AABB

Release

Beatriz Silva

Apr 12, 2023

Chapter 1

AABB

1.1 package_aab.src package

1.1.1 Submodules

1.1.2 package_aab.src.Automata module

class `package_aab.src.Automata.`**`Automata`** (*alphabet: str*, *pattern: str*)

Bases: `object`

Class implemented for the search of patterns in nucleotide sequences. This pattern search method processes a sequence in a linear fashion, reading from left to right. The search is started at the first state of the "machine" and at the first character of the text, at each step of the search the next character of the sequence is considered, the next state is searched state and moves to a new state. The number of states is equal to the length of the pattern plus one. The pattern is found when the pattern being searched for is equal to the length of the pattern entered.

`applyNextState` (*seq: str*) → list

Method that returns a list of all next states. :param seq: sequence entered :return: list of next states

`buildTransitionTable` (*pattern: str*)

Method that constructs the transition table. The transition table returns the next state of the automata machine from the current state and previous states. :param pattern: pattern to search in the sequence

`nextState` (*current: int*, *char: str*) → int

Method that returns the next state. :param current: current state param char: character of the pattern to search for :return: the next state

`patternSeqPosition` (*seq: str*) → list

Method that returns the list of positions where an occurrence of the pattern in the sequence starts. param seq: sequence entered :return: list of the positions where an occurrence of the pattern in the sequence starts.

`printAutomata` ( )

Method responsible for printing the results of the pattern search using the automata.

`package_aab.src.Automata.`**`overlap`** (*seq1: str*, *seq2: str*) → int

Method that overlaps two sequences and checks for matching :param seq1: first string :param seq2: second string :return: last position of the smallest string that matches

1.1.3 package_aab.src.BTW module

class `package_aab.src.BTW.`**`BWT`** (*seq='', buildsufarray=False, sa=None*)

Bases: `object`

Class for implementing the Burrows-Hweeler algorithm where different methods have been created. The algorithm is useful for compressing large sequences, thus reducing their space. Moreover, it is possible to find patterns in the compression format efficiently.

`build_bwt` (*text, buildsufarray=False*) → str

Method to build the matrix for Burrows-Wheeler transformation. param text: sequence we want to use param buildsufarray: parameter to create a suffix array. Default: False.

`bw_matching` (*pattern=<class 'str'>*) → List[int]

Method to look for patterns from the Burrows-Wheeler transform. :param pattern: pattern we want to find :return: list with matches

`bw_matching_pos` (*patt*) → List

Method that finds the matches of a pattern :param patt: pattern we want to find :return: list of matches found

`get_first_col` ( ) → List[str]

Method to retrieve the first column. note that the first column is the alphabetical ordering of the transform :return: list of the first column

`inverse_bwt` ( ) → str

Method to get the original sequence :return: string of the original sequence

`last_to_first` ( ) → List

Method to create the table with the last column and the first column :return: list of indices of the transform

`set_bwt` (*bw*)

`package_aab.src.BTW.`**`findithocc`** (*le, elem, index*)

Method to find out the position of the i-th occurrence of a symbol in a list (returns -1 if it doesn't occur). param le: array to look for param elem: element to search for :param index: occurrence to look for

1.1.4 package_aab.src.BoyerMoore module

class `package_aab.src.BoyerMoore.`**`BoyerMoore`** (*alphabet=<class 'str'>, pattern=<class 'str'>*)

Bases: `object`

`process_bcr` ( )

Bad Character rule implementation. Method where a dictionary is created with all possible symbols (occ) as keys, and the values define the rightmost position at which the symbol appears in the pattern (-1 means it does not occur). This allows you to quickly calculate the number of positions to follow to search according to the mismatch in the pattern (value for the symbol in the dictionary). Note that this value can be negative, meaning that the rule in this case is not useful and is ignored in the next iteration.

`process_gsr` ( )

Good Suffix rule implementation. Calculates the value of the f list (the length of the longest proper suffix that matches the suffix of the pattern), and s list (the length of the longest proper suffix that matches the prefix of the pattern).

