Escola de Engenharia – Departamento de Informática da Universidade do Minho
Mestrado em Bioinformática
Algoritmos Avançados de Bioinformática 2021/2022

Algorithm Package

Tutorial

Alexandre Esperança pg45963, Cristiana Albuquerque pg45468, Mónica Leiras pg45473, Tomás Sá pg45477

Links:

GitHub: https://github.com/alexesperanca/AlgAvancados2k22

To access Doxygen output go to Doxygen_Results -> html -> index

Table of Contents

1 Introduction

This Algorithm package constitutes a set of classes and functions for biological sequences manipulation and analysis available for Python.

This package has some functionalities such as:

- A set of classes that deals with sequences, namely DNA, RNA and amino acid sequences;
- Code for dealing with sequence processing, patterns and motifs, including Burrows-Wheeler transformation, Boyer-Moore algorithm and pattern organization in tries, and evolutionary algorithms for motifs extraction;
- Code for dealing with graphs and genome sequencing;
- Documentation and help with using the classes, including this file.

2 Classes and usage

This package contains the following classes:

Searching for patterns in sequences:

- Class Automata;
- Class BoyerMoore;
- Class Trie.

Burrows-Wheeler transformations:

- Class BWT.

Motif search and evolutionary computing:

- Class Sequence;
- Class Indiv;
- Class Popul;
- Class Motifs;
- Class MotifFinding;
- Class EvolAlgorithm;
- Class EAMotifs.

Graphs:

- Class MyGraph;
- Class OverlapGraph.

Biological networks:

- Class MetabolicNetwork.

Graphs and genome sequencing:

- Class DeBruijnGraph.

The features of each class will be addressed in the next sections, along with some usage examples.

2.1. Searching for patterns in sequences

2.1.1 Class Automata

The **class Automata** aims to facilitate the encounter of patterns in sequences,and includes:

- Transition tables that indicates the states transition along the pattern;
- The next state for a given current state and character;
- Occurrences of the patter;

- The size of the overlap between two sequences.

The parameters of the **class Automata** are:

- alphabet: characters present in the pattern;
- pattern: pattern for the transition table construction;
- current: current state;
- symbol: character of the alphabet;
- seq: sequence to apply and verify the state alteration;
- text: model to cross the pattern and identify the occurrences;
- s1: first sequence;
- s2: second sequence

2.1.1.1 Usage example

An example of the *applySeq* function (that returns the states along the given sequence) of the **class Automata** is:

```
print (auto.applySeq("CACAACAA"))
```

An output example of the function is:

```
States:  4
Alphabet:  AC
Transition table:
0 , A  ->  1
0 , C  ->  0
1 , A  ->  1
1 , C  ->  2
2 , A  ->  3
2 , C  ->  0
3 , A  ->  1
3 , C  ->  2
```

2.1.2 Class BoyerMoore

The **class BoyerMoore** aims to apply different methods to facilitate pattern matching in a text or sequence. These methods are:

- Bad Character Rule.
- Good Suffix Rule.

The parameters of the **class BoyerMoore** are:

- alphabet: possible characters present in the pattern;
- pattern: string to search for matches;
- text: text to find the place where the pattern matches.

2.1.2.1 Usage example

An example of the *search_pattern* function (that searches the perfect match of the pattern in the given text by executing the Bad Character Rule and Good Suffix Rule) of the **class BoyerMoore** is:

```
padrao = "ACCA"
```

```
texto = "ATAGAACCAATGAACCATGATGAACCATGGATACCCAACCACC"
bm = BoyerMoore("ACTG", padrao)
print (bm.search_pattern(texto))
```

An output example of the function is:

```
[5, 13, 23, 37]
```

2.1.3 Class Trie

The **class Trie** allows the construction of a wide tree with several ramifications enabling improved analysis. The subclass SuffixTrie allows the user to construct a wide tree of a given initial sequence.

The parameters of the class Trie are:

- seq_list: sequences to be incremented in the tree;
- seq: sequence to be incremented in the tree;
- pat: pattern to identify in the tree;
- seq: sequence to run against the tree.

