers.

**The module contains the following functions:**

- **remove_overlaps**
  Removes any pairs of repeats with the same length and annotation marker where at least one pair of repeats overlap in time.
- **__create_anno_remove_overlaps**
  Turns rows of repeats into marked rows with annotation markers for the start indices and zeroes otherwise. After removing the annotations that have overlaps, the function creates separate arrays for annotations with overlaps and annotations without overlaps. Finally, the annotation markers are checked and fixed if necessary.
- **__separate_anno_markers**
  Expands vector of non-overlapping repeats into a matrix representation. The matrix representation is a visual record of where all of the repeats in a song start and end.

repytah.transform.**remove_overlaps**(*input_mat*, *song_length*)

Removes any pairs of repeat length and specific annotation marker where there exists at least one pair of repeats that overlap in time.

**Parameters**

- **input_mat** (*np.ndarray[int]*) – List of pairs of repeats with annotations marked. The first two columns refer to the first repeat or the pair, the second two refer to the second repeat of the pair, the fifth column refers to the length of the repeats, and the sixth column contains the annotation markers.
- **song_length** (*int*) – Number of audio shingles.

**Returns**

A tuple (lst_no_overlaps, matrix_no_overlaps, key_no_overlaps, annotations_no_overlaps, all_overlap_lst). All variables have data type np.ndarray[int].

lst_no_overlaps is a list of pairs of repeats with annotations marked where all the repeats of a given length and with a specific annotation marker do not overlap in time.

matrix_no_overlaps is a matrix representation of lst_no_overlaps with one row for each group of repeats.

key_no_overlaps is a vector containing the lengths of the repeats encoded in each row of matrix_no_overlaps.

annotations_no_overlaps is a vector containing the annotation markers of the repeats encoded in each row of matrix_no_overlaps.

all_overlap_lst is a list of pairs of repeats where for each pair of repeat length and specific annotation marker, there exists at least one pair of repeats that do overlap in time.

repytah.transform.__create_anno_remove_overlaps(*k_mat*, *song_length*, *band_width*)

Turns k_mat into marked rows with annotation markers for the start indices and zeroes otherwise. After removing the annotations that have overlaps, the function outputs k_lst_out which only contains rows that have no overlaps, then takes the annotations that have overlaps from k_lst_out and puts them in overlap_lst. Lastly, it checks if the proper sequence of annotation markers was given and fix them if necessary.

**Parameters**

- **k_mat** (*np.ndarray*) – List of pairs of repeats of length 1 with annotations marked. The first two columns refer to the first repeat of the pair, the second two refer to the second repeat of the pair, the fifth column refers to the length of the repeats, and the sixth column contains the annotation markers.
- **song_length** (*int*) – Number of audio shingles.
- **band_width** (*int*) – Length of repeats encoded in k_mat.

**Returns**

A tuple (pattern_row, k_lst_out, overlap_lst) where all variables have data type np.ndarray.

pattern_row marks where non-overlapping repeats occur, marking start indices with annotation markers and 0's otherwise.

k_lst_out is a list of pairs of repeats of length band_width that contain no overlapping repeats with annotations marked.

overlap_lst is a list of pairs of repeats of length band_width that contain overlapping repeats with annotations marked.

repytah.transform.__separate_anno_markers(*k_mat*, *song_length*, *band_width*, *pattern_row*)

Expands pattern_row, a row vector that marks where non-overlapping repeats occur, into a matrix representation or np.array. The dimension of this array is twice the pairs of repeats by song_length. k_mat provides a list of annotation markers that is used in separating the repeats of length band_width into individual rows. Each row will mark the start and end time steps of a repeat with 1's and 0's otherwise. The array is a visual record of where all of the repeats in a song start and end.

**Parameters**

- **k_mat** (*np.ndarray*) – List of pairs of repeats of length band_width with annotations marked. The first two columns refer to the start and end time steps of the first repeat of the pair, the second two refer to the start and end time steps of second repeat of the pair, the fifth column refers to the length of the repeats, and the sixth column contains the annotation markers. We will be indexing into the sixth column to obtain a list of annotation markers.
- **song_length** (*int*) – Number of audio shingles.
- **band_width** (*int*) – Length of repeats encoded in k_mat.
- **pattern_row** (*np.ndarray*) – Row vector of the length of the song that marks where non-overlapping repeats occur with the repeats' corresponding annotation markers and 0's otherwise.

**Returns**

A tuple (pattern_mat, pattern_key, anno_id_lst) where all variables have data type np.ndarray.

pattern_mat is a matrix representation where each row contains a group of repeats marked.

pattern_key is a column vector containing the lengths of the repeats encoded in each row of pattern_mat.

anno_id_lst is a column vector containing the annotation markers of the repeats encoded in each row of pattern_mat.

## 1.6 Assemble

assemble.py

This module finds and forms essential structure components, which are the smallest building blocks that form every repeat in the song.

These functions ensure that each time step of a song is contained in at most one of the song's essential structure components by checking that there are no overlapping repeats in time. When repeats overlap, they undergo a process where they are divided until there are only non-overlapping pieces left.

The module contains the following functions:

- **breakup_overlaps_by_intersect**
  Extracts repeats in input_pattern_obj that has the starting indices of the repeats, into the essential structure components using bw_vec, that has the lengths of each repeat.
- **check_overlaps**
  Compares every pair of groups, determining if there are any repeats in any pairs of the groups that overlap.
- **__compare_and_cut**
  Compares two rows of repeats labeled RED and BLUE, and determines if there are any overlaps in time between them. If there are overlaps, we cut the repeats in RED and BLUE into up to 3 pieces.
- **__num_of_parts**
  Determines the number of blocks of consecutive time steps in a list of time steps. A block of consecutive time steps represents a distilled section of a repeat.
- **__inds_to_rows**
  Expands a vector containing the starting indices of a piece or two of a repeat into a matrix representation recording when these pieces occur in the song with 1's. All remaining entries are marked with 0's.
- **__merge_based_on_length**
  Merges repeats that are the same length, as set by full_bandwidth, and are repeats of the same piece of structure.
- **__merge_rows**
  Merges rows that have at least one common repeat. These common repeat(s) must occur at the same time step and be of a common length.
- **hierarchical_structure**
  Distills the repeats encoded in matrix_no_overlaps (and key_no_overlaps) to the essential structure components and then builds the hierarchical representation. Optionally outputs visualizations of the hierarchical representations.

`repytah.assemble.breakup_overlaps_by_intersect(input_pattern_obj, bw_vec, thresh_bw)`

Extracts repeats in input_pattern_obj that has the starting indices of the repeats, into the essential structure components using bw_vec, that has the lengths of each repeat. The essential structure components are the smallest building blocks that form every repeat in the song.

**Parameters**

- **input_pattern_obj** (*np.ndarray*) – Binary matrix with 1's where repeats begin and 0's otherwise.

- **bw_vec** (*np.ndarray*) – Vector containing the lengths of the repeats encoded in input_pattern_obj.
- **thresh_bw** (*int*) – One less than the smallest allowable repeat length.

**Returns**

A tuple (pattern_no_overlaps, pattern_no_overlaps_key) where all variables have data type np.ndarray.

pattern_no_overlaps is a binary matrix with 1's where repeats of essential structure components begin.

pattern_no_overlaps_key is a vector containing the lengths of the repeats of essential structure components in pattern_no_overlaps.

repytah.assemble.**check_overlaps**(*input_mat*)

Compares every pair of repeat groups and determines if there are any repeats in any pairs of the groups that overlap.

**Parameters**

**input_mat** (*np.ndarray*) – Binary matrix with blocks of 1's equal to the length of repeats to be checked for overlaps.

**Returns**

Logical array where (i,j) = 1 if row i of input_mat and row j of input_mat overlap and (i,j) = 0 elsewhere.

**Return type**

overlap_mat (np.ndarray)

repytah.assemble.**__compare_and_cut**(*red*, *red_len*, *blue*, *blue_len*)

Compares two rows of repeats labeled RED and BLUE, and determines if there are any overlaps in time between them. If there is, then we cut the repeats in RED and BLUE into up to 3 pieces.

**Parameters**

- **red** (*np.ndarray*) – Binary row vector encoding a set of repeats with 1's where each repeat starts and 0's otherwise.
- **red_len** (*np.ndarray*) – Length of repeats encoded in red.
- **blue** (*np.ndarray*) – Binary row vector encoding a set of repeats with 1's where each repeat starts and 0's otherwise.
- **blue_len** (*np.ndarray*) – Length of repeats encoded in blue.

**Returns**

A tuple (union_mat, union_length) where all variables have data type np.ndarray.

union_mat is a binary matrix representation of up to three rows encoding non-overlapping repeats cut from red and blue.

union_length is a vector containing the lengths of the repeats encoded in union_mat.

repytah.assemble.**__num_of_parts**(*input_vec*, *input_start*, *input_all_starts*)

Determines the number of blocks of consecutive time steps in a list of time steps. A block of consecutive time steps represents a distilled section of a repeat. This distilled section will be replicated and the starting indices of the repeats within it will be returned.

**Parameters**

- **input_vec** (*np.ndarray*) – Vector that contains one or two parts of a repeat that are overlap(s) in time that may need to be replicated.
- **input_start** (*np.ndarray*) – Starting index for the part to be replicated.
- **input_all_starts** (*np.ndarray*) – Starting indices for replication.

**Returns**

A tuple (start_mat, length_vec) where all variables have data type np.ndarray.

start_mat is an array of one or two rows containing the starting indices of the replicated repeats.

length_vec is a column vector containing the lengths of the replicated parts.

repytah.assemble.**__inds_to_rows**(*start_mat*, *row_length*)

Expands a vector containing the starting indices of a piece or two of a repeat into a matrix representation recording when these pieces occur in the song with 1's. All remaining entries are marked with 0's.

**Parameters**

- **start_mat** (*np.ndarray*) – Matrix of one or two rows, containing the starting indices.
- **row_length** (*int*) – Length of the rows.

**Returns**

Binary matrix of one or two rows, with 1's where the starting indices and 0's otherwise.

**Return type**

new_mat (np.ndarray)

repytah.assemble.**__merge_based_on_length**(*full_mat*, *full_bw*, *target_bw*)

Merges repeats that are the same length, as set by full_bw, and are repeats of the same piece of structure.

**Parameters**

- **full_mat** (*np.ndarray*) – Binary matrix with ones where repeats start and zeroes otherwise.
- **full_bw** (*np.ndarray*) – Length of repeats encoded in input_mat.
- **target_bw** (*np.ndarray*) – Lengths of repeats that we seek to merge.

**Returns**

A tuple (out_mat, one_length_vec) where all variables have data type np.ndarray.

out_mat is a binary matrix with 1's where repeats start and 0's otherwise with rows of full_mat merged if appropriate.

one_length_vec is a vector that contains the length of repeats encoded in out_mat.

repytah.assemble.**__merge_rows**(*input_mat*, *input_width*)

Merges rows that have at least one common repeat; said common repeat(s) must occur at the same time step and be of common length.

**Parameters**

- **input_mat** (*np.ndarray*) – Binary matrix with ones where repeats start and zeroes otherwise.
- **input_width** (*int*) – Length of repeats encoded in input_mat.

**Returns**

Binary matrix with ones where repeats start and zeroes otherwise.

**Return type**
merge_mat (np.ndarray)

repytah.assemble.**hierarchical_structure**(*matrix_no_overlaps*, *key_no_overlaps*, *sn*, *vis=False*)

Distills the repeats encoded in matrix_no_overlaps (and key_no_overlaps) to the essential structure components and then builds the hierarchical representation. Optionally shows visualizations of the hierarchical structure via the vis argument.

**Parameters**

- **matrix_no_overlaps** (*np.ndarray*) – Binary matrix with 1's where repeats begin and 0's otherwise.
- **key_no_overlaps** (*np.ndarray*) – Vector containing the lengths of the repeats encoded in matrix_no_overlaps.
- **sn** (*int*) –
- **length** (*Song*) –
- **shingles.** (*which is the number of audio*) –
- **vis** (*bool*) – Shows visualizations if True (default = False).

**Returns**

A tuple (full_visualization, full_key, full_matrix_no_overlaps, full_anno_lst) where all variables have data type np.ndarray.

full_visualization is a binary matrix representation for full_matrix_no_overlaps with blocks of 1's equal to the lengths prescribed in full_key.

full_key is a vector containing the lengths of the hierarchical structure encoded in full_matrix_no_overlaps.

full_matrix_no_overlaps is a binary matrix with 1's where hierarchical structure begins and 0's otherwise.

full_anno_lst is a vector containing the annotation markers of the hierarchical structure encoded in each row of full_matrix_no_overlaps.

## 1.7 Search

search.py

This module holds functions used to find and record the diagonals in the thresholded matrix, T. These functions prepare the diagonals found for transformation and assembling later. The module contains the following functions:

- **find_complete_list**
  Finds all smaller diagonals (and the associated pairs of repeats) that are contained in pair_list, which is composed of larger diagonals found in find_initial_repeats.
- **__find_add_rows**
  Finds pairs of repeated structures, represented as diagonals of a certain length, k, that neither start nor end at the same time steps as previously found pairs of repeated structures of the same length.
- **find_all_repeats**
  Finds all the diagonals present in thresh_mat. This function is nearly identical to find_initial_repeats except for two crucial differences. First, we do not remove diagonals after we find them. Second, there is no smallest bandwidth size as we are looking for all diagonals.

- **find_complete_list_anno_only**
  Finds annotations for all pairs of repeats found in find_all_repeats. This list contains all the pairs of repeated structures with their starting/ending indices and lengths.

repytah.search.**find_complete_list**(*pair_list*, *song_length*)

Finds all smaller diagonals (and the associated pairs of repeats) that are contained in pair_list, which is composed of larger diagonals found in find_initial_repeats.

**Parameters**

- **pair_list** (*np.ndarray*) – List of pairs of repeats found in earlier steps (bandwidths MUST be in ascending order). If you have run find_initial_repeats before this script, then pair_list will be ordered correctly.
- **song_length** (*int*) – Song length, which is the number of audio shingles.

**Returns**
List of pairs of repeats with smaller repeats added.

**Return type**
final_lst (np.ndarray)

repytah.search.**__find_add_rows**(*lst_no_anno*, *check_inds*, *k*)

Finds pairs of repeated structures, represented as diagonals of a certain length, k, that that start at the same time step, or end at the same time step, or neither start nor end at the same time step as previously found pairs of repeated structures of the same length.

**Parameters**

- **lst_no_anno** (*np.ndarray*) – List of pairs of repeats.
- **check_inds** (*np.ndarray*) – List of starting indices for repeats of length k that we use to check lst_no_anno for more repeats of length k.
- **k** (*int*) – Length of repeats that we are looking for.

**Returns**
List of newly found pairs of repeats of length K that are contained in larger repeats in lst_no_anno.

**Return type**
add_rows (np.ndarray)

repytah.search.**find_all_repeats**(*thresh_mat*, *bw_vec*)

Finds all the diagonals present in thresh_mat. This function is nearly identical to find_initial_repeats, with two crucial differences. First, we do not remove diagonals after we find them. Second, there is no smallest bandwidth size as we are looking for all diagonals.

**Parameters**

- **thresh_mat** (*np.ndarray*) – Thresholded matrix that we extract diagonals from.
- **bw_vec** (*np.ndarray*) – Vector of lengths of diagonals to be found. Should be 1, 2, 3, . . . , n where n = number of timesteps.