`search_pattern` (*text=<class 'str'>*) → List[int]

This method allows to find a pattern in a given text, based on the object of the class that contains the pattern and its alphabet. :param text: string of the text where we want to look for our pattern :return: list with the indexes where the pattern starts

1.1.5 package_aab.src.EAMotifs module

1.1.6 package_aab.src.EvolAlgorithm module

class `package_aab.src.EvolAlgorithm.`**`EvolAlgorithm`** (*popsize: int*, *numits: int*, *noffspring: int*, *indsize: int*)

Bases: `object`

`evaluate` (*indivs: list*) → None

Method that calculates the score for each individual, setting its fitness. :param indivs: list that represents the individuals solution. :return:

`initPopul` (*indsize: int*) → None

Method that initializes the population with a given size for individuals. :param indsize: size of individuals (list of genes) :return:

`iteration` ( ) → None

Method auxiliary of the Evolutionary Algorithm cycle (based on the number of iterations wanted). :return:

`printBestSolution` ( )

`run` ( ) → None

Method that runs the evolutionary algorithm cycle until finding the best solution or the number of iterations is reached. :return:

1.1.7 package_aab.src.Indiv module

class `package_aab.src.Indiv.`**`Indiv`** (*size*, *genes=[]*, *lb=0*, *ub=1*)

Bases: `object`

# The Indiv class represents an individual in a genetic algorithm. It contains methods for initialization, mutation, crossover, and fitness evaluation.

`crossover` (*indiv2*)

`getFitness` ( )

`getGenes` ( )

`initRandom` (*size*)

`mutation` ( )

`one_pt_crossover` (*indiv2*)

`setFitness` (*fit*)

class `package_aab.src.Indiv.`**`IndivInt`** (*size: int*, *genes: list = []*, *lb: int = 0*, *ub: int = 1*)

Bases: `Indiv`

`initRandom` (*size: int*) → None

Method that generates a list of genes of the individual (random int numbers between upper and lower bounds) :param size: number of genes to generate :return:

`mutation` ( )

class `package_aab.src.Indiv.`**`IndivReal`** (*size*, *genes=[]*, *lb=0*, *ub=1*)

Bases: `Indiv`

`initRandom` (*size: int*) → None

`mutation` ( ) → None

`package_aab.src.Indiv.`**`random`** ( ) → x in the interval [0, 1).

1.1.8 package_aab.src.MotifFinding module

Class: MotifFinding

class `package_aab.src.MotifFinding.`**`MotifFinding`** (*size: int = 8*, *seqs=None*)

Bases: `object`

Class implemented for searching for recurrent patterns in a biological sequence, which can be DNA or proteins. The pattern to look for can be an exact sequence or a degenerate consensus in which there are ambiguous characters.

`branchAndBound` ( ) → Optional[Union[list, List[int]]]

Method that implements to iterate over the initial position of the list of solutions to find the best solution motif. :return: the vector of starting positions s (representative of the best motif found)

`bypass` (*s: list*) → list

Method implemented to bypass the condition explained above. :param s: index list of the motifs in the entered sequences. :return: list of indexes with zeros in the non-corresponding positions.

`createMotifFromIndexes` (*indexes: list*)

Method that implements probabilistic motifs of type MyMotif. :param indexes: list of the indexes of the initial positions of the motif of the sub-sequences used to create the motif. :return:

`exhaustiveSearch` ( ) → list

Method implemented to find the motif with the vector of best scores. This method allows to derive the profile and the consensus sequence. :return: vector of best scores

`gibbs` (*num_iterations: int*) → list

Method that implements the Gibbs Sampling algorithm, by choosing new segments at random (increasing the possibilities of converging to a correct solution). The algorithm starts with a set of randomly selected motif positions, and then in each iteration, it chooses one sequence at random, removes it from consideration, and recomputes the motif position based on the remaining sequences. This process continues for a fixed number of iterations, after which the algorithm returns the best set of motif positions found so far. :param num_iterations: number of iterations :return: list of best scores

`heuristicConsensus` ( ) → list

‘’’Method that computes the heuristic consensus algorithm: - Considering only the first two sequences, choose the initial positions s1 and s2 that give a better score. - For each of the following sequences, iteratively, choose the best starting position in the sequence, in order

to maximize the score.

Returns list of starting positions s (representative of the best motif found)

`heuristicStochastic` ( ) → list

Method that computes the heuristic stochastic consensus algorithm, using the most likely segments to adjust starting positions to achieve the best profile (motif) :return: the vector of starting positions s (representative of the best motif found)

nextSol (*s: list*) → list
Auxiliary method (to exhaustiveSearch) that gives the next vector of starting positions. Method implemented to iterate over all the possible values of the motif position in the entered sequence. :param s: list of the motif indexes in the entered sequence. :return: list of all the possible values of the position of the motif in the entered sequence.

nextVertex (*s: list*) → list
Method implemented to find the next vertex. :param s: list of indices of the motifs in the entered sequences. :return: list of next vertices

readFile (*file: str*, *type: str*) → None
Method that reads the sequence file. param file: sequence file. param type: type of the sequences.

roulette (*f: list*) → int
Method implemented to simulate examples of a roulette wheel. :param f: list of positions :return: chosen value

score (*s: list*) → int
Method implemented for the scoring function. The scoring function iterates over all motif positions and determines the maximum value of the motif score. :param s: list of motif indices in the entered sequences. :return: maximum score of the motif score.

scoreMult (*s*)
Method implemented for the multiplication of the maximum motif scores. The scoreMult() method provides a way to quantify the importance of each occurrence of a motif by calculating the maximum score for each position in the motif, and then multiplying these scores together to get an overall score. :param s: list of the motif indexes in the introduced sequences. :return: multiplication of maximum scores.

seqSize (*i: int*) → int
Method that returns the length of the sequence. :param i: index of the sequence in the sequence list. :return: length of the sequence with index i from the sequence list.

