Structure of the library

The library is separated to three folders: Applications, Core and Utils. Applications are features built on top of the core based on callbacks. Core is the actual core of the library, it includes the iterate-function and all the functions and data structures needed for it to work. Utils have useful functions and data structures. Bit vector and wavelet tree data structures are used by all parts of the library.

There isn't a straight connection between the Applications and Core folders. They work together through a callback system:

Features need to have a function to be used as a function pointer that takes iterator state and a results pointer as parameters. This function will be used by the iterate function which iterates through the strings and upon finding an interesting node in the string, makes a callback to the aforementioned callback function of the feature.

For now, features also have printing methods that are done with the mapper. It deconstructs the bwt back to the original string and makes a mapping table based on a bit vector of interesting indexes in the BWT.

Individual files

Structs:

Interval
A starting and ending index in a BWT.

Substring
A representation of a substring in a string based on BWTs.

Has an interval in the normal BWT.
Has also an interval in the BWT of the reverse of the string.
And length of the substring.

Triplet

This struct contains information about a certain substring which has some interesting feature, for example it is a MUM. The triplet has starting positions of the substring in 2 different input strings and the length of the substring.

Core:

backward_search

Function to extend a substring by one character to the left. It takes in an interval and outputs another. Used widely throughout the library, most notably by iterate and MEMs.

BWT_to_s

Takes as input a BWT (wavelet tree) and returns the original string from which the BWT was created. Notice that this function really returns a string, not a wavelet tree.

c_array

A c-array is needed in a backward search. Every index in the c-array corresponds a distinct letter in an original string, and they are in alphabetical order. Every index has the number of letters of the original string that sort alphabetically before the index letter.

iterate

The single most important file in the library. When given one string, it iterates over all right maximal substrings in the string. When given two strings it goes through all substrings that occur in both strings.

The iterate-function is given a callback function that it calls when it finds a right maximal substring (in the case of single-iterate) or when it finds a substring that occurs in both strings (in the case of double-iterate).

The iterate function starts with an interval of the whole string which it adds to a stack. Then it extends the interval to the left with every possible character. For each extension, it checks if it is right maximal (it occurs at least twice in the string and has at least two different characters that can be to it's immediate right*). If the extension is right maximal, it adds it to the stack and makes a callback call, It keeps doing this until it has gone through all the right maximal substrings, i.e. the stack is empty.

*This is done by doing a rank operation on a runs vector, counting the number of distinct characters in the interval. See RBWT.

RBWT

Takes in a string and outputs the BWT of the reverse of the string. This enables us to move both directions from anywhere in a string. We can also make a "runs vector" that shows where the letters change in the BWT. With this we can determine if a substring is right maximal.

Uses s_to_BWT.

<u>s_to_BWT</u>
Takes in a string and outputs a wavelet tree representation of the BWT of the string. BWT is what the whole library is built upon, so it's a pretty important function. The resulting BWT is used by almost all of the applications and iterate.

Uses the DBWT library by Kunihiko Sadakane.

<u>substring_stack</u>
A stack that holds substring structs. Used by iterate to iterate.

Applications:

<u>distinct_substrings</u>
Calculates the number of distinct substrings in an input string. This is still a naive version which does not use iterate. This has to be replaced with a version that uses iterate.

<u>draw_tree</u>
Draws a suffix tree based on an input string in a DOT format. The resulted suffix tree is created to a file.

<u>map_bwt_to_s</u>
This contains a helper function for tree drawing. Returns a suffix array which is based on a BWT.

<u>mapper</u>
Maps indexes in BWT to indexes in the original string. It expects to be given a bit vector that corresponds to the BWT and has interesting bits (starting locations of found substrings) marked. The function goes through the BWT and adds a mapping from the BWT to the original string for each marked bit in the bit vector.

With the mapping table, we can map all the found substrings into indexes in the original string.

The functionality differs slightly on different features:

- Maximal repeats can have nested maximal repeats, so you have to go through some parts of the mapping table multiple times. They also can have multiple occurrences in a string, so all of the locations of maximal repeats are listed.
- MUMs are unique and thus mapping them is easy.
- MEMs can have multiple pairs that have the same starting position, so each of these has to be mapped based on one entry in the mapping table. Also, the algorithm that finds MEMs actually finds the index of the character to the immediate left of the MEM, so 1 is incremented to each MEM position.

maximal_repeats
Maximal repeats are substrings that are both left and right maximal.

All the substrings given from single_iterate are right maximal. So to see which are maximal repeats, we only check if they are also left maximal. This is done with a runs vector and the rank operation.

We save the whole interval of the maximal repeat.

mems
MEMs (Maximal Exact Matches) are maximal repeats over 2 strings. So substrings that occur in both and at least in one case the characters to the immediate right and left of the characters are different over the 2 strings.

We find them by extending (with backward search) all the common substrings in both strings first to the left, then to the right and then checking all the intervals made this way so that you check if the first and last character are different over the two different strings. This way we get intervals of the characters to the immediate left of MEMs.

We make "MEM candidates" by extending all the common strings to both directions with all possible letters. Then we check all these and see where the characters to the immediate left and right are different (or end of string signs). If so we found a MEM.

mum
MUMs (Maximal Unique Matches) are substrings that occur only once in both strings.

Iterate gives the function all the common substrings between 2 strings. So for each of these we check that the intervals in both strings are of size 1 and that they have different characters to both directions of the substring.

Bit vectors are made from the completed MUMs list. They should be updated every time we find a MUM, but we did not have time to implement that feature.

triplet_sorter
A general purpose collection of functions that can sort lists of maximal repeats, BWT->string mapping pairs and triplets. The sorting is implemented as a quicksort.

ui
A text-based user interface for testing purposes. This is not actually a part of the library. Notice that the function "draw tree" is not supported yet through ui.

Utils:

bit_vector

A sequence of bits of a certain length. Bits can be marked and read based on index. You can also ask for the length of the bit vector.

Used for making a runs vector that is useful for counting how many distinct characters are in a given interval. Also used with mapping.

print_node

A prototype callback function for iterate. Prints the intervals of a substring.

utils

General purpose function for other files. Swap, quick_sort, etc. Also has printing functions for bit vectors and wavelet trees for debugging purposes.

wavelet_tree

A wavelet tree is a tree where a string can be stored so that it needs very few space. See http://en.wikipedia.org/wiki/Wavelet_Tree for more explanation.