**Returns**
Pairs of repeats that correspond to diagonals in thresh_mat.

**Return type**
all_lst (np.ndarray)

repytah.search.**find_complete_list_anno_only**(*pair_list*, *song_length*)

Finds annotations for all pairs of repeats found in find_all_repeats. This list contains all the pairs of repeated structures with their starting/ending indices and lengths.

**Parameters**

- **pair_list** (*np.ndarray*) – List of pairs of repeats. WARNING: Bandwidths must be in ascending order.
- **song_length** (*int*) – Number of audio shingles in song.

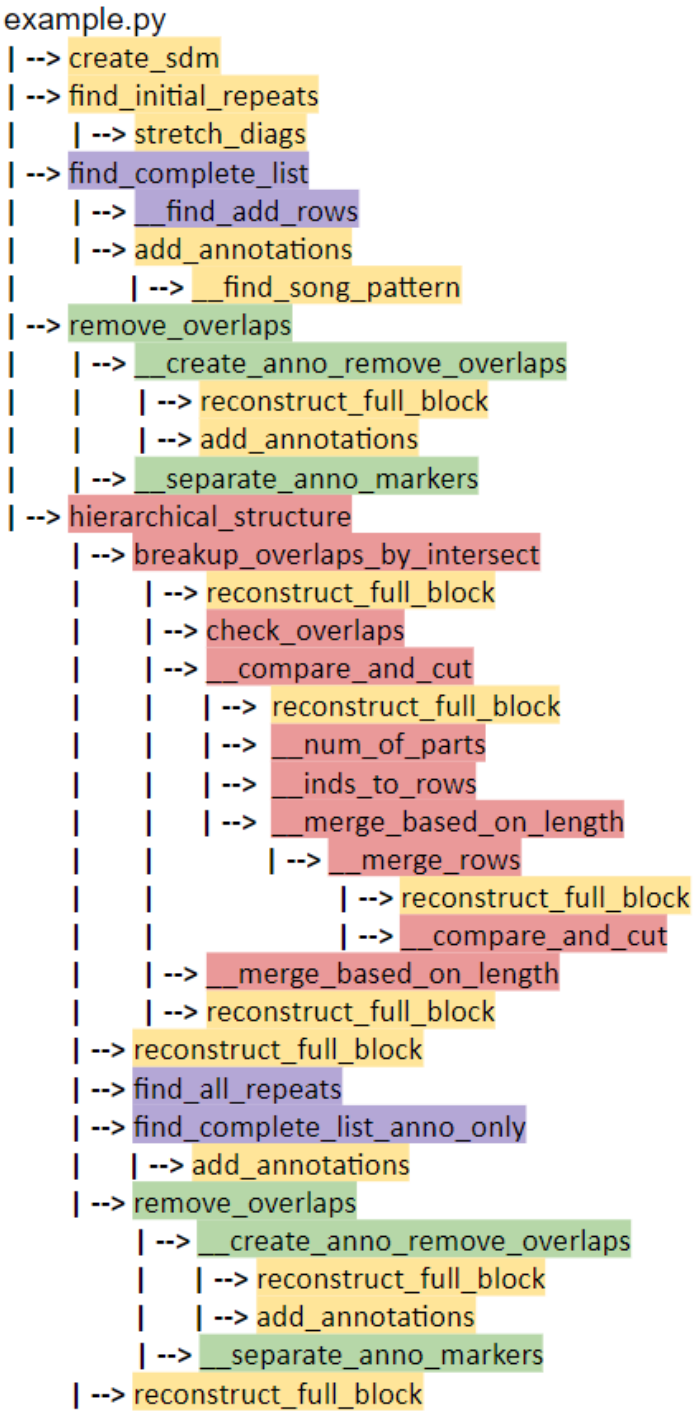
**Returns**
List of pairs of repeats with smaller repeats added and with annotation markers.

**Return type**
out_lst (np.ndarray)

## 1.8 Function Pipeline

Before we dive deep into the implementation of the package via the walk-through of a complete example, a function pipeline is shown below to illustrate the function calls in each step. For an in-depth look at each function, check the corresponding Jupyter Notebooks for each module.

- Yellow: Utilities
- Purple Search
- Green: Transform
- Red: Assemble


example.py
| --> create_sdm
| --> find_initial_repeats
| | --> stretch_diags
| --> find_complete_list
| | --> __find_add_rows
| | --> add_annotations
| | | --> __find_song_pattern
| --> remove_overlaps
| | --> __create_anno_remove_overlaps
| | | --> reconstruct_full_block
| | | --> add_annotations
| | --> __separate_anno_markers
| --> hierarchical_structure
| | --> breakup_overlaps_by_intersect
| | | --> reconstruct_full_block
| | | --> check_overlaps
| | | --> __compare_and_cut
| | | | --> reconstruct_full_block
| | | | --> __num_of_parts
| | | | --> __inds_to_rows
| | | | --> __merge_based_on_length
| | | | | --> __merge_rows
| | | | | | --> reconstruct_full_block
| | | | | | --> __compare_and_cut
| | | --> __merge_based_on_length
| | | --> reconstruct_full_block
| | --> reconstruct_full_block
| | --> find_all_repeats
| | --> find_complete_list_anno_only
| | | --> add_annotations
| | --> remove_overlaps
| | | --> __create_anno_remove_overlaps
| | | | --> reconstruct_full_block
| | | | --> add_annotations
| | | --> __separate_anno_markers
| | --> reconstruct_full_block

## 1.9 A Quick Start to Use the Package `repytah`

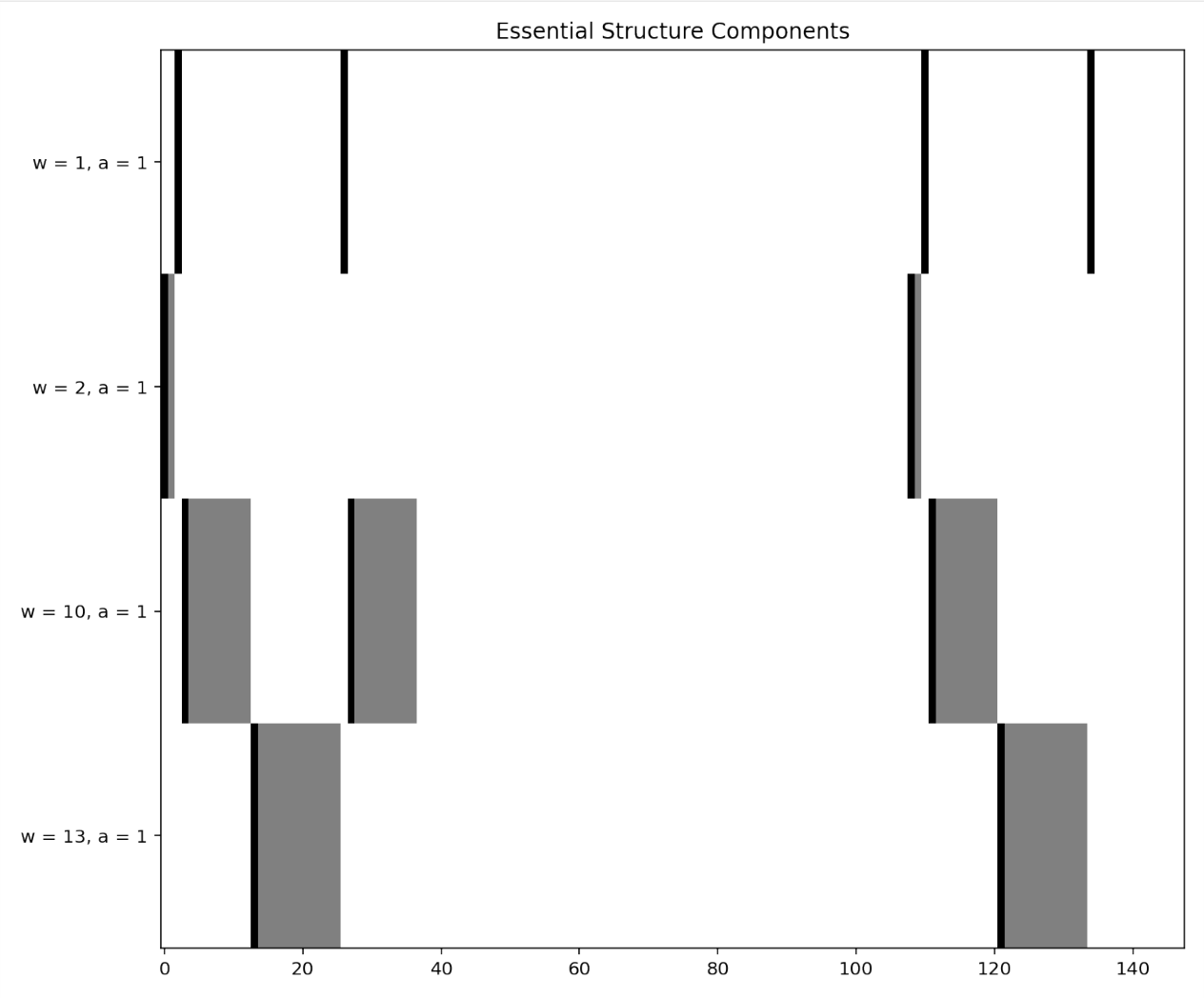
This notebook demonstrates how to use `repytah` to create aligned hierarchies for a music-based data stream. Before diving into the details, we'll walk through a brief example program.

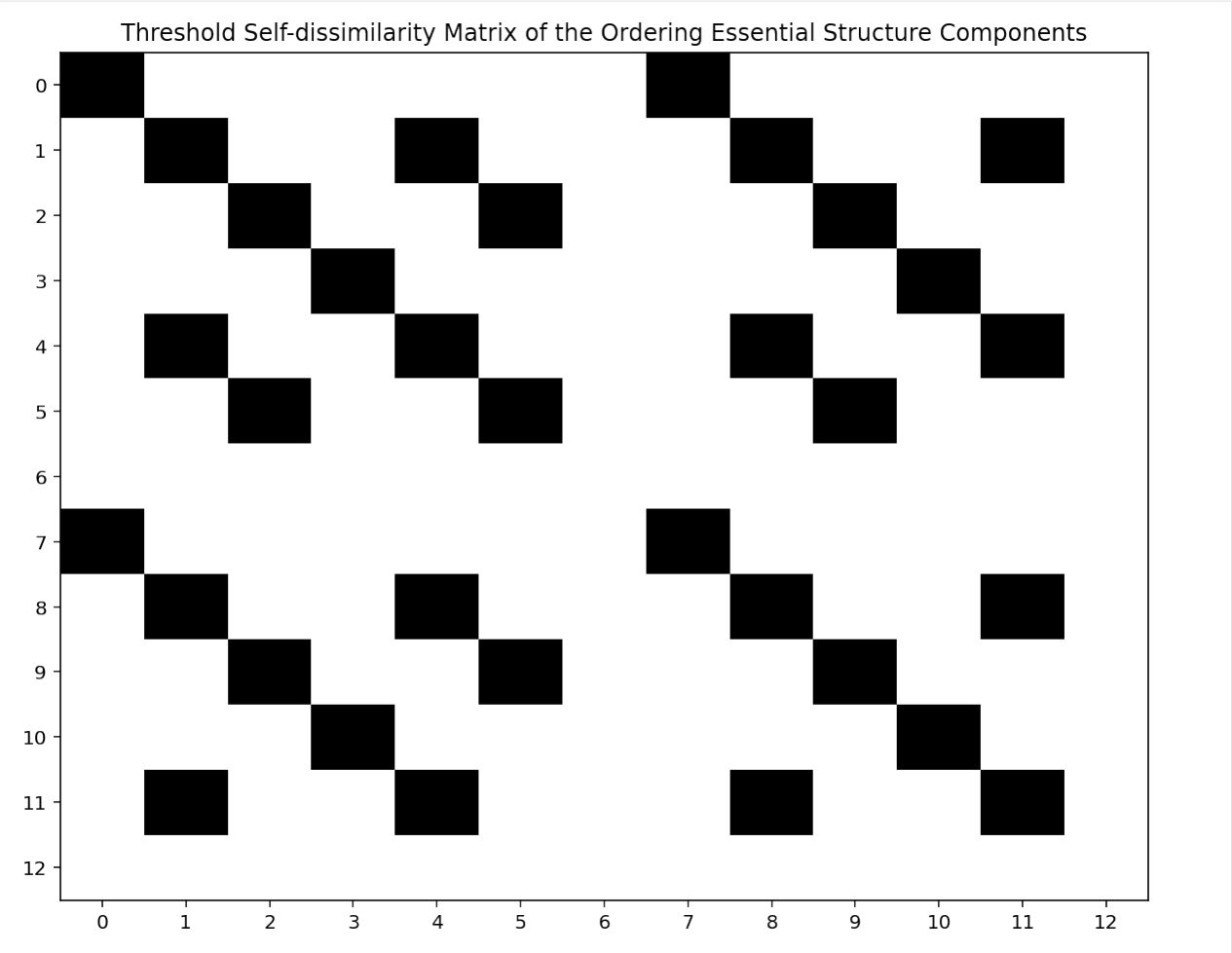
The example input is a csv file containing Chroma feature vectors for each beat of Chopin's Mazurka Op.30, No.1.

