Frequent Sequence Mining

In this tutorial, you will implement missing parts of two simple algorithms for frequent sequence mining. There are many variants of this problem (one sequence / multiple sequences, various definitions of frequency etc). We will restrict our attention to the case when we have one sequence and frequency of a subsequence is defined as the number of occurrences of the subsequence divided by the maximum possible number of occurrences, which is for a subsequence of length m and f occurrences in a sequence of length n equal to $\frac{f}{n-m+1}$.

Algorithm

We will use tha algorithm for frequent sequence mining using canonical codes which was presented at the lectures. The following Matlab code is listed in general form, i.e. it is not restricted to directed or undirected sequence mining. (Do not worry that the code is not commented. A commented version is available in the zip file with Matlab files for this tutorial.) Note that we will use direct representation of the sequences in the code - the theory of canonical codes presented in the lectures is used only indirectly here as we use it only as a recipe how to extend candidate frequent subsequences.

```
function freqPatterns = aprioriFPM(data, sMin, initializeWords,...
    expandWords, findSupport, sortedAlphabet, prune)

freqPatterns = {};

sortedA = sortedAlphabet(data);
words0 = initializeWords(sortedA);

words = {};
for i = 1:size(words0,2)
    s = findSupport(words0{i}, data);
    words0{i}.support = s;
    if s ≥ sMin
```

```
        words{end+1} = words0{i};
    end
end

% we repeat this loop as long as new frequent patterns are found
while ¬isempty(words)
    for i = 1:size(words,2)
        freqPatterns{end+1} = words{i};
    end
    wordCand = expandWords(words,sortedA)
    % checks if all subsequences are frequent (i.e. present in words)
    wordCand = prune(words, wordCand);

    words = {};
    for i = 1:length(wordCand)
        s = findSupport(wordCand{i},data);
        wordCand{i}.support = s;
        if s≥sMin
            words{end+1} = wordCand{i};
        end
    end
end
```

Implementation

Your task is to implement function *expandSequences(oldPatterns, alphabet)* to files *directedSequenceMining.m* a *undirectedSequenceMining.m*.

```
function frequentPatterns = directedSequenceMining(data, sMin)

frequentPatterns = aprioriFPM(data, sMin, @initializeWords,...
    @expandSequences, @findSupport, @sortAlphabet, @prune);

function emptyWordArray = initializeWords(sortedAlphabet)
emptyWord.sequence = {};
emptyWordArray = {emptyWord};

function nextPatterns = expandSequences(oldPatterns, alphabet)
% zacatek kodu, ktery studenti mit nebudou

% konec kodu, ktery studenti mit nebudou


function support = findSupport(pattern, data)
% ...
```

```
function sortedAlphabet = sortAlphabet(data)
sortedAlphabet = sort(unique(data.sequence));

function filtered = prune(shorter, longer)
% ...
```

In case of directed sequences, function *expandSequences(oldPatterns, alphabet)* expects on its input a cell array containing frequent sequences of length n and an alphabet of used symbols (which will be created for you by function *sortAlphabet(data)*) and it returns a cell array containing candidate sequences of length $n + 1$. The following piece of code demonstrates a way to construct a new subsequence from old shorter subsequence in Matlab by adding one symbol at the end of the old sequence.

```
    newPattern.sequence = [oldPattern.sequence { 'a'} ]
```

You will implement function *expandSequences(oldPatterns, alphabet)* twice - once for directed sequence mining and once for undirected sequence mining. The skeleton of the main algorithm and description above correspond to the directed sequence mining whereas the code below corresponds to the undirected version.

```
function frequentPatterns = undirectedSequenceMining(data, sMin)

frequentPatterns = aprioriFPM(data, sMin, @initializeWords,...
    @expandSequences, @findSupport, @sortAlphabet, @prune);

function initial = initializeWords(sortedAlphabet)

emptyWord.sequence = {};
emptyWord.symmetryFlag = 1;
initial = {emptyWord};
for i = 1:size(sortedAlphabet,2)
    iw.sequence = {sortedAlphabet{i}};
    iw.symmetryFlag = 1;
    initial{end+1} = iw;
end

function nextPatterns = expandSequences(oldPatterns, alphabet)
% zacatek kodu, ktery studenti mit nebudou

% konec kodu, ktery studenti mit nebudou


function support = findSupport(pattern, data)
% ...
```

```
function sortedAlphabet = sortAlphabet(data)
sortedAlphabet = sort(unique(data.sequence));

function filtered = prune(shorter, longer)
% ...
```

You can notice that there is a difference in functions *initilizeWords* in case of directed and undirected sequences. This difference is due to the fact that the canonical codes for directed and undirected sequences are different. The first difference is that there is an attribute called *symmetryFlag* which has value 1 if the given (sub)sequence is a palindrome and is 0 otherwise. The second difference is that we first generate sequences of length 0 (i.e. the empty sequence), which we also do in case of directed sequences, but also all sequences of length 1 in case of undirected sequences. The reason is that, in case of undirected sequences, we create new sequences by adding two letters - one at the end and one at the beginning, i.e. sequences of length $n+1$ and $n+2$ are created from sequences of length $n-1$ and n respectively.