The parameters of the subclass SuffixTree are:

- seq: sequence model for the several sequences to tree construction;
- p: sequence to add to the tree;
- pat: sequence reference to cross with the tree;
- node: character to iterate over the tree;
- pat: sequence to cross from the tree.

2.1.3.1 Usage example

An example of the *trie_matches* function (that print the matches and positions of the matches obtained from the *match* function) of the **class Trie** is:

```
test = Trie(["CTG", "CATA", "CAAGG"])
test.trie_matches("CTGCATACAAGG")
```

An output example of the function is:

```
CTGCATACAAGG contains CTG in position 0
CTGCATACAAGG contains CATA in position 3
CTGCATACAAGG contains CAAGG in position 7
```

An example of the *find_pattern* function (that encounters the tree patterns and gives the corresponding positions in the provided sequence) of the **subclass SuffixTree** is:

```
seq = "TACTA"
st = SuffixTree(seq)
st.print_tree()
```

An output example of the function is:

```
[('TA', 0), ('A', 1), ('TA', 2), ('A', 3)]
```

2.2 Burrows-Wheeler transformations

2.2.1 Class BWT

The **class BWT** aims to analyze big sequences and pattern discovery through a Burrows-Wheeler transformation. The parameters are:

- seq: sequence to be the model for Burrows-Wheeler matrix;
- pat: pattern to match the matrix.

2.2.1.1 Usage example

An example of the *find_pattern* (that finds a pattern in the BWT matrix built) and *suffixarray* (that retrieves a list with the initial positions of each ordered suffix) functions of the **class BWT** is:

```
seq = "TAGACAGAGA$"
test = BWT(seq)
print(test.find_pattern("AGA"))
print(test.suffixarray())
```

An output example of the functions is:

```
[3, 4, 5]
(10, 9, 3, 7, 1, 5, 4, 8, 2, 6, 0)
```

2.3 Motif search and evolutionary computing

This section is represented by 7 classes that may interact with each other. The classes and respective parameters are:

2.3.1 Class Sequence

The **class Sequence** aims to verify the type of sequences (DNA, RNA or protein) and includes functions to determine mRNA transcripts, complement inverse, open reading frames and translated proteins and others. For that, the parameters are:

- seq : sequence inputted by the user, that can be DNA, RNA or amino acid.

2.3.2 Class Indiv

The **class Indiv** implements individuals with binary (IndivInt subclass) or real representations (IndivReal subclass). The parameters of the **class Indiv** are:

- size: size of the list of genes;
- genes: list of genes representative of the genome;
- lb: lower bound of the range for representing genes;
- ub: upper bound of the range for representing genes;
- solution: individual;
- fit: fitness of the individual;;
- size: number of genes to generate;
- indiv2: individual.

The parameters of the **subclasses IndivInt** and **IndivReal** are:

- size: size of the list of genes;
- genes: list of genes representative of the genome;
- lb: lower bound of the range for representing genes;
- ub: upper bound of the range for representing genes;
- size: number of genes to generate.

2.3.3 Class Popul

The **class Popul** includes a binary (**subclass PopulInt**) and real (**subclass PopulReal**) representations for creating populations of individuals. The parameters of the **class Popul** are:

- popsize: size of population;
- indsize: size of individuals;
- indivs: list of genes;
- index: index of the individual on the list of genes;
- indivs: list of individuals;
- n: size of selection;
- f: list of fitnesses;
- fitnesses: list of fitnesses;
- parents: list of individuals (parents);
- noffspring: number of offspring individuals;
- offspring: list of individuals (offspring).

The parameters of the **subclass PopulInt** are:

- popsize: size of population;
- indsize: size of individuals;
- ub: upper limits of the range for representing genes;
- indivs: list of genes.

The parameters of the **subclass PopulReal** are:

- popsize: size of population;
- indsize: size of individuals;
- lb: lower limits of the range for representing genes;
- ub: upper limits of the range for representing genes;
- indivs: list of genes.