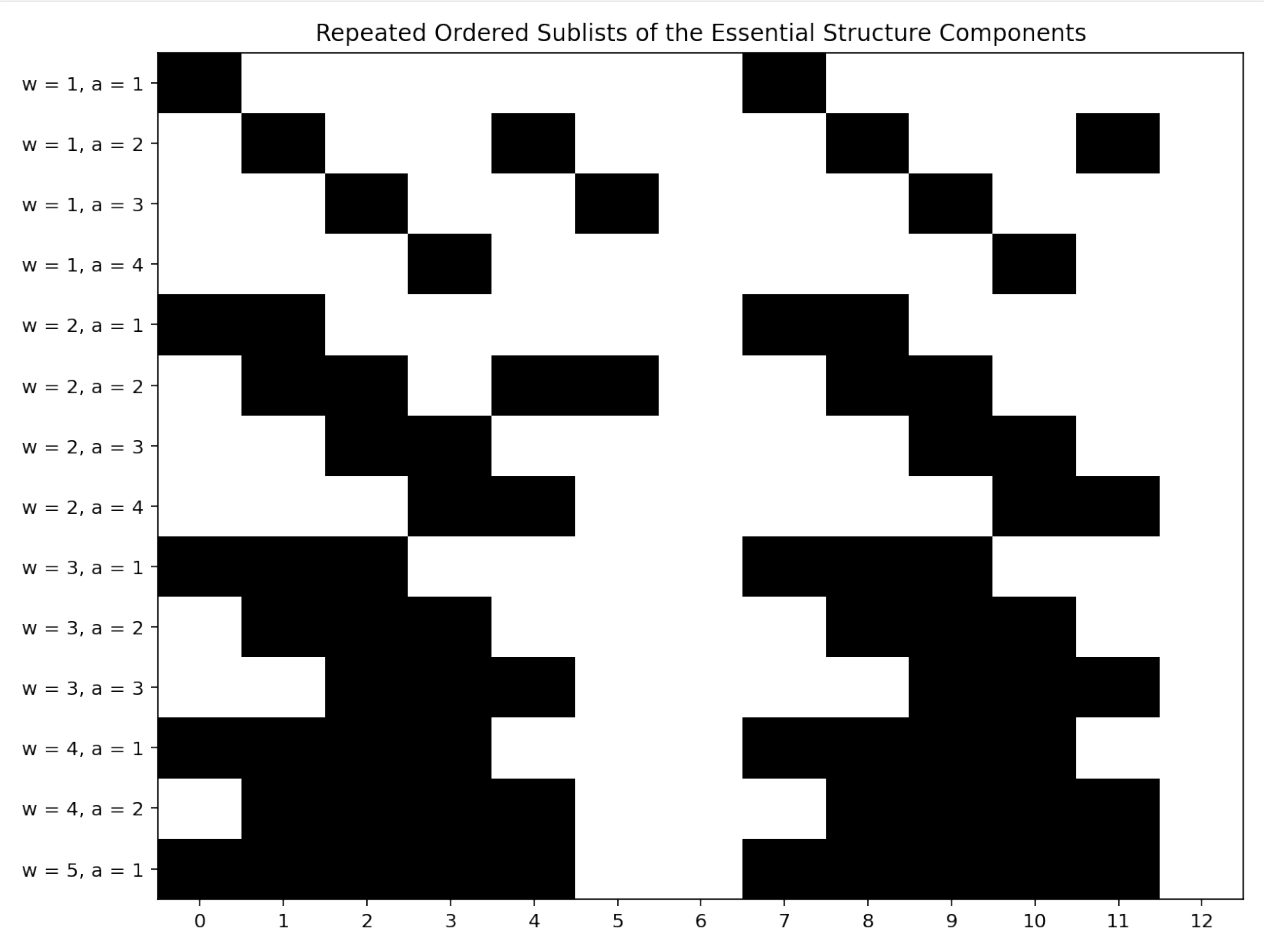
[1]:
```
# Standard imports
import scipy.io as sio
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Import the package repytah
from repytah import *

# Make the images clear
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

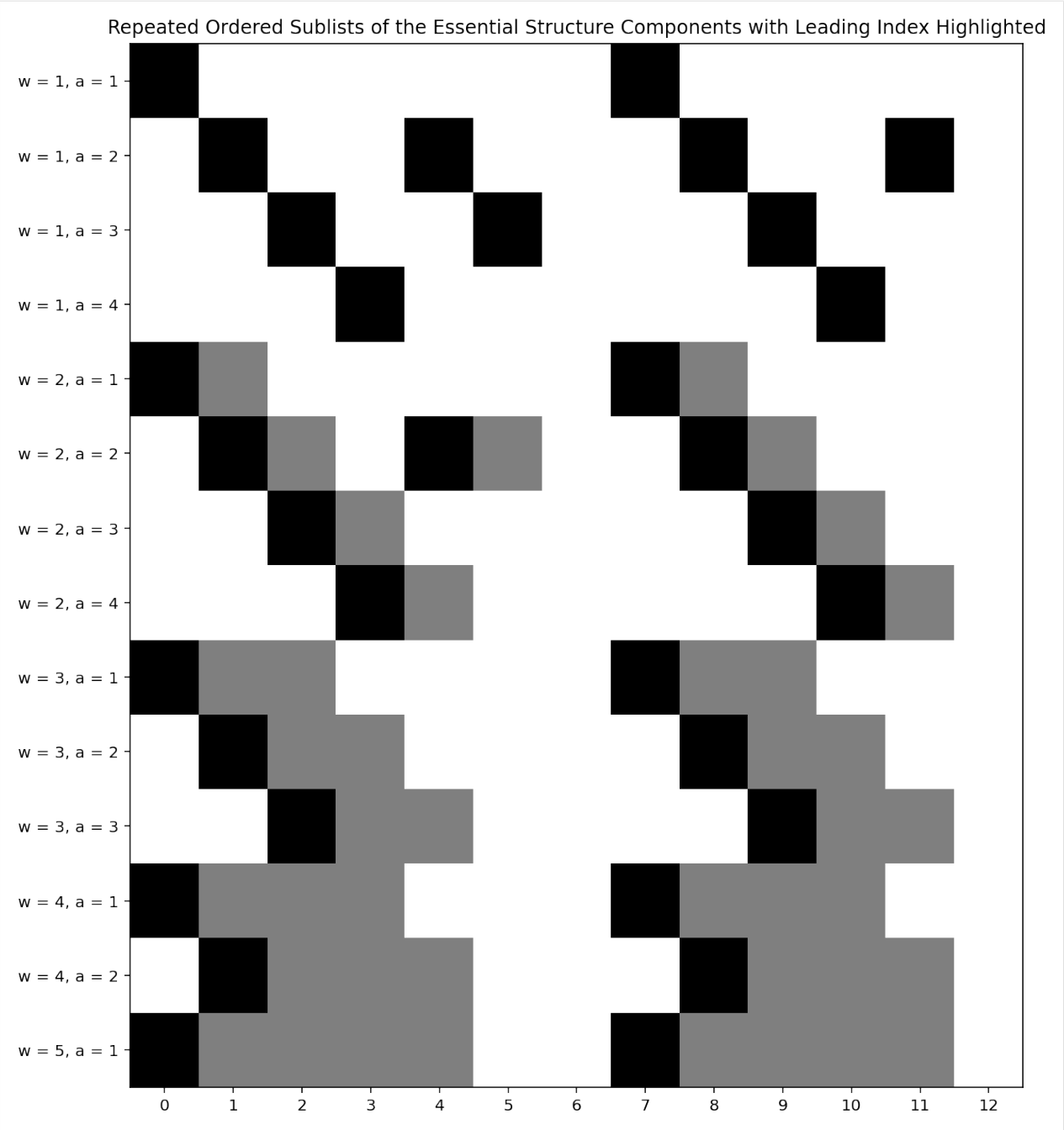
### 1.9.1 To skip visualizing the middle steps and get the output directly, try the following code:

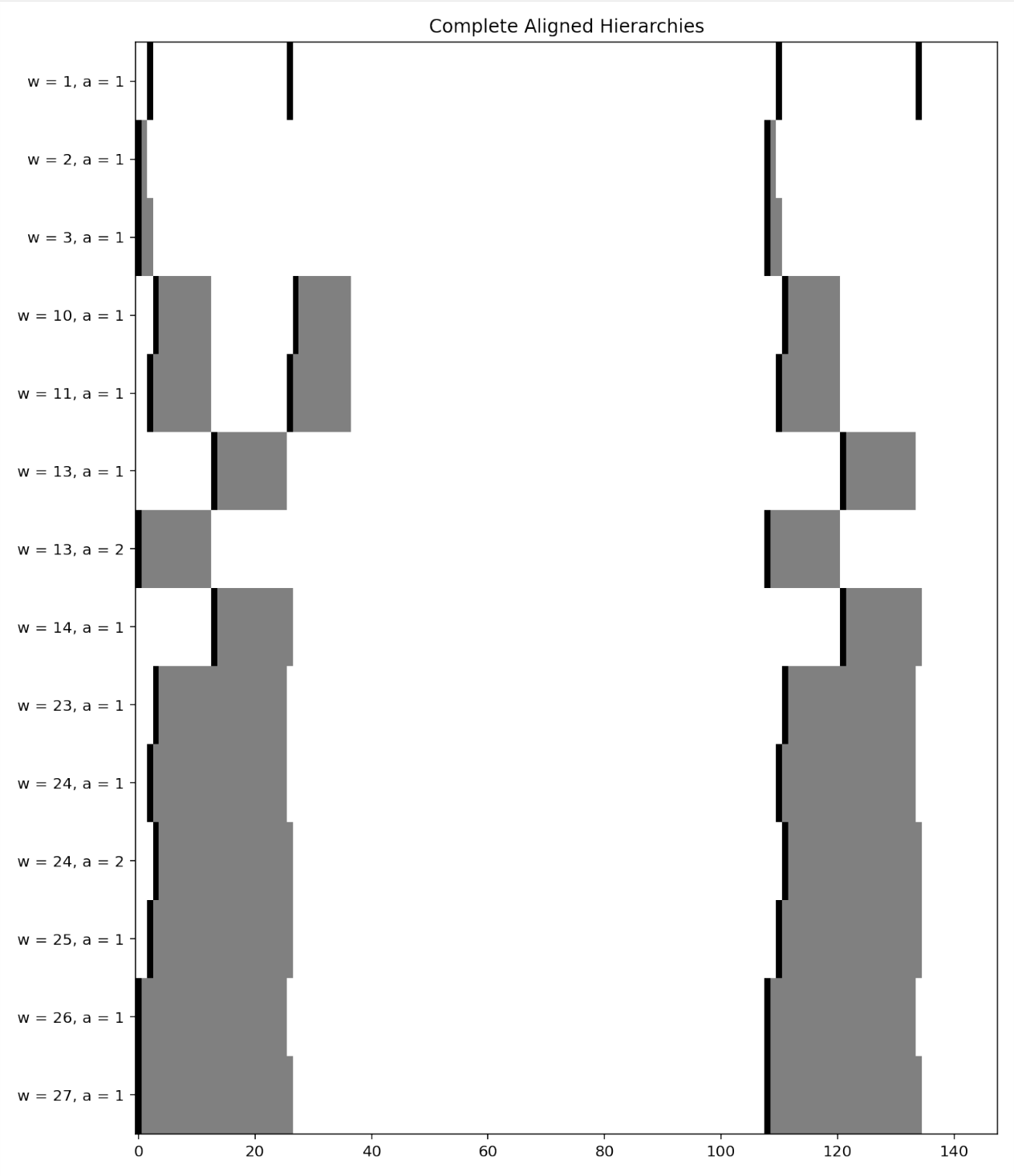
[2]:
```
file_in = load_ex_data('../repytah/data/mazurka30-1.csv').to_numpy()
file_out = "hierarchical_out_file.mat"
num_fv_per_shingle = 12
thresh = 0.02
csv_to_aligned_hierarchies(file_in, file_out, num_fv_per_shingle, thresh, True)
```


Essential Structure Components
w = 1, a = 1
w = 2, a = 1
w = 10, a = 1
w = 13, a = 1
0
20
40
60
80
100
120
140


Threshold Self-dissimilarity Matrix of the Ordering Essential Structure Components
0
1
2
3
4
5
6
7
8
9
10
11
12
0
1
2
3
4
5
6
7
8
9
10
11
12


Repeated Ordered Sublists of the Essential Structure Components
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 2, a = 1
w = 2, a = 2
w = 2, a = 3
w = 2, a = 4
w = 3, a = 1
w = 3, a = 2
w = 3, a = 3
w = 4, a = 1
w = 4, a = 2
w = 5, a = 1
0
1
2
3
4
5
6
7
8
9
10
11
12

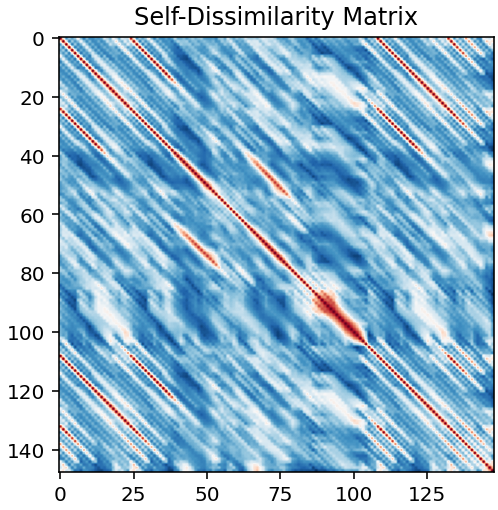

Repeated Ordered Sublists of the Essential Structure Components with Leading Index Highlighted
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 2, a = 1
w = 2, a = 2
w = 2, a = 3
w = 2, a = 4
w = 3, a = 1
w = 3, a = 2
w = 3, a = 3
w = 4, a = 1
w = 4, a = 2
w = 5, a = 1
0
1
2
3
4
5
6
7
8
9
10
11
12


Complete Aligned Hierarchies
w = 1, a = 1
w = 2, a = 1
w = 3, a = 1
w = 10, a = 1
w = 11, a = 1
w = 13, a = 1
w = 13, a = 2
w = 14, a = 1
w = 23, a = 1
w = 24, a = 1
w = 24, a = 2
w = 25, a = 1
w = 26, a = 1
w = 27, a = 1
0
20
40
60
80
100
120
140

### 1.9.2 To visualize the middle steps, try the following code:

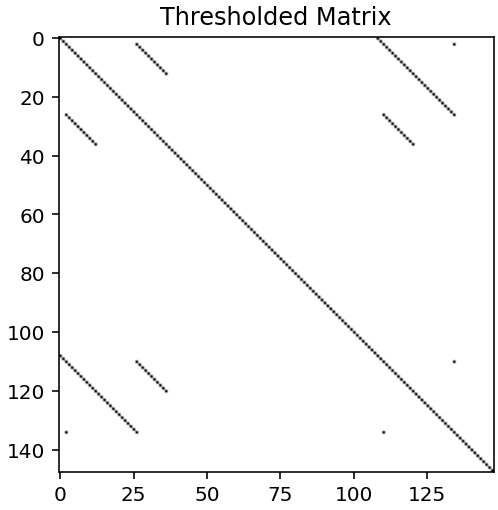
Load the input file and modify the music based data to a matrix representation:

[4]:
```
# Load the input file
file_in = load_ex_data('../repytah/data/mazurka30-1.csv').to_numpy()
fv_mat = file_in

# Number of feature vectors per shingle
num_fv_per_shingle = 12

# Create the self-dissimilarity matrix
self_dissim_mat = create_sdm(fv_mat, num_fv_per_shingle)

# Produce a visualization
SDM = plt.imshow(self_dissim_mat, cmap="RdBu")
plt.title('Self-Dissimilarity Matrix')
plt.show()
```




Threshold the above matrix:

[12]:
```
song_length = self_dissim_mat.shape[0]
thresh = 0.02