How You Should Implement This...

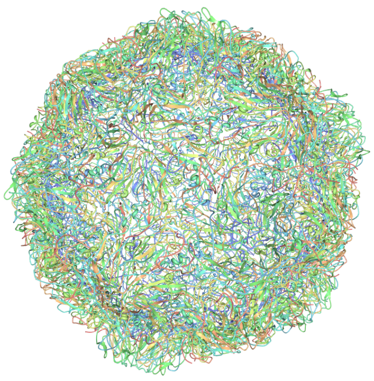
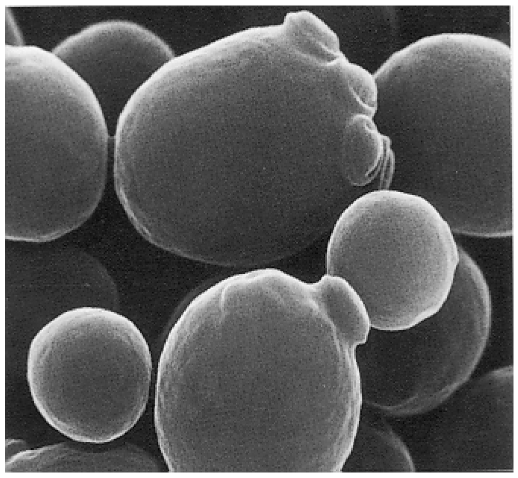
... you have learnt this at the lecture. There, you have learnt how canonical codes for directed and undirected sequences look like. Except that, you have learnt how we can exploit these canonical codes for effective generation of subsequences. You can find several useful tips how to work with so called *cell arrays* and *structures* at the end of this document.

Test Data (DNA of a Virus and Yeast)

Use the algorithms to find frequent sequences (directed and undirected) in the DNA of virus *Equine rhinitis A* (Figure 1) and yeast *Saccharomyces cerevisiae* (specifically its chromosome no III). When debugging your code, you may want to use only a subset of data.

Load data using Matlab command *load_dna*, which loads the DNA sequence of the virus into variable *virus_dna* and the DNA sequence of the yeast into variable *saccharomyces_dna*. Below you can see an example how to use the algorithms on the virus sequence.

```
load_dna;
dna_data.sequence = virus_dna;
```

Obrázek 1: *Left:* Spatial structure of protein envelope of virus *Equine rhinitis A. Right:* Yeast *Saccharomyces cerevisiae.*

```
%% directed sequence mining

seqs = directedSequenceMining(dna_data, 0.01);
fprintf('Printing results of directed sequence mining...\n');
printResults(seqs, 3);
```

Try to discuss differences between frequent sequences of the virus and the yeast (Note that yeasts are eucaryotic organisms and the complementary DNA bases are A-T and C-G).

Expected Result

You should obtain the next four frequent oriented subsequences of length 4 for minimum frequency 0.01: *attt*, *ttga*, *tttg*, *tttt*. Similarly, you should get the next two frequent subsequences of length 4 for frequency 0.01: *gttt*, *tttt*.

Several useful tips for working with structures and ”cell arrays”in Matlab

If you have not encountered *structures* or *cell arrays* in Matlab, you might find the next few tips useful.

Cell arrays

Cell arrays differ from matrices mainly in that they can contain not just numbers, but also strings, structures, matrices or even other nested cell arrays. An empty cell array may be constructed e.g. as `a = {};`.

The entries of a cell array are accessed as follows: `a = {'a', 'b', 'c'}; x = a{1}; % x = 'a'`. Notice that we use curly brackets for accessing cell array entries. If we used curly brackets instead of square brackets here, we would not get a cell array with four entries but a cell array containing the two nested cell arrays.

Structures

Structures in Matlab are very flexible. For example

```
data.sequence = {'a','b','c'};
```

creates a structure containing one cell array called *sequence* (it is not necessary to define a structure contents apriori). It is possible to add any number of other items to a structure. The items in a structure are then accessed easily, e.g. $x = data.sequence$.