`package_aab.src.MotifFinding.`**random** ( ) → x in the interval [0, 1).

1.1.9 package_aab.src.MyMotifs module

Class: MyMotifs

class `package_aab.src.MyMotifs.`**MyMotifs** (*lseqs: list = [], pwm: list = [], alphabet: Optional[str] = None*)
Bases: `object`
Class that presents the methods that allow the manipulation and search of recurring patterns (motifis) in biology sequences and do PWM.

consensus ( ) → str
Method that generates a consensus sequence. Consensus sequences store the most conserved characters in each position of the pattern, that is, the highest value of each column of the count matrix. :return: string of the consensus sequence.

createPWM ( ) → None
Method that creates the probabilistic matrix. PWMs are probabilistic representations of the characters in biological sequences. It calculates the probability of nucleotide i being found at position j.

`doCounts` ( )
Method that implements the counting matrices.

`maskedConsensus` ( ) → str
Method that generates the consensus sequence that is obtained with the characters that have an incidence higher than 50%. :return: string of the consensus sequence with incidence higher than 50%.

`mostProbableSeq` (*seq: str*) → int
Method implemented to determine the sub-sequence with the highest probability of matching the pattern being searched. :param seq: sequence introduced return: the index of the sub-sequence with higher probability of matching the pattern in search.

`probAllPositions` (*seq: str*) → list
Method implemented to calculate the probability of finding patterns in longer sequences and calculate the probability of the pattern occurring at each character in the sequence, ie, of occurring at each sub-sequence of the pattern size. For example, if the pattern size is 3 and the sequence is "ATCGTACG", the method would calculate the probability of the pattern occurring at each of the sub-sequences of size 3, which are "ATC", "TCG", "CGT", "GTA", "TAC", "ACG". :param seq: sequence introduced :return: list of probabilities of the pattern occurring at each subsequence

`probabSeq` (*seq: str*) → float
Method that calculates the probability of a pattern being found in a sequence. :param seq: sequence intoduced :return: the probability of a pattern being found in the sequence

`package_aab.src.MyMotifs.`**`createMatZeros`** (*line_num: int*, *col_num: int*) → list
Method that creates the matrix of zeros. The matrix of zeros is used as a base to store the counts or probabilities of each character in each position of the biological sequences that are being analyzed. :param line_num: number of lines in the matrix :param col_num: number of columns in the matrix :return: matrix of zeros

`package_aab.src.MyMotifs.`**`printMat`** (*mat: list*)
Method to print the matrix :param mat: matrix to print

1.1.10 package_aab.src.MySeq module

Class:MySeq

class `package_aab.src.MySeq.`**`MySeq`** (*seq: str*, *seq_type: str*)
Bases: `object`
Class presenting the methods that allow the manipulation of DNA, RNA and protein sequences.

`allProtein` ( )
Method that constructs of a list with all the existing and possible proteins in the sequence. The sequence can be proteic or DNA Only executes in case the object is a DNA or AMINO sequence :return: DNA or amino sequence

`alphabet` ( )
Method that verifies the type of sequence :return: returns the possible caracters of the sequence.

`codonTranslate` (*cod: str*) → str
Método que traduz os codões nos respetivos aminoácidos. :param cod: codão a procurar na tabela de codões :return: sequência de aminoácidos

`largestOrfProtein` ( )
Method to get the bigger protein in the sequence :return: string with the bigger protein

longestProteinSeq ( )
: Método que procura a sequência proteíca mais comprida. :return: maior sequência proteíca.

orfs ( )
: Método que determina as open reading frames (ORF), i.e, as sequências compreendidas entre o codão de iniciação e o codão STOP. Gera seis reading frames da sequência de DNA e do complemento inverso. :return: devolve as ORF's

printSeq ( )
: Method that prints the complete sequence :return: string of the complete sequence

reverseComplement ( )
: Method that inverts and complements the sequence Only executes in case the object is a DNA sequence :return: DNA sequence

rnaCodon ( ) → list
: Method that searches for the codons in the sequence. :returns: list of codons

seqTranslation (*initial_pos: int = 0*)
: Method that performs the translation of the sequence :param initial_pos: determines the initial position of the reading sequence :return:

transcription ( )
: Method that returns the transcript sequence. Replacing of the thymine base 'T' for the uracil base 'U' Only executes in case the object is a DNA sequence :return: RNA sequence

validateSeqRE ( ) → bool
: Method that validates the sequence according to the type of characters present through regular expressions. :return: value True or False if the sequences are valid or invalid, respectively