# Threshold the SDM to produce a binary matrix
thresh_dist_mat = (self_dissim_mat <= thresh)

# Produce a visualization
SDM = plt.imshow(thresh_dist_mat, cmap="Greys")
plt.title('Thresholded Matrix')
plt.show()
```

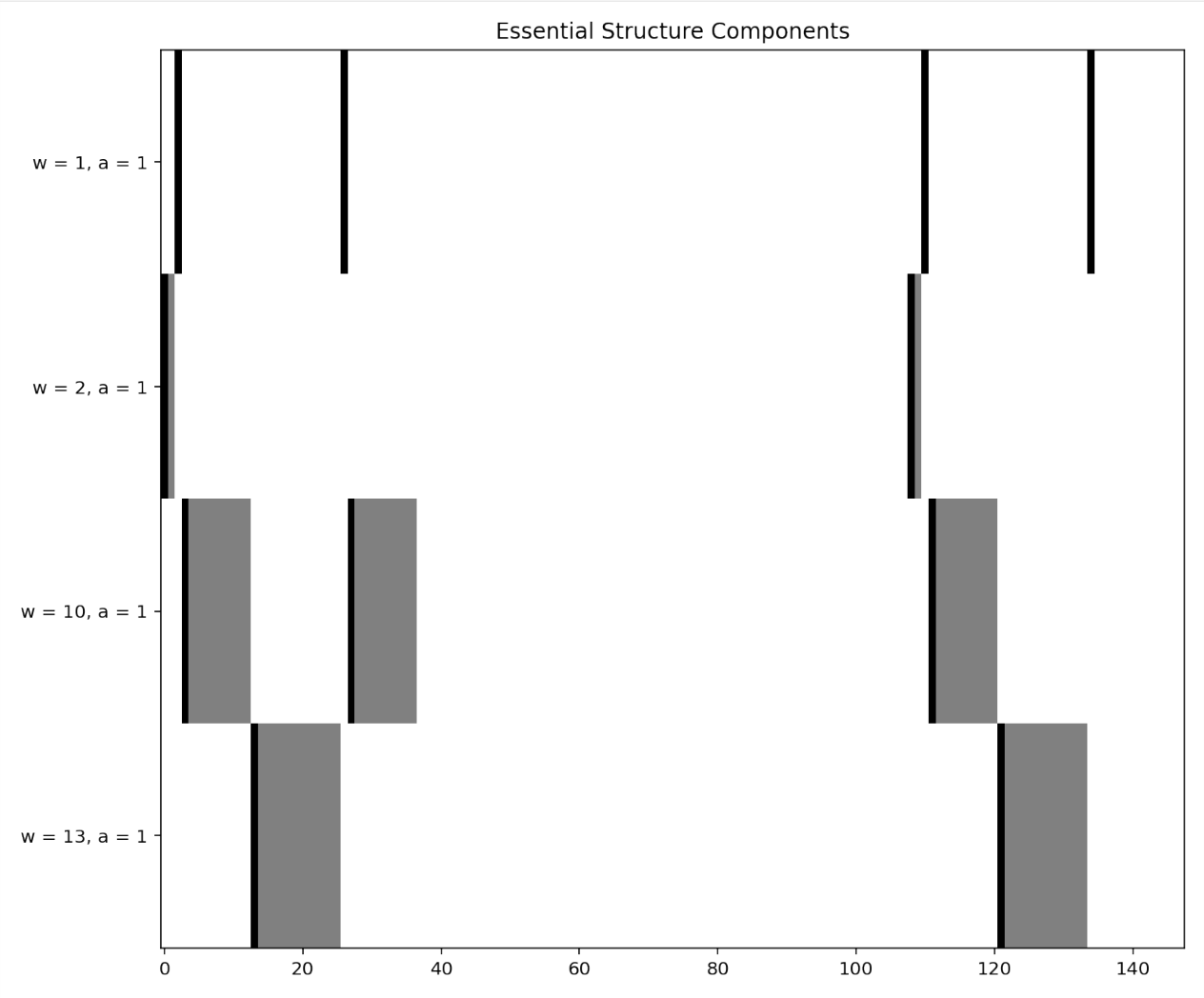


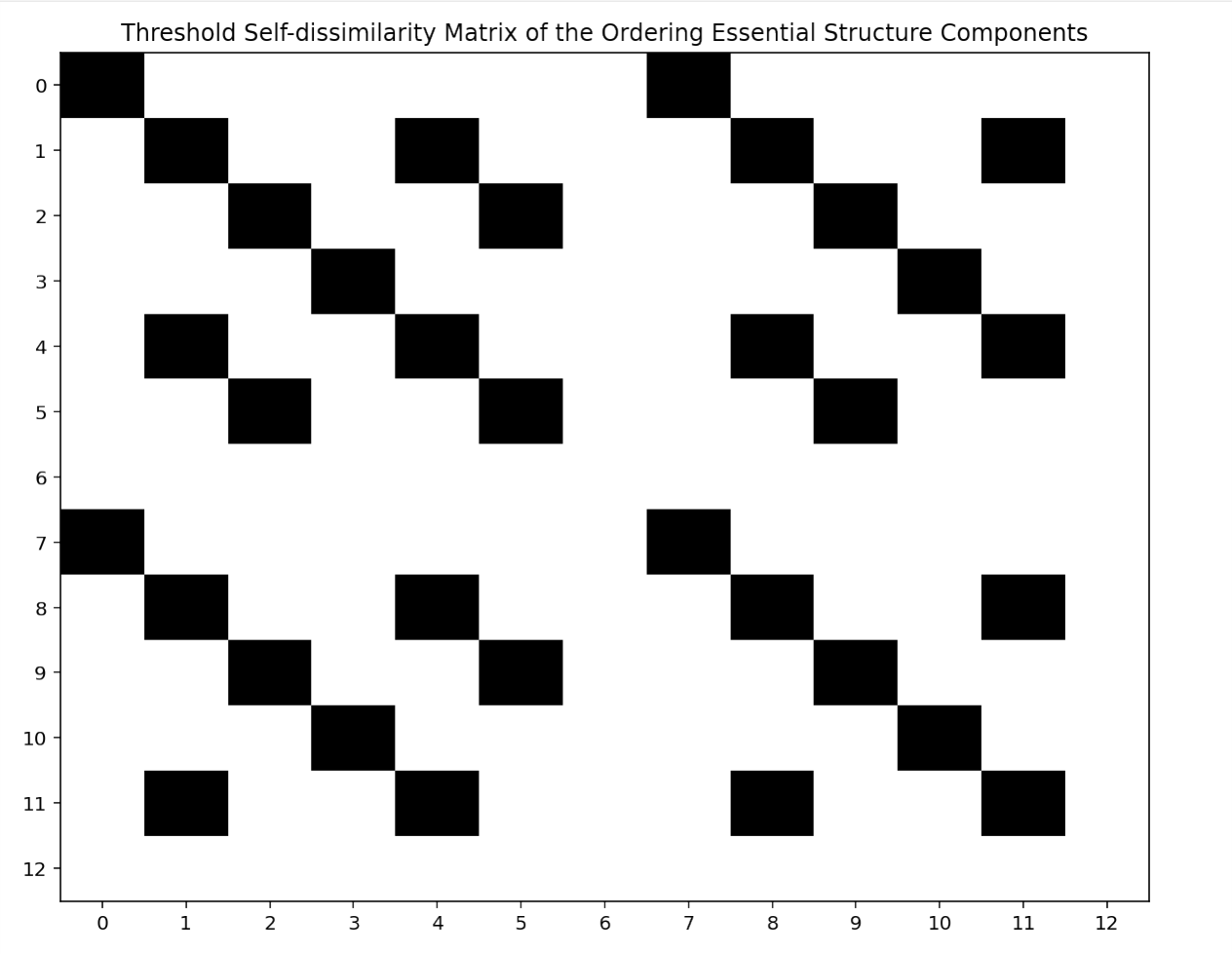

Find and group each pair of repeats:

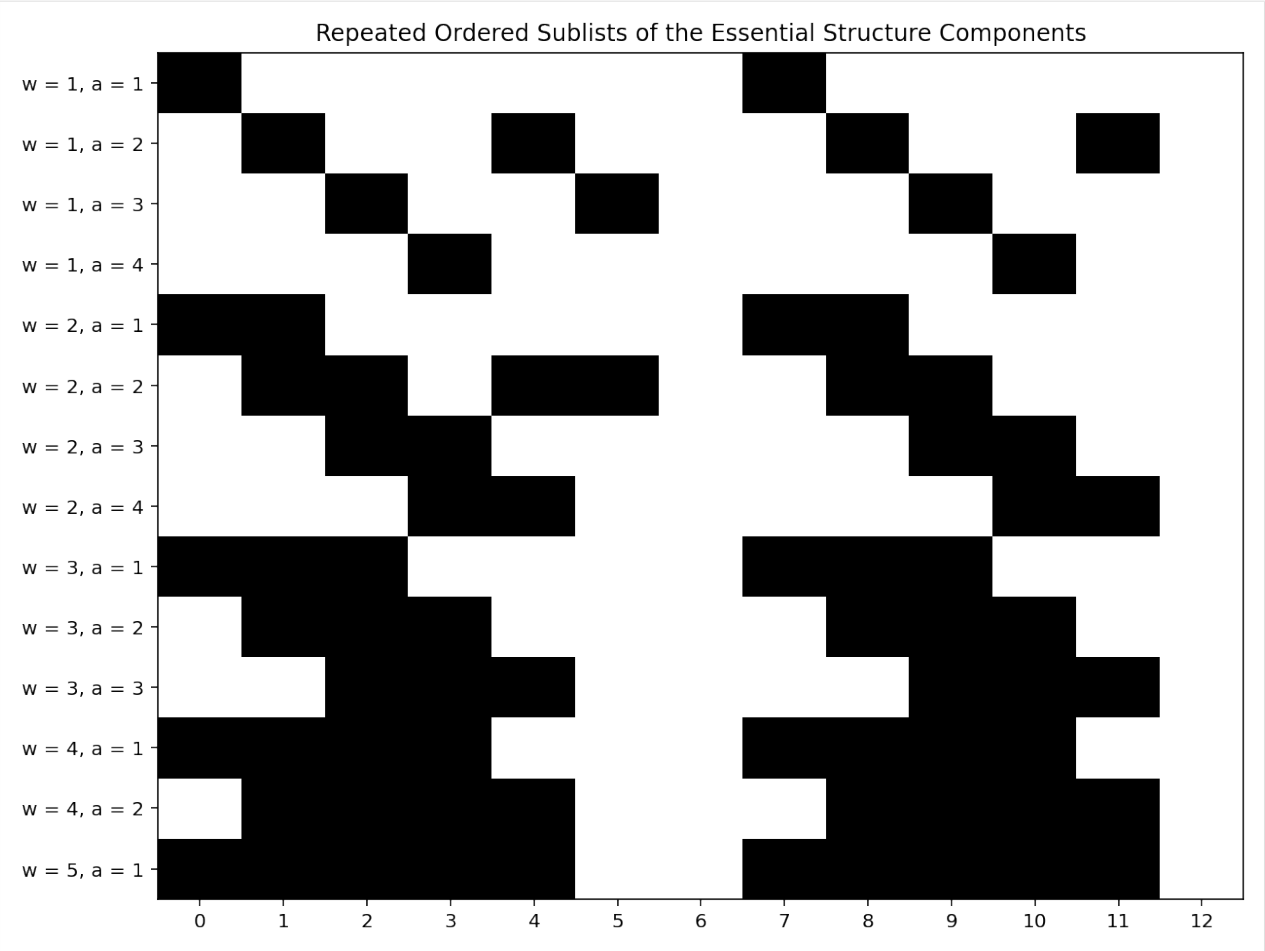
[13]:
```
all_lst = find_initial_repeats(thresh_dist_mat, np.arange(1, song_length + 1), 0)
complete_lst = find_complete_list(all_lst, song_length)
mat_no_overlaps, key_no_overlaps = remove_overlaps(complete_lst, song_length)[1:3]
```

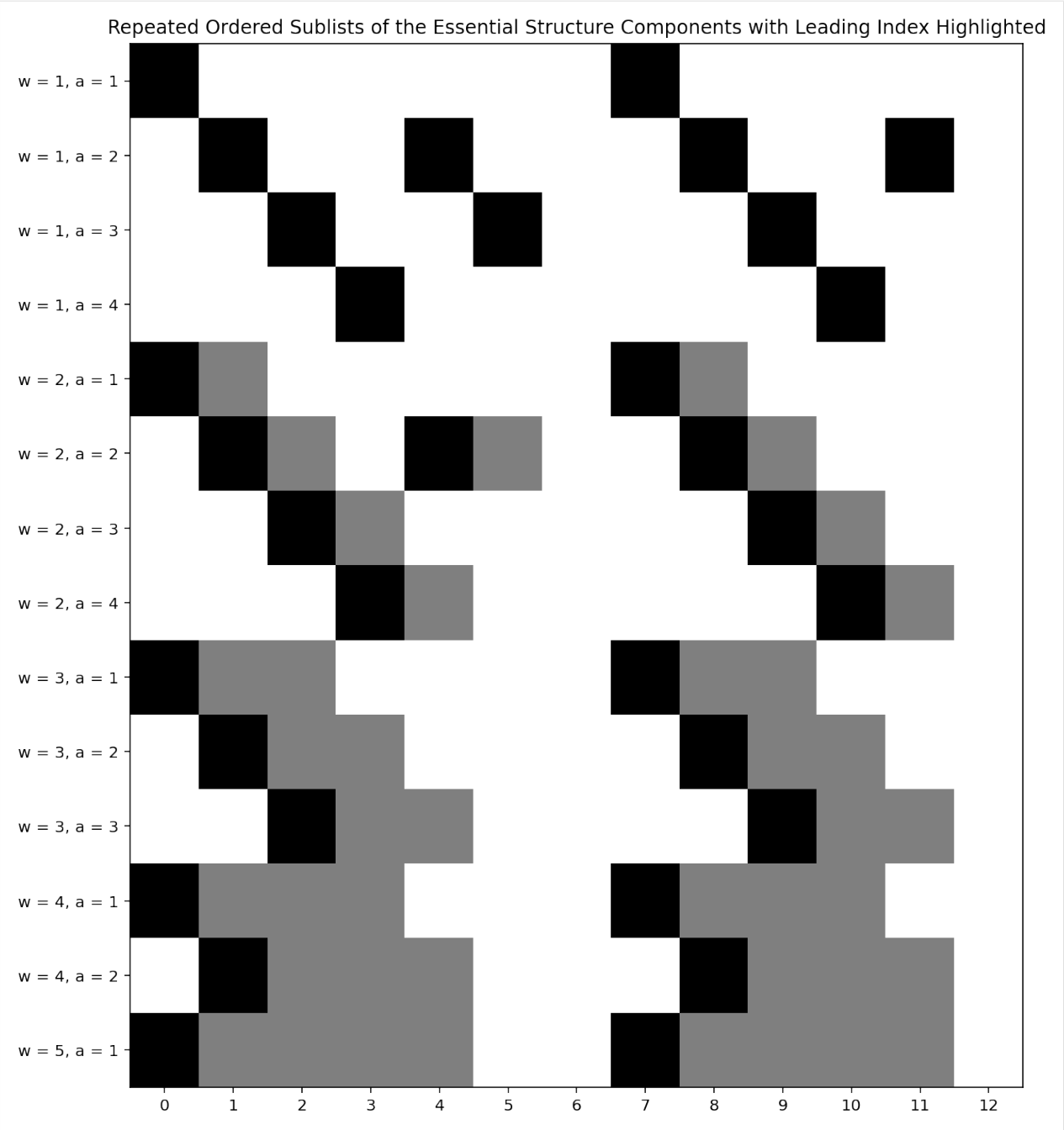
Find the essential structure components of the song and build the aligned hierarchies:

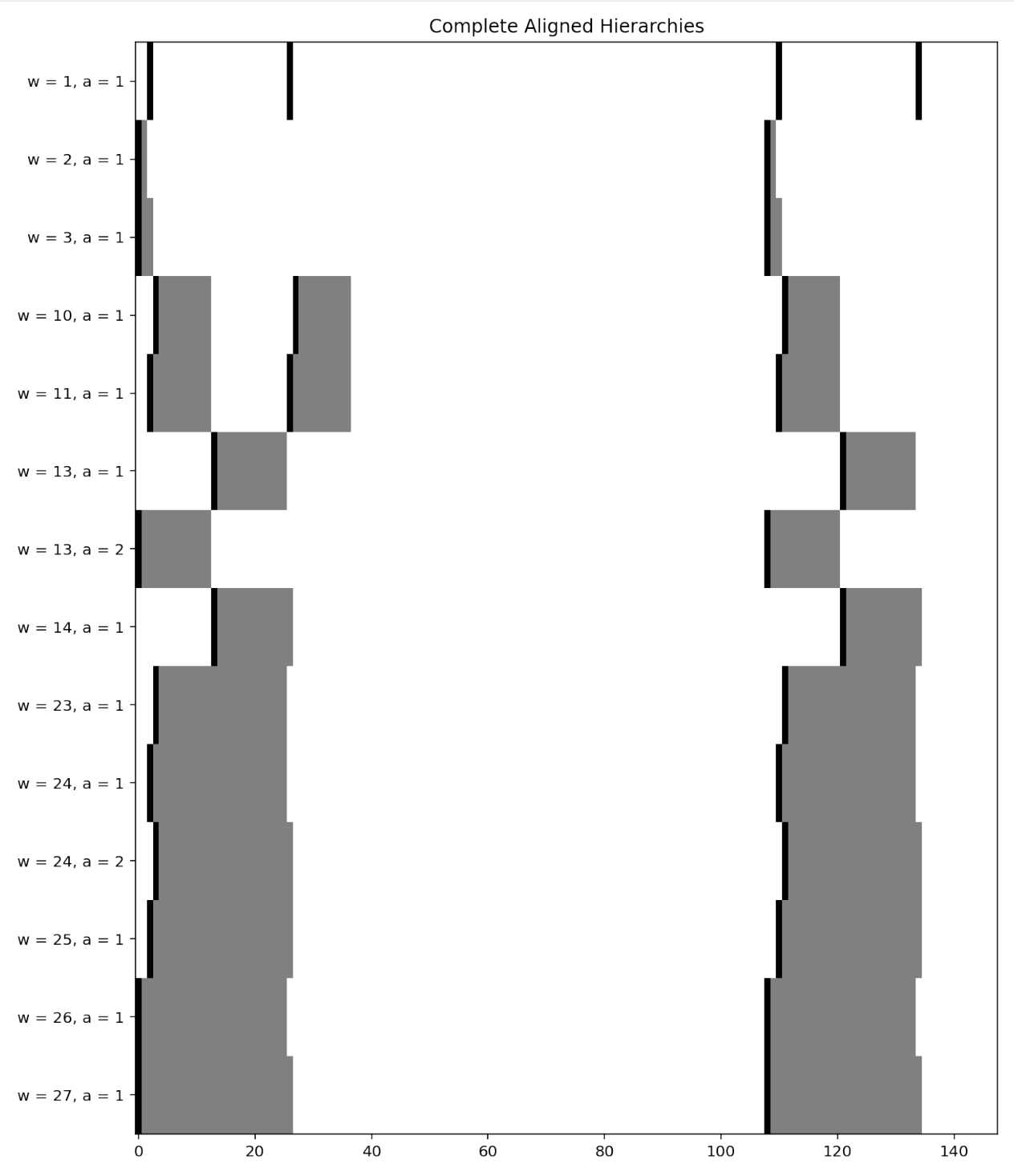
[14]:
```
output_tuple = hierarchical_structure(mat_no_overlaps, key_no_overlaps, song_length,␣
↪vis=True)
```


Essential Structure Components
w = 1, a = 1
w = 2, a = 1
w = 10, a = 1
w = 13, a = 1
0
20
40
60
80
100
120
140

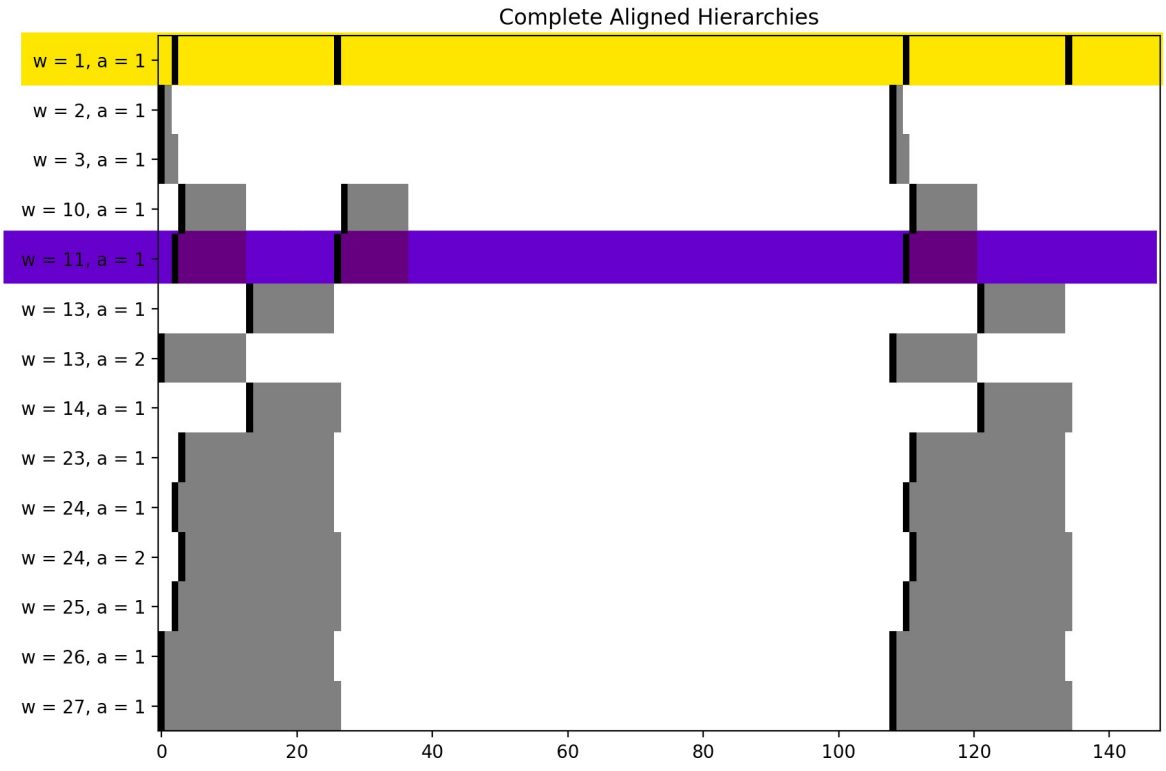

Threshold Self-dissimilarity Matrix of the Ordering Essential Structure Components
0
1
2
3
4
5
6
7
8
9
10
11
12
0
1
2
3
4
5
6
7
8
9
10
11
12

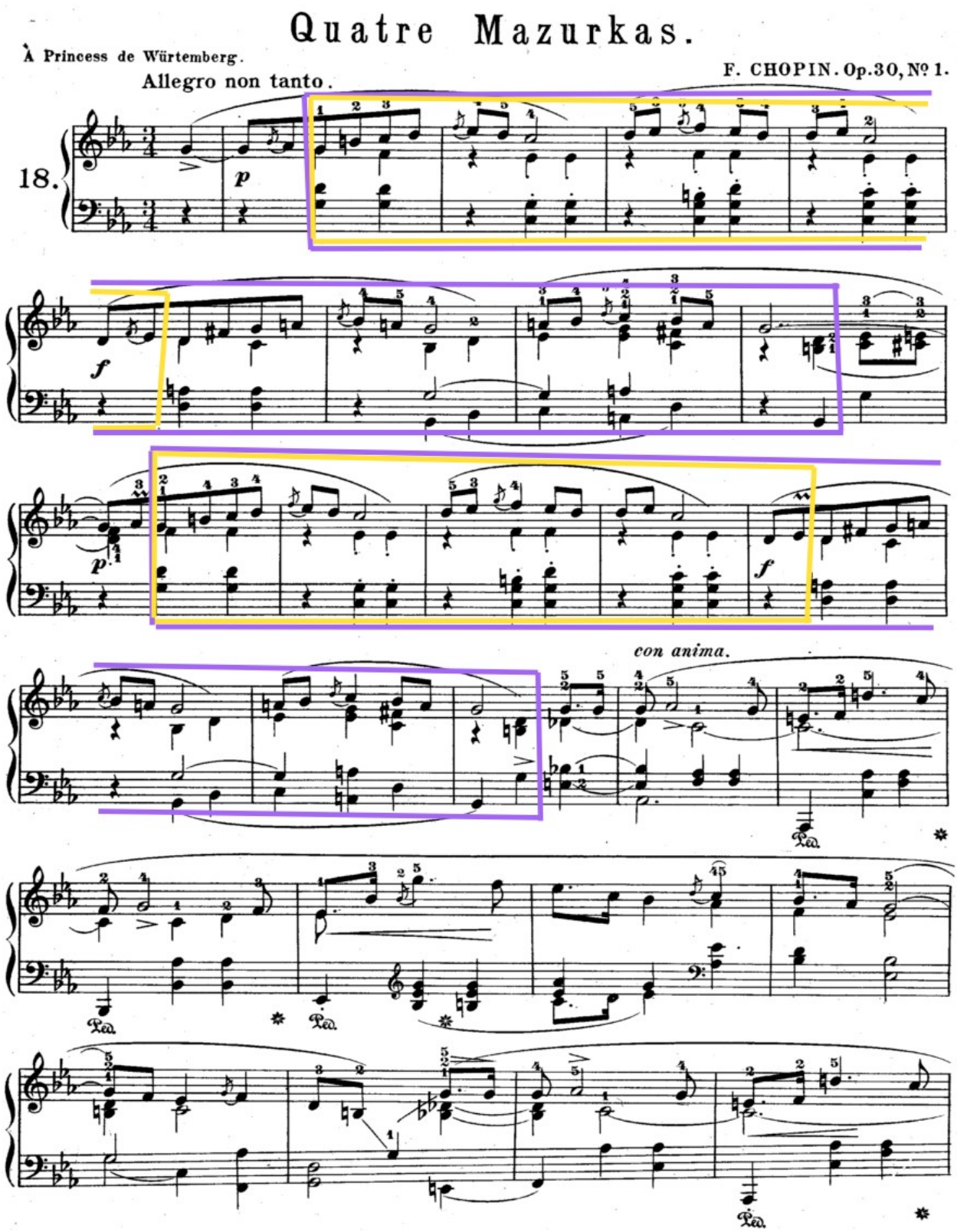

Repeated Ordered Sublists of the Essential Structure Components
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 2, a = 1
w = 2, a = 2
w = 2, a = 3
w = 2, a = 4
w = 3, a = 1
w = 3, a = 2
w = 3, a = 3
w = 4, a = 1
w = 4, a = 2
w = 5, a = 1
0
1
2
3
4
5
6
7
8
9
10
11
12

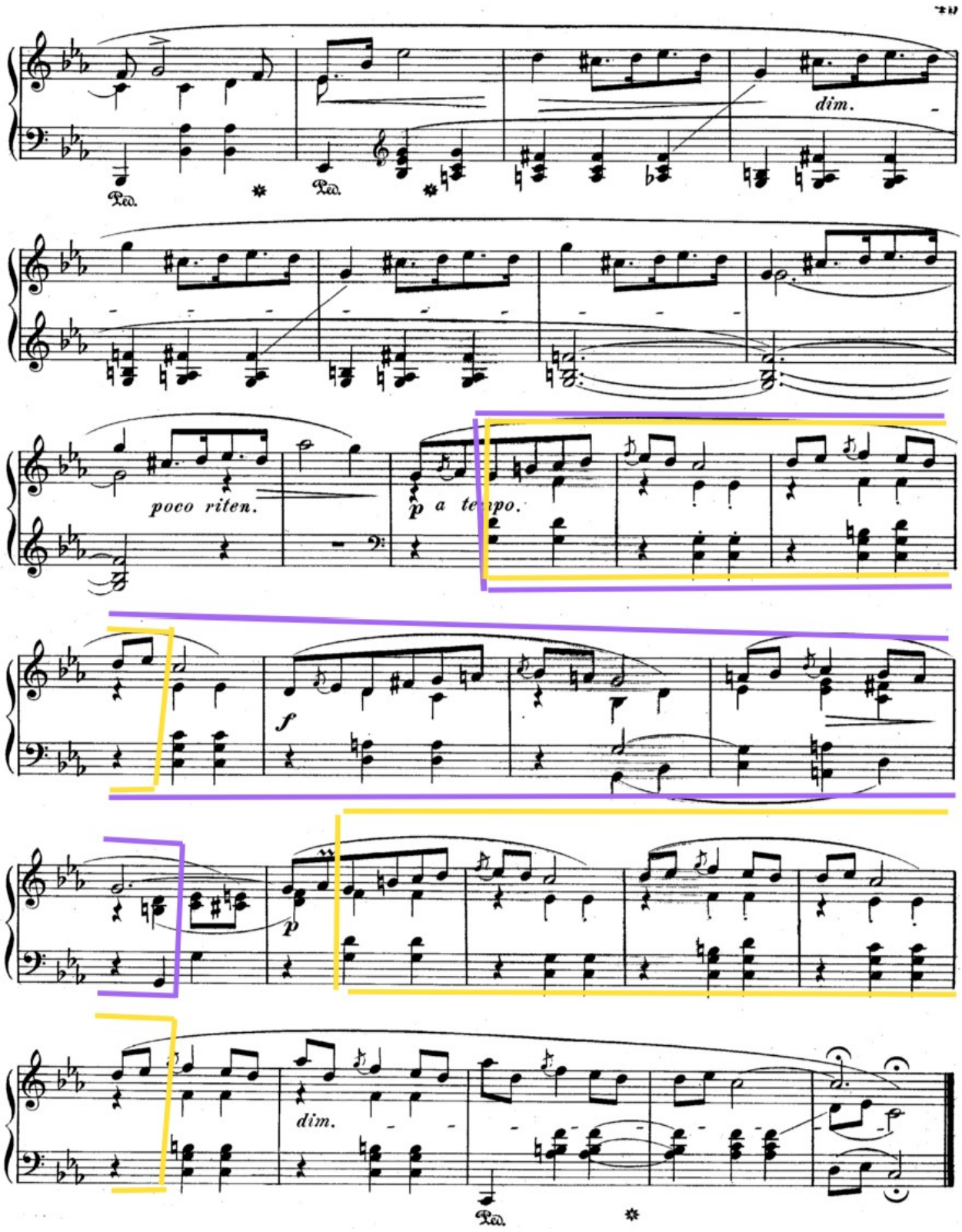

Repeated Ordered Sublists of the Essential Structure Components with Leading Index Highlighted
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 2, a = 1
w = 2, a = 2
w = 2, a = 3
w = 2, a = 4
w = 3, a = 1
w = 3, a = 2
w = 3, a = 3
w = 4, a = 1
w = 4, a = 2
w = 5, a = 1
0
1
2
3
4
5
6
7
8
9
10
11
12




To visualize the repeated structures in the actual music score, the examples in color yellow and purple are shown:


