Bioinformatics

The suffix trie and the suffix tree

As you progress within the tutorial, you should develop your intuition on the data structure, as well as gaining automy for good practices in algorithmics (proper definition of problems, running time analysis, high-level description of algorithms) and implementations (extensive code documentation with docstrings and comments, robust tests).

The aim of this tutorial is to implement Python classes for the tries, suffix tries and suffix trees data structures.

(1) ■ Tries

The trie T_S of a string collection $\mathcal{S}$ is a tree such that a string S belongs to $\mathcal{S}$ if and only if it spells a root-to-marked-node path within T_S. It is also such that the paths that correspond to string S and S' coincide in the tree while spelling their longest common prefix.

▶ **Question 1.** *(📓) Recall the algorithm that constructs the trie of a set of string given as input, together with its space and time complexities.*

▶ **Question 2.** *(✎) Use it to build the trie of $\mathcal{S} = \{climate, climb, climbing, close, cloth\}$ by hand.*

Now, open `suffixes_template.py`.

▶ **Question 3.** *(💻) What does the function `Trie.__init__` do?*

▶ **Question 4.** *(💻) Fill in the method `Trie.insert` so that it insert the word `word` into the trie `self`.*

▶ **Question 5.** *(💻) Check your implementation by building the trie of $\mathcal{S} = \{climate, climb, climbing, close, cloth\}$. You can display the resulting trie by using the `pretty_print()` method.*

(2) ■ Suffix tries

Suffix tries are particular tries suited for pattern matching, because a pattern P is a substring of some string S if it is the prefix of one of S's suffixes.

▶ **Question 6.** *(📓) Recall the definition of the suffix trie, highlighting the purpose of the fresh letter \$.*

▶ **Question 7.** *(✎) Build the suffix trie of "MISSISSIPPI".*

▶ **Question 8.** *(💻) Rename a copy of the `Trie` class by `SuffixTrie`, and change the `__init__` method so that it takes a single word as input and build its suffix trie.*

(3) ■ Pattern matching from the suffix tries

Pattern matching is the task of finding occurences of a given pattern within a string. Depending on the output, this define different flavours of pattern matching.

▶ **Question 9.** *(📓) Recall the definition of the Membership, Count and LocateAll problems, being as formal as possible.*

▶ **Question 10.** *(✎) What are the solutions to these three problems on the instance $(ST(\text{"MISSISSIPPI"}), \text{"SI"})$?*

(3.1) ◆ The Membership problem

▶ **Question 11.** *(✎) Give a family of tries $(\mathcal{T}_i)_{i\in\mathbb{N}}$ and a family of pairs of patterns of same length $((P_i, P'_i))_{i\in\mathbb{N}}$ such that the execution of the Membership algorithm seen in class on the inputs $(\mathcal{T}_i, P_i)$ takes $\Theta(|P|)$ time, but only $\Theta(1)$ time on the inputs $(\mathcal{T}_i, P'_i)$.*

▶ **Question 12.** *(💻) Implement a method* `membership` *within the* `SuffixTrie` *class.*

(3.2) ◆ The Count problem

▶ **Question 13.** *(💻) Implement a method* `count_covered_leaves` *for* `Node`*, and use it to implement a method* `count` *for* `SuffixTrie`

▶ **Question 14.** *(💻) Implement a method* `preprocess_leaf_covering` *for* `SuffixTrie` *that stores, at the level of nodes, the number of leaves it covers. You can use the dictionnary* `Node.misc`*, and add a value for the key* `'nb_covered_leaves'`*. This method should run in time* $\mathcal{O}(|\mathcal{T}|)$.

▶ **Question 15.** *(💻) Implement a method* `count_fast` *for* `SuffixTrie`*, that take advantage of the information* `'nb_covered_leaves'`.

▶ **Question 16.** *(✎) Compare the theoretical complexity of the algorithms under* `count` *and* `count_fast`.

▶ **Question 17.** *(💻 **BONUS**) Can you observe this difference in practice? You can use the* `time` *module for timing executions.*

(3.3) ◆ The LocateAll problem

▶ **Question 18.** *(💻) Implement a method* `locate_all` *for* `SuffixTrie`*. This requires to change the* `__init__` *function of* `TrieNode` *to link leaves with their corresponding suffix starting position. You are encouraged to modify the* `pretty_print` *function to help debugging.*

(4) ■ Suffix trees

The suffix trees can be seen as compaction of the suffix tries, whose storage requirements are only linear in the size of the considered string. This compaction also fasten depth-first traversal of the tree, greatly enhancing the complexity of many algorithms compared to their trie version.

▶ **Question 19.** *(✎) Build the suffix tree of "MISSISSIPPI".*

▶ **Question 20.** *(💻) Create a new class* `SuffixTree` *together with its* `__init__` *function.*

Hint. You can structure the `__init__` function as follows: **(1)** create the suffix trie **(2)** traverse it top-down while compacting the branches along the way, and keeping track of the trie depth (eg. storing it in the `misc` dictionnary) **(3)** traverse it bottom-up to propagate the knowlegde of some covered starting position into every nodes.

▶ **Question 21.** *(💻) Implement a* `membership` *method for the* `SuffixTree` *class.*

▶ **Question 22.** *(💻 **BONUS**) Implement methods* `count` *and* `locate_all` *for the* `SuffixTree` *class.*

(5) ■ Challenge: longest common subsequence

One can generalize the suffix tries to collections of sequences as follows: the suffix trie of $\mathcal{C} = (S_1, S_2 \cdots, S_n)$ is defined as the trie of $\text{Suffs}(S_1\$_1) \cup \text{Suffs}(S_2\$_2) \cup \cdots \cup \text{Suffs}(S_n\$_n)$, where $\{\$_1, \$_2, \cdots, \$_n\}$ are distinct symbols that do not appear in $\mathcal{C}$. The suffix trees can be generalized similarly.

▶ **Question 23.** *(🔍 **BONUS**) What is the longest common substring between the genomes of the following widespread viruses:* Hepatitis delta virus *and* Human immunodeficiency virus-1.