1.1.11 package_aab.src.Popul module

class `package_aab.src.Popul.`**Popul** (*popsize, indsize, indivs=[]*)
: Bases: `object`

bestFitness ( )

bestSolution ( )

getFitnesses (*indivs=None*)

getIndiv (*index*)

initRandomPop ( )

linscaling (*fitnesses*)

recombination (*parents, noffspring*)

reinsertion (*offspring*)

roulette (*f*)

selection (*n, indivs=None*)

class `package_aab.src.Popul.`**PopulInt** (*popsize, indsize, ub, indivs=[]*)
: Bases: `Popul`

initRandomPop ( )

class package_aab.src.Popul.**PopulReal** (*popsize: int*, *indsize: int*, *lb: float = 0.0*, *ub: float = 1.0*, *indivs: list = []*)

Bases: Popul

initRandomPop ( ) → None

Method that initializes the population (creates instances of IndivReal class) :return:

package_aab.src.Popul.**random** ( ) → x in the interval [0, 1).

1.1.12 package_aab.src.SuffixTree module

Class: SuffixTree - Trie of sufixes

class package_aab.src.SuffixTree.**SuffixTree**

Bases: object

Class that creates a suffix tree of a pattern that will be searched in a sequence. It allows pre-processing a target sequence, making its search more efficient. It is the solution to pre-process very large sequences, discovering which trees contain a given pattern, finding out the longest common substring in a set of sequences and calculating the maximum overlap of a set of sequences.

addNode (*origin: int*, *symbol: str*, *leafnum=- 1*) → None

Method that adds nodes to the tree. :param origin: current node :param symbol: character to be added to the node :param leafnum: leaf number (-1 is default)

addSuffix (*p: list*, *sufnum: int*) → None

Method that adds suffix to the tree. :param p: pattern :param sufnum: suffix number for leaves or -1 for non-leaves

findPattern (*pattern: str*)

Method that searches for patterns (trie) starting from the root until reaching the final node or failing the search. :param pattern: pattern to search for :return: list of leaves below a given node or None (if search fails)

getLeavesBelow (*node: int*) → list

Auxiliary method for collecting all leaves below a given node. :param node: node from which to search for leaf information below it :return: list of leaves below a given node

printTree ( ) → None

Method that prints the suffix tree.

suffixTreeFromSeq (*text: str*) → None

Method that creates the suffix tree, adding a symbol (“$”) at the end of the sequence and calling the previous method for each suffix of the sequence -> no suffix will be the prefix of another suffix. :param text: sequence that will be added to the tree

1.1.13 package_aab.src.Trie module

Class: Trie - Trie of prefixes

class package_aab.src.Trie.**Trie**

Bases: object

Class responsible for implementing the prefix tree that allows pre-processing of a set of patterns. The symbols of a given alphabet are associated with the arcs of a tree. The trie is built from a set of patterns, starting from the root node and iterating each pattern, adding the necessary nodes so that the tree contains the path from the root to the leaf, representing the pattern.

addNode (*origin: int*, *symbol: str*) → None
Method that adds a node to the trie. This method is used by the add_pattern method. :param origin: existing node :param symbol: character referring to the node to be added (arc identification)

addPattern (*p: list*) → None
Method that adds a pattern to the trie :param p: input pattern

prefixTrieMatch (*text: str*)
Method to search if a trie pattern is a sequence prefix. Scrolls through the sequence of characters and the tree, starting at the root, following the arcs corresponding to the characters of the sequence until one of the following occurs: - If it reaches a leaf of the trie, the pattern has been identified; - The character of the sequence does not exist -> no pattern was identified in the trie. :param text: sequence of characters :return: prefix match or None

printTrie ( )
Method that print the trie

trieFromPatterns (*pats: list*) → None
Method that adds each pattern of the input to the trie. :param pats: input patterns

trieMatches (*text: str*) → list
Method to, using the prefixTrieMatch method, look for occurrences (matches) of the pattern in the sequence. Iterative process - matches the sequence, removes the first symbol from the sequence, repeats the process is repeated for all suffixes in the sequence. :param text: sequence :return: list of matches

1.1.14 Module contents

- genindex
- modindex
- search

Python Module Index