Complete Aligned Hierarchies
w = 1, a = 1
w = 2, a = 1
w = 3, a = 1
w = 10, a = 1
w = 11, a = 1
w = 13, a = 1
w = 13, a = 2
w = 14, a = 1
w = 23, a = 1
w = 24, a = 1
w = 24, a = 2
w = 25, a = 1
w = 26, a = 1
w = 27, a = 1
0
20
40
60
80
100
120
140


Quatre Mazurkas.
À Princess de Würtemberg.
F. CHOPIN. Op. 30, Nº 1.
Allegro non tanto.
18.
p
f
p
f
con anima.




This notebook is just a quick walk-through that demonstrates how to use the package `repytah`, more technical details are in the `example <https://github.com/smith-tinkerlab/repytah/blob/main/docs/example_vignette.ipynb>`__ notebook.

## 1.10 Example of Creating Aligned Hierarchies for a Mazurka Score

In this example, we will walk through the elements of the `repytah` package that pertain to the creation of aligned hierarchies for a music-based data stream (eg. a song).

Beginning with features (such as MFCCs and chroma features) for each timestep in your music-based data stream, there are several steps to this process:

0. Create the self-dissimilarity matrix (SDM).
1. Highlight pairs of timesteps that are close enough to be considered as repetitions of each other. (In other words, threshold the SDM)
2. Find pairs of structure repetitions (represented as diagonals within the thresholded SDM).
3. Find any pairs of structure repetitions not found in step 2, and group the structure repetitions.
4. Remove any repeated structures that have overlapped instances.
5. Distill the collection of repeated structures into the *essential structure components* , i.e. the smallest meaningful repetitions on which all larger repeats are constructed. Each timestep will be contained in no more than one essential structure component

*Note*: The walk-through of this example is very similar to the code in `example.py` found in this package.

We begin by importing the necessary packages:

[1]:
```
# NumPy and SciPy are required for mathematical operations
import scipy.io as sio
import numpy as np

# Pandas is used to import the csv
import pandas as pd

# Import all modules from repytah
from repytah import *

# Matplotlib is used to display outputs
import matplotlib.pyplot as plt

# Make the images clear
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

# Hide code (optional)
import ipywidgets as widgets
from IPython.display import display, HTML
javascript_functions = {False: "hide()", True: "show()"}
button_descriptions  = {False: "Show code", True: "Hide code"}
def toggle_code(state):
    output_string = "<script>$(\"div.input\").{}</script>"
    output_args   = (javascript_functions[state],)
    output        = output_string.format(*output_args)
    display(HTML(output))
def button_action(value):
    state = value.new
    toggle_code(state)
```

(continues on next page)

(continued from previous page)

```
    value.owner.description = button_descriptions[state]
state = True
toggle_code(state)
button = widgets.ToggleButton(state, description = button_descriptions[state])
button.observe(button_action, "value")
display(button)
```

```
<IPython.core.display.HTML object>
```

```
ToggleButton(value=True, description='Hide code')
```

### 1.10.1 Step 0: Create the SDM

In this phase, there are a few crucial details, namely importing the data file that we would like to unearth hierarchical structural information for and determining the appropriate dissimilarity measure to use. If you already have (symmetrical) matrix representations for your data stream, then you may find it more appropriate to load your matrix and then skip ahead to Step 1 or Step 2.

This step assumes that your music-based data stream (ie. a recording or score) has already had your preferred features extracted (like chroma or MFCCs) and is arranged into columns such that each column represents a time step (or beat). We refer to this as a *feature vector matrix* as each feature vector is laid out as column within one cohesive matrix.

#### Importing data for structure analysis

Note: For this demonstration, we are using the `load_ex_data()` built-in function from the `example` module to load our data. To recreate our example, you could do the same. This is an optional task and the normal method of reading in a data file will work.

We are using Chopin's Mazurka Op.6, No.1 as input for this demonstration.

[2]:
```
# Import csv
file_in = load_ex_data('data/input.csv').to_numpy()

fv_mat = file_in
```

#### Creating the SDM

In just one line, we define the self-dissimilarity matrix. This function `create_sdm`, uses feature vectors to create an audio shingle for each time step and represents these shingles as vectors by stacking the relevant feature vectors on top of each other. Then, the cosine distance is found between these shingles.

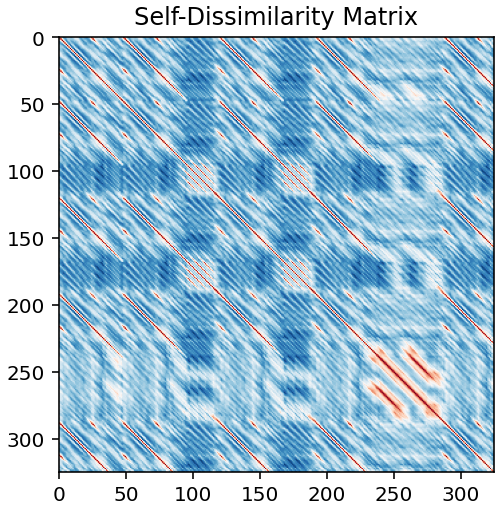
[3]:
```
# Number of feature vectors per shingle
num_fv_per_shingle = 12

# Create the self-dissimilarity matrix
self_dissim_mat = create_sdm(fv_mat, num_fv_per_shingle)

print('self_dissim_mat:\n', self_dissim_mat)

# Produce a visualization
SDM = plt.imshow(self_dissim_mat, cmap="RdBu")
```

(continues on next page)

(continued from previous page)

```
plt.title('Self-Dissimilarity Matrix')
plt.show()
```

```
self_dissim_mat:
 [[0.         0.36468911 0.58429924 ... 0.57425539 0.77909953 0.82434591]
 [0.36468911 0.         0.41660702 ... 0.68829818 0.64039838 0.80480641]
 [0.58429924 0.41660702 0.         ... 0.72072778 0.59670143 0.57191158]
 ...
 [0.57425539 0.68829818 0.72072778 ... 0.         0.60941866 0.70720396]
 [0.77909953 0.64039838 0.59670143 ... 0.60941866 0.         0.39416349]
 [0.82434591 0.80480641 0.57191158 ... 0.70720396 0.39416349 0.        ]]
```




### 1.10.2 Step 1: Threshold the SDM

In this step, the self-dissimilarity matrix is thresholded to produce a binary matrix of the same dimensions. This matrix is used to identify repeated structures, which are represented by diagonals of the same length.

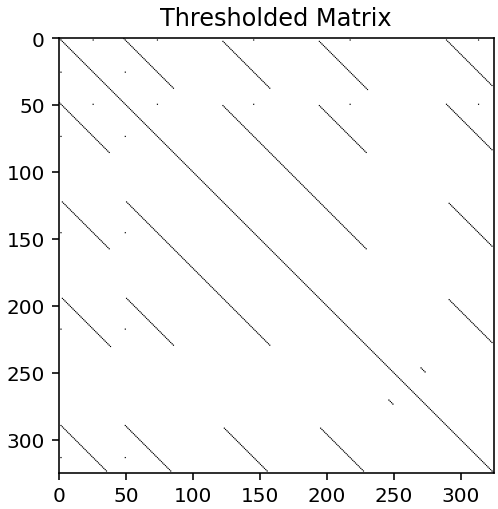
[4]:
```
song_length = self_dissim_mat.shape[0]
thresh = 0.02

# Threshold the SDM to produce a binary matrix
thresh_dist_mat = (self_dissim_mat <= thresh)

print('thresh_dist_mat:\n', thresh_dist_mat)

# Produce a visualization
SDM = plt.imshow(thresh_dist_mat, cmap="Greys")
plt.title('Thresholded Matrix')
plt.show()
```

```
thresh_dist_mat:
 [[ True False False ... False False False]
 [False  True False ... False False False]
 [False False  True ... False False False]
```

(continues on next page)

(continued from previous page)

```
...
[False False False ...  True False False]
[False False False ... False  True False]
[False False False ... False False  True]]
```




### 1.10.3 Step 2: Find the diagonals present and store all pairs of repeats found in a list

The diagonals in the thresholded matrix are found and recorded in an array of repeats. `find_initial_repeats` does this by looking for the largest repeated structures in **thresh_dist_mat**, which are illustrated in the above Thresholded Matrix diagram. Once all repeated structures are found, which are represented as diagonals present in **thresh_dist_mat**, they are stored with their start/end indices and lengths in a list. As each diagonal is found, it is removed to avoid identifying repeated sub-structures.

Below is the listing of the pairs of repeats found by `find_initial_repeats`:

[5]:
```
all_lst = find_initial_repeats(thresh_dist_mat, np.arange(1, song_length + 1), 0)

print('all_lst:\n', all_lst)
```

```
all_lst:
 [[  2   2  26  26   1]
 [  2   2  74  74   1]
 [  2   2 146 146   1]
 [  2   2 218 218   1]
 [  2   2 314 314   1]
 [ 26  26  50  50   1]
 [ 50  50  74  74   1]
 [ 50  50 146 146   1]
 [ 50  50 218 218   1]
 [ 50  50 314 314   1]
 [247 250 271 274   4]
 [124 156 292 324  33]
 [196 228 292 324  33]
 [  2  36 290 324  35]
```

(continues on next page)

(continued from previous page)

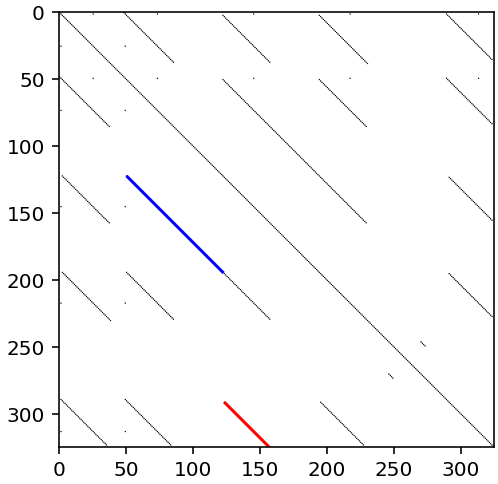
```
[ 50  84 290 324  35]
[  3  38 123 158  36]
[ 51  86 195 230  36]
[  3  39 195 231  37]
[  1  38  49  86  38]
[ 51 122 123 194  72]
[ 87 158 159 230  72]
[  1 325   1 325 325]]
```

Let's take moment to examine what we have and time it back to the original thresholded SDM. The first ten pairs listed in **all_lst** are repeats of length 1. We also note that the last "pair" of repeats is the whole score being matched to itself. For now, we set this "repeat" and the ones that are of length 1 aside.

In the image below, for each pair of repeats, we color one of the two diagonals associated to that pairing. The second one is the matching diagonal when flipped over the main diagonal.

For example, the red diagonal represents the pair of repeats that start at beats 124 and 292 and are 33 beats long, and the blue diagonal represents the pair of repeats that start at beat 51 and 123 and are 72 beats long.

[8]:
```
visualize_all_lst(thresh_dist_mat)
```




### 1.10.4 Step 3: Find any diagonals in the thresholded matrix T that are contained in larger diagonals in T but not found in step 2, then group pairs of repeats

Any smaller diagonals that are contained in larger diagonals and are not found in step 2 are found and added to the array of repeats in this step. All possible repeat lengths are looped over in **all_lst**, and larger repeats containing smaller repeats are broken up into up to 3 sections each, the part before the overlap, the overlap, and the part after the overlap. With this, a more complete list of repeated structures is created.

[9]:
```
complete_lst = find_complete_list(all_lst, song_length)

print('complete_lst:\n', complete_lst)
```

```
complete_lst:
 [[  1   1  49  49   1   1]
 [  2   2  26  26   1   2]
```

(continues on next page)

(continued from previous page)

```
[  2   2  50  50   1   2]
[  2   2  74  74   1   2]
[  2   2 146 146   1   2]
[  2   2 218 218   1   2]
[  2   2 290 290   1   2]
[  2   2 314 314   1   2]
[ 26  26  50  50   1   2]
[ 26  26  74  74   1   2]
[ 26  26 146 146   1   2]
[ 26  26 218 218   1   2]
[ 26  26 314 314   1   2]
[ 50  50  74  74   1   2]
[ 50  50 146 146   1   2]
[ 50  50 218 218   1   2]
[ 50  50 290 290   1   2]
[ 50  50 314 314   1   2]
[ 74  74 146 146   1   2]
[ 74  74 218 218   1   2]
[ 74  74 314 314   1   2]
[146 146 218 218   1   2]
[146 146 314 314   1   2]
[218 218 314 314   1   2]
[  3   3 123 123   1   3]
[  3   3 195 195   1   3]
[ 51  51 123 123   1   3]
[ 51  51 195 195   1   3]
[ 39  39 231 231   1   4]
[  1   2  49  50   2   1]
[  2   3 290 291   2   2]
[ 50  51 290 291   2   2]
[ 37  38  85  86   2   3]
[ 37  38 157 158   2   3]
[ 85  86 229 230   2   3]
[157 158 229 230   2   3]
[ 37  39 229 231   3   1]
[247 250 271 274   4   1]
[ 27  36 315 324  10   1]
[ 75  84 315 324  10   1]
[147 156 315 324  10   1]
[219 228 315 324  10   1]
[ 27  38  75  86  12   1]
[ 27  38 147 158  12   1]
[ 75  86 219 230  12   1]
[147 158 219 230  12   1]
[ 27  39 219 231  13   1]
[124 145 292 313  22   1]
[196 217 292 313  22   1]
[  3  25 123 145  23   1]
[  3  25 195 217  23   1]
[ 51  73 123 145  23   1]
[ 51  73 195 217  23   1]
[  2  25 290 313  24   1]
```

(continues on next page)

(continued from previous page)

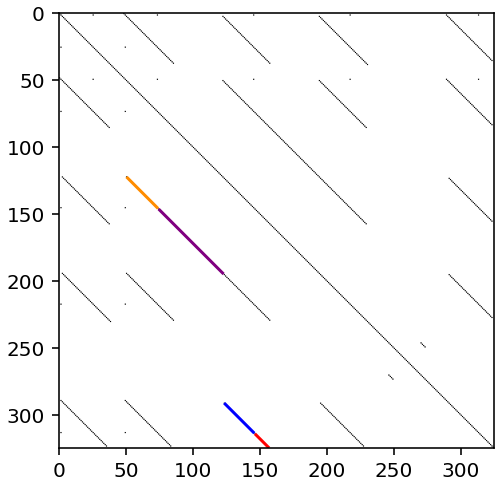
```
[ 50  73 290 313  24   1]
[  1  25  49  73  25   1]
[  4  36 124 156  33   1]
[  4  36 196 228  33   1]
[  4  36 292 324  33   1]
[ 52  84 124 156  33   1]
[ 52  84 196 228  33   1]
[ 52  84 292 324  33   1]
[124 156 196 228  33   1]
[124 156 292 324  33   1]
[196 228 292 324  33   1]
[  3  36 291 324  34   1]
[ 51  84 291 324  34   1]
[  2  36  50  84  35   1]
[  2  36 290 324  35   1]
[ 50  84 290 324  35   1]
[  3  38  51  86  36   1]
[  3  38 123 158  36   1]
[  3  38 195 230  36   1]
[ 51  86 123 158  36   1]
[ 51  86 195 230  36   1]
[123 158 195 230  36   1]
[ 87 122 159 194  36   2]
[  3  39 195 231  37   1]
[ 87 123 159 195  37   2]
[  1  38  49  86  38   1]
[ 85 122 157 194  38   2]
[ 75 122 147 194  48   1]
[ 87 145 159 217  59   1]
[ 51 122 123 194  72   1]
[ 87 158 159 230  72   2]]
```

It is clear that **complete_list** is much longer that **all_lst**, which makes sense because we are adding smaller pieces of the larger repeats, when an overlap in time has occurred between a smaller repeat and a larger one.

For example, the repeat of length 72 starting at beat 51 overlaps with the smaller repeat of length 48 starting at beat 75 (the purple section).

The image below visualizes the idea of distilling smaller parts from larger repeats in step 2 by showing some of the smaller repetitions found within the larger repeats.

[10]:
```
visualize_complete_lst(thresh_dist_mat)
```




### 1.10.5 Step 4: Remove any repeated structure that has at least two repeats that overlap in time

In this step, repeated structures with the same annotation and length are removed if they are overlapping.

This is done by looping over all possible repeat lengths, finding all the groups of repeats of the same length. For each of those groups, `remove_overlaps` determines whether there exists any pair of repeats that overlaps in time. If a pair like that exists, all the overlapping repeats are removed.

Along with a list of all the repeats with no overlaps, this step also forms a matrix, a list of the associated lengths of the repeats in the matrix, the annotations of the repeats in the matrix, and a list of the overlaps. The matrix is a binary matrix that visualizes the repeats, representing the start of a repeat with a 1. This matrix, in combination with the list of the lengths of the repeats, can be used to visualize the repeats.

[11]:
```
output_tuple = remove_overlaps(complete_lst, song_length)

print('List with no overlaps:\n', output_tuple[0])
print('Matrix with no overlaps:\n', output_tuple[1])
print('Lengths of the repeats in the matrix:', output_tuple[2])
print('Annotations of the repeats in the matrix:', output_tuple[3])
print('List of overlaps:\n', output_tuple[4])
```

```
List with no overlaps:
 [[  1   1  49  49   1   1]
 [  2   2  26  26   1   2]
 [  2   2  50  50   1   2]
 [  2   2  74  74   1   2]
 [  2   2 146 146   1   2]
 [  2   2 218 218   1   2]
 [  2   2 290 290   1   2]
 [  2   2 314 314   1   2]
 [  3   3 123 123   1   3]
 [  3   3 195 195   1   3]
 [ 26  26  50  50   1   2]
 [ 26  26  74  74   1   2]
 [ 26  26 146 146   1   2]
```

(continues on next page)

(continued from previous page)

```
[ 26  26 218 218   1   2]
[ 26  26 314 314   1   2]
[ 39  39 231 231   1   4]
[ 50  50  74  74   1   2]
[ 50  50 146 146   1   2]
[ 50  50 218 218   1   2]
[ 50  50 290 290   1   2]
[ 50  50 314 314   1   2]
[ 51  51 123 123   1   3]
[ 51  51 195 195   1   3]
[ 74  74 146 146   1   2]
[ 74  74 218 218   1   2]
[ 74  74 314 314   1   2]
[146 146 218 218   1   2]
[146 146 314 314   1   2]
[218 218 314 314   1   2]
[  1   2  49  50   2   1]
[  2   3 290 291   2   2]
[ 37  38  85  86   2   3]
[ 37  38 157 158   2   3]
[ 50  51 290 291   2   2]
[ 85  86 229 230   2   3]
[157 158 229 230   2   3]
[ 37  39 229 231   3   1]
[247 250 271 274   4   1]
[ 27  36 315 324  10   1]
[ 75  84 315 324  10   1]
[147 156 315 324  10   1]
[219 228 315 324  10   1]
[ 27  38  75  86  12   1]
[ 27  38 147 158  12   1]
[ 75  86 219 230  12   1]
[147 158 219 230  12   1]
[ 27  39 219 231  13   1]
[124 145 292 313  22   1]
[196 217 292 313  22   1]
[  3  25 123 145  23   1]
[  3  25 195 217  23   1]
[ 51  73 123 145  23   1]
[ 51  73 195 217  23   1]
[  2  25 290 313  24   1]
[ 50  73 290 313  24   1]
[  1  25  49  73  25   1]
[  4  36 124 156  33   1]
[  4  36 196 228  33   1]
[  4  36 292 324  33   1]
[ 52  84 124 156  33   1]
[ 52  84 196 228  33   1]
[ 52  84 292 324  33   1]
[124 156 196 228  33   1]
[124 156 292 324  33   1]
[196 228 292 324  33   1]
```

(continues on next page)

(continued from previous page)

```
 [  3  36 291 324  34   1]
 [ 51  84 291 324  34   1]
 [  2  36  50  84  35   1]
 [  2  36 290 324  35   1]
 [ 50  84 290 324  35   1]
 [  3  38  51  86  36   1]
 [  3  38 123 158  36   1]
 [  3  38 195 230  36   1]
 [ 51  86 123 158  36   1]
 [ 51  86 195 230  36   1]
 [ 87 122 159 194  36   2]
 [123 158 195 230  36   1]
 [  3  39 195 231  37   1]
 [ 87 123 159 195  37   2]
 [  1  38  49  86  38   1]
 [ 85 122 157 194  38   2]
 [ 75 122 147 194  48   1]
 [ 87 145 159 217  59   1]
 [ 51 122 123 194  72   1]
 [ 87 158 159 230  72   2]]
Matrix with no overlaps:
 [[1 0 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [0 0 1 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
Lengths of the repeats in the matrix: [ 1  1  1  1  2  2  2  3  4 10 12 13 22 23 24 25␣
↪33 34 35 36 36 37 37 38
 38 48 59 72 72]
Annotations of the repeats in the matrix: [1 2 3 4 1 2 3 1 1 1 1 1 1 1 1 1 1 1 1 2 1 2␣
↪1 2 1 1 1 2]
List of overlaps:
 []
```

### 1.10.6 Step 5: Find the essential structure components of the song and build the aligned hierarchies

Building the aligned hierarchies takes two final steps: finding the essential structure components and using them to create the full aligned hierarchies. `hierarchical_structure` first finds the essential structure components with `breakup_overlaps_by_intersect`. After this, modified versions of steps 2-4 take place to create the full hierarchical structure.

### Essential Structure Components

The essential structure components are the structure building blocks; that is they are the smallest meaningful repeated structures. All repeated structures within a piece are constructed with right and left unions of the essential structure components. Each timestep can be in at most **one** essential structure component. The first panel in the below image shows the essential structure components for our example piece. The visualization is organized such that there is one row per type of repeat. Notice that we have nine types of repeats in the below example.

### Creating the aligned hierarchies

After finding the essential structure components for a song, we build the aligned hierarchies using a process whose result is akin to taking right and left unions of the essential structure components. The process begins by creating a list of the essential structure components in the order they appear. We assign each essential structure component the number of the row that it sits in. For consectutive timesteps where there is not essential structure component, we use 0 for that block. In the below example, this list would be: 1,2,3,4,2,5,6,7,0,1,2,3,4,2,5,6,8,3,4,2,5,6,8,3,4,2,5,6,7,0,9,0,9,0,2,3,4,2,5,0.

Using this list, a thresholded dissimilarity matrix is created. Using steps similar to steps two through four above, we can extract information about the combinations of essential structures. From the thresholded dissimilarity matrix, all diagonals are extracted, starting with the longest one, but are not removed. This way, all possible combinations of essential structure components are found. Then, all these combinations are grouped and checked for overlapping repeated combinations. The final product is then restructured to show the widths and annotations for each type of repeated structure (represented by each row).

The figures below represent the final result we get, including two of the most important images: the essential structure components (top most panel) and the complete hierarchical structure (bottom panel).

The figure below with the name *Threshold Self-dissimilarity matrix of the ordering Essential Structure Components* shows the square thresholded self-dissimilarity matrix such that the (i,j) entry is 1 if the following three conditions are true:

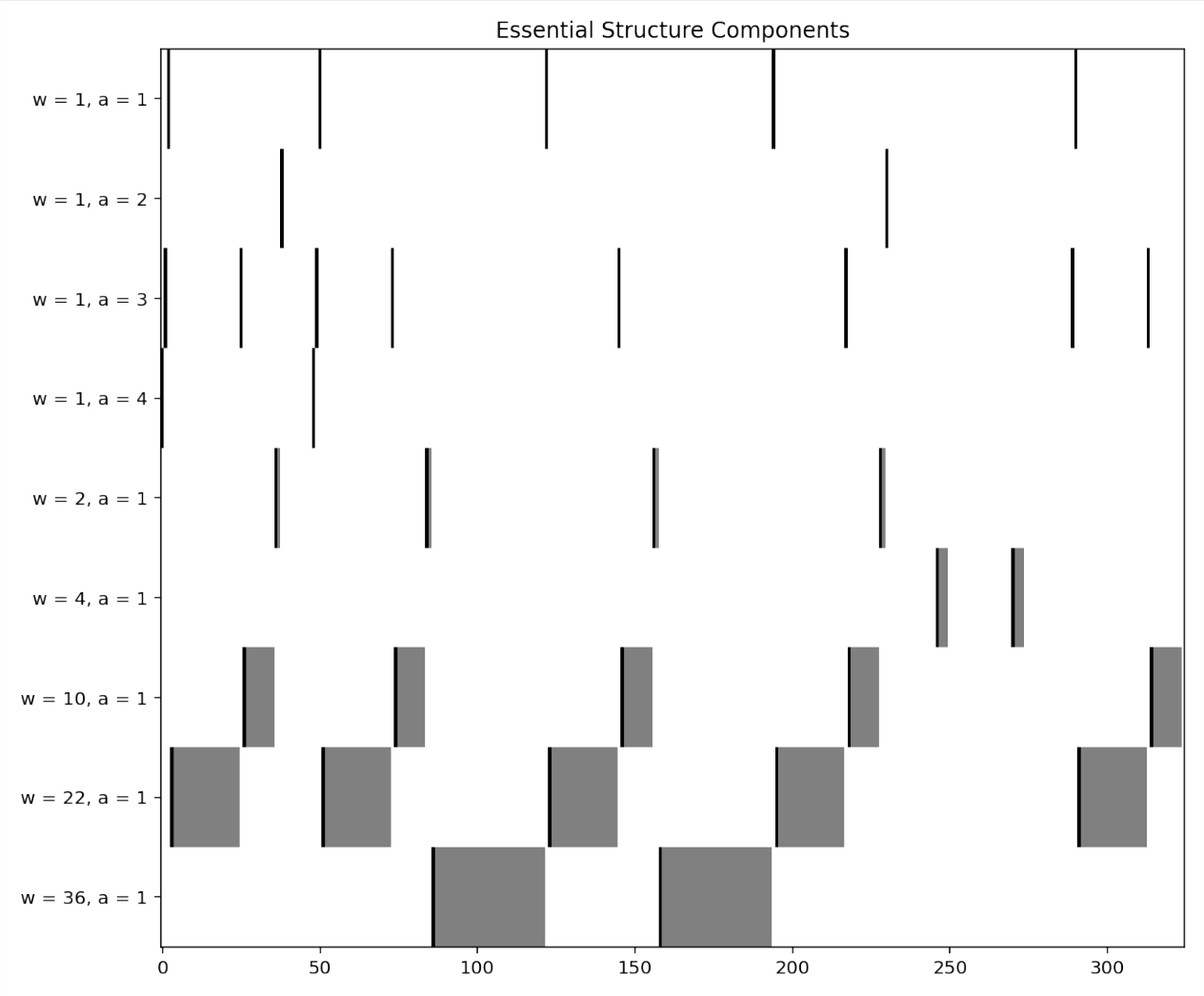
- A repeat of an essential structure component is the i-th item in the ordering.
- A repeat of an essential structure component is the j-th item in the ordering.
- The repeat occurring in the i-th place of the ordering and the one occurring in the j-th place of the ordering are repeats of the same essential structure component.

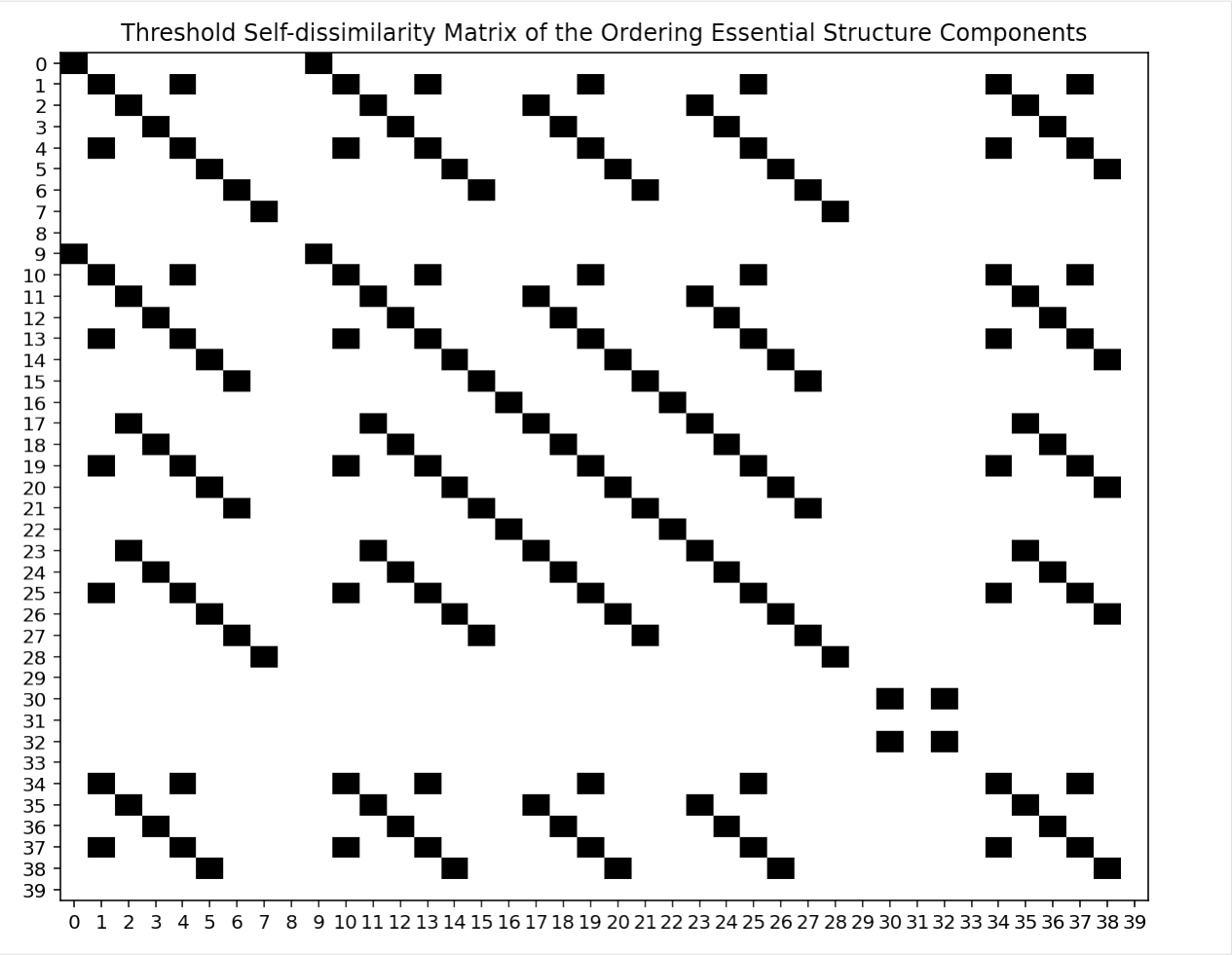
The figure below with the name *Repeated ordered sublists of the Essential Structure Components* is the result of extracting all the diagonals and getting pairs of repeated ordered sublists of the essential structure components. The repetitive copies of the same repeat and overlaps are also removed.

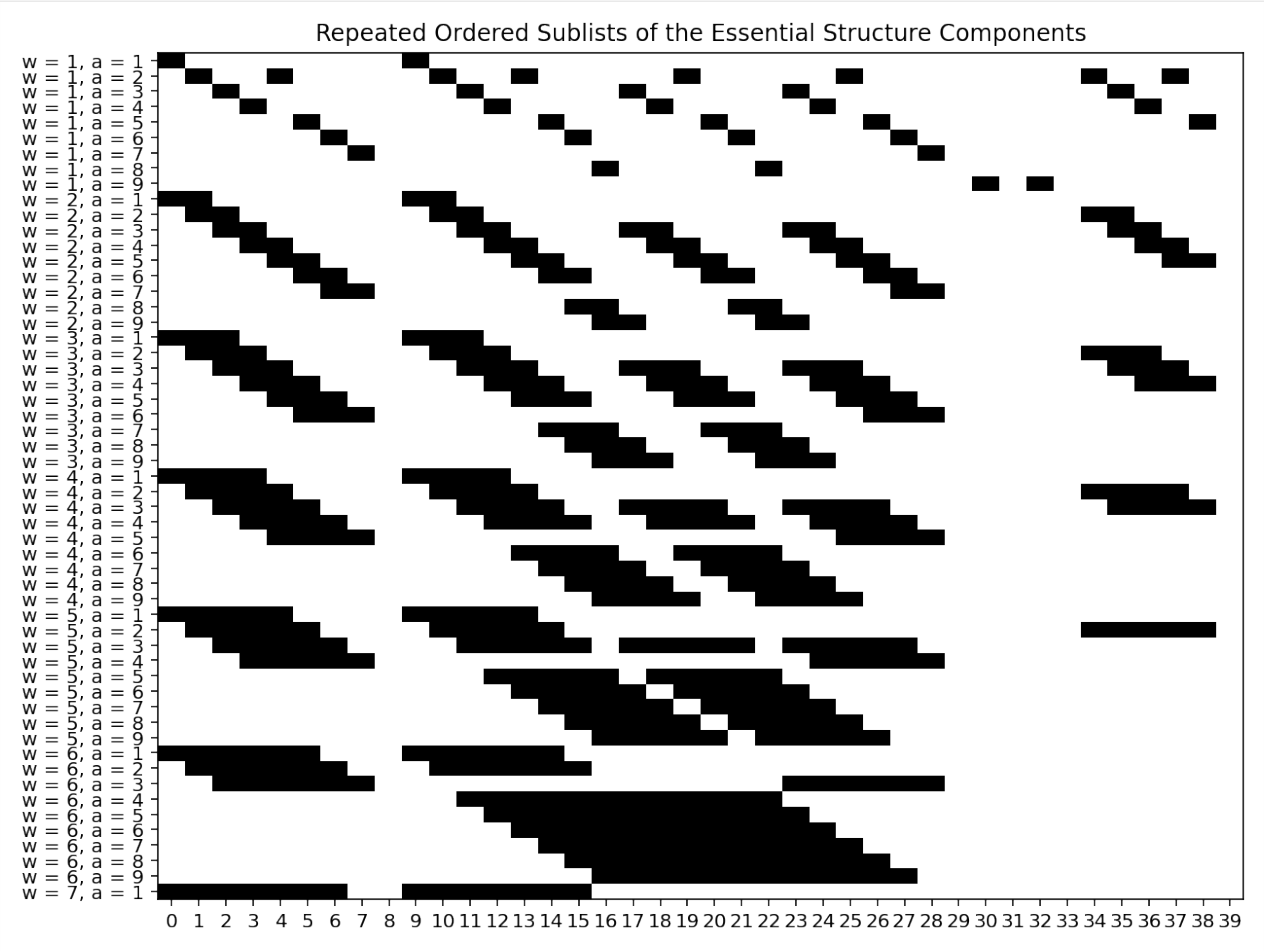
The second to last figure labeled *Repeated ordered sublists of the Essential Structure Components with leading index highlighted* contains the same repeats as *Repeated ordered sublists of the Essential Structure Components* but only notes the starting item in black and shows the rest of the repeat in gray.

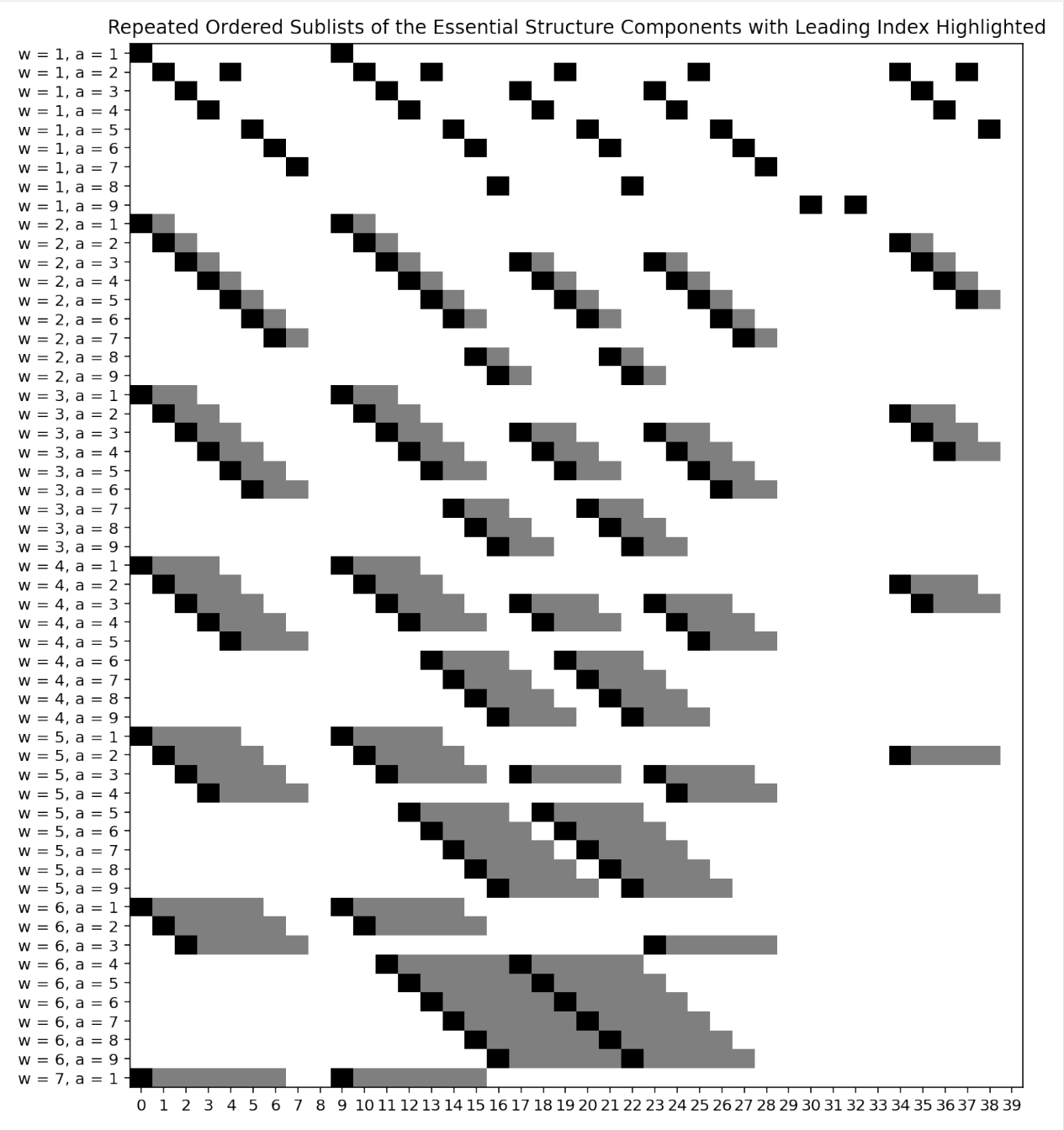
The aligned hierarchies are shown in the last image. This is the result of stretching each essential structure component to its correct length (ie. the one noted in the first image below). This transformation results in a visualization that is the number of time steps of the original song.

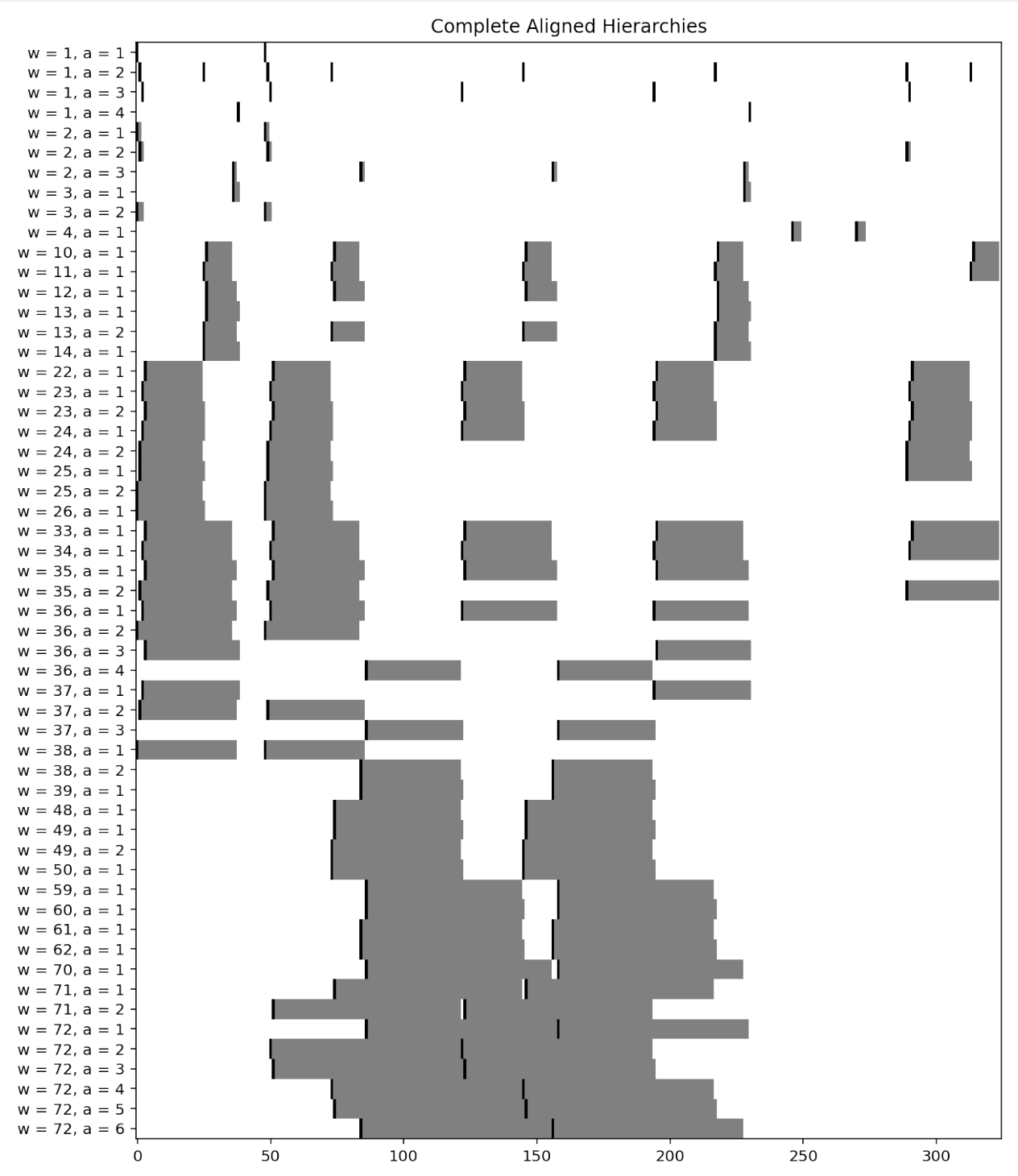
[12]:
```
(mat_no_overlaps, key_no_overlaps) = output_tuple[1:3]

# Distill non-overlapping repeats into essential structure components and
# use them to build the hierarchical representation
output_tuple = hierarchical_structure(mat_no_overlaps, key_no_overlaps, song_length,␣
↪vis=True)
```


Essential Structure Components
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 2, a = 1
w = 4, a = 1
w = 10, a = 1
w = 22, a = 1
w = 36, a = 1
0
50
100
150
200
250
300


Threshold Self-dissimilarity Matrix of the Ordering Essential Structure Components
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39


Repeated Ordered Sublists of the Essential Structure Components
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 1, a = 5
w = 1, a = 6
w = 1, a = 7
w = 1, a = 8
w = 1, a = 9
w = 2, a = 1
w = 2, a = 2
w = 2, a = 3
w = 2, a = 4
w = 2, a = 5
w = 2, a = 6
w = 2, a = 7
w = 2, a = 8
w = 2, a = 9
w = 3, a = 1
w = 3, a = 2
w = 3, a = 3
w = 3, a = 4
w = 3, a = 5
w = 3, a = 6
w = 3, a = 7
w = 3, a = 8
w = 3, a = 9
w = 4, a = 1
w = 4, a = 2
w = 4, a = 3
w = 4, a = 4
w = 4, a = 5
w = 4, a = 6
w = 4, a = 7
w = 4, a = 8
w = 4, a = 9
w = 5, a = 1
w = 5, a = 2
w = 5, a = 3
w = 5, a = 4
w = 5, a = 5
w = 5, a = 6
w = 5, a = 7
w = 5, a = 8
w = 5, a = 9
w = 6, a = 1
w = 6, a = 2
w = 6, a = 3
w = 6, a = 4
w = 6, a = 5
w = 6, a = 6
w = 6, a = 7
w = 6, a = 8
w = 6, a = 9
w = 7, a = 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39


Repeated Ordered Sublists of the Essential Structure Components with Leading Index Highlighted
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 1, a = 5
w = 1, a = 6
w = 1, a = 7
w = 1, a = 8
w = 1, a = 9
w = 2, a = 1
w = 2, a = 2
w = 2, a = 3
w = 2, a = 4
w = 2, a = 5
w = 2, a = 6
w = 2, a = 7
w = 2, a = 8
w = 2, a = 9
w = 3, a = 1
w = 3, a = 2
w = 3, a = 3
w = 3, a = 4
w = 3, a = 5
w = 3, a = 6
w = 3, a = 7
w = 3, a = 8
w = 3, a = 9
w = 4, a = 1
w = 4, a = 2
w = 4, a = 3
w = 4, a = 4
w = 4, a = 5
w = 4, a = 6
w = 4, a = 7
w = 4, a = 8
w = 4, a = 9
w = 5, a = 1
w = 5, a = 2
w = 5, a = 3
w = 5, a = 4
w = 5, a = 5
w = 5, a = 6
w = 5, a = 7
w = 5, a = 8
w = 5, a = 9
w = 6, a = 1
w = 6, a = 2
w = 6, a = 3
w = 6, a = 4
w = 6, a = 5
w = 6, a = 6
w = 6, a = 7
w = 6, a = 8
w = 6, a = 9
w = 7, a = 1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39


Complete Aligned Hierarchies
w = 1, a = 1
w = 1, a = 2
w = 1, a = 3
w = 1, a = 4
w = 2, a = 1
w = 2, a = 2
w = 2, a = 3
w = 3, a = 1
w = 3, a = 2
w = 4, a = 1
w = 10, a = 1
w = 11, a = 1
w = 12, a = 1
w = 13, a = 1
w = 13, a = 2
w = 14, a = 1
w = 22, a = 1
w = 23, a = 1
w = 23, a = 2
w = 24, a = 1
w = 24, a = 2
w = 25, a = 1
w = 25, a = 2
w = 26, a = 1
w = 33, a = 1
w = 34, a = 1
w = 35, a = 1
w = 35, a = 2
w = 36, a = 1
w = 36, a = 2
w = 36, a = 3
w = 36, a = 4
w = 37, a = 1
w = 37, a = 2
w = 37, a = 3
w = 38, a = 1
w = 38, a = 2
w = 39, a = 1
w = 48, a = 1
w = 49, a = 1
w = 49, a = 2
w = 50, a = 1
w = 59, a = 1
w = 60, a = 1
w = 61, a = 1
w = 62, a = 1
w = 70, a = 1
w = 71, a = 1
w = 71, a = 2
w = 72, a = 1
w = 72, a = 2
w = 72, a = 3
w = 72, a = 4
w = 72, a = 5
w = 72, a = 6
0
50
100
150
200
250
300

## 1.11 Changelog

### 1.11.1 v0.1.0

Initial public release.

## 1.12 Index

:ref:genindex

# PYTHON MODULE INDEX

## r

# INDEX

## Symbols



## A



## B



## C



## F



## G



## H



## M



## R



## S