2.3.4 Class Motifs

The **class Motifs** allows the creation of probabilistic profiles given a list of sequences. The parameters are:

- list_seq: sequences of DNA, RNA or amino acids;
- profile: list with a dictionary of the probabilistic profile (PWM or PSSM);
- s1: string sequence of DNA or amino acids;
- s2: string sequence of DNA or amino acids.

2.3.4.1 Usage example

An example of the *seq_most_probable* function (that returns the subsequence with the highest probability according to the associated profile) of the **class Motifs** is:

```
seq1 = "aGgtacTt".upper()
seq2 = "CcAtacgt".upper()
seq3 = "acgtTAgt".upper()
```

```
seq4 = "acgtCcAt".upper()
seq5 = "CcgtacgG".upper()
lseqs = [seq1, seq2, seq3, seq4, seq5]
motifs = Motifs(lseqs)
motifs.print_profile()
print(motifs.seq_most_probable("CTATAAACCTTACATC"))
```

An output example of the function is:

```
('CTATAAAC', 0)
```

2.3.5 Class MotifFinding

The **class MotifFinding** allows the search of conserved motifs given a list of sequences, including strategies as:

- Exhaustive search type;
- Branch and Bound (search trees);
- Consensus (heuristic and estochastic algorithms);
- Gibbs sampling.

The parameters are:

- size: size of motif to find;
- seqs: list of sequences;
- n: number of the index;
- i: number of the index;
- fic: name of the file;
- indexes: list of indexes of the sequences;
- pseudocount: pseudocount of the profile;
- s: list of indexes;
- s: current vector of starting positions s;
- iterations: maximum number of iterations;
- f: list of probabilities.

2.3.5.1 Usage example

An example of the exhaustive search of the **class MotifFinding** is:

```
mf = MotifFinding(seqs = ["GTAAACAATATTTATAGC", "AAAATTTACCTCGCAAGG",
"CCGTACTGTCAAGCGTGG", "TGAGTAAACGACGTCCCA", "TACTTAACACCCTGTCAA"])
sol = mf.exhaustiveSearch()
print ("Solution", sol)
print ("Score: ", mf.score(sol))
print("Consensus:", mf.createMotifFromIndexes(sol).consensus())
```

An output example of the function is:

```
Solution [0, 4, 0, 0, 0]
Score:  18
Consensus: ACGT
```

2.3.6 Class EvolAlgorithm

The **class EvolAlgorithm** is a genetic and evolutionary algorithm that works with populations, where each individual represents a solution to a given problem. The parameters are:

- popsize: size of population;
- numits: number of iterations to perform;
- noffspring: number of new individuals (descendants);
- indsize: size of individuals (list of genes);
- indivs: list that represents the individuals solution.

2.3.6.1 Usage example

An example of the *run* and *printBestSolution* functions (that runs the evolutionary algorithm cycle until finding the best solution or the number of iterations is reached) of the **class MotifFinding** is:

```
ea = EvolAlgorithm(100, 20, 50, 10)
ea.run()
ea.printBestSolution()
```

An output example of the function is:

```
Best solution:  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Best fitness: 10.0
```

2.3.7 Module EAMotifs

The module **EAMotifs** is a part of a genetic and evolutionary algorithm. It provides two subclasses of the **class EvolAlgorithm**: the **subclass EAMotifsInt** is a binary representation while the **subclass EAMotifsReal** is a real representation of the evolutionary algorithm for finding motifs. The parameters of the subclass EAMotifsInt are:

- popsize: size of population;
- numits: number of iterations to perform;
- noffspring: number of new individuals;
- filename: name of the file to read the sequences;
- indsize: size of individuals;
- indivs: list that represents the individuals solution.

The parameters of the **subclass EAMotifsReal** are:

- popsize: size of population;
- numits: number of iterations to perform;
- noffspring: number of new individuals;
- filename: name of the file to read the sequences;
- indsize: size of individuals;
- pwm_list: list with values;
- seq: sequence of DNA, RNA or amino acids;
- profile: list with a dictionary of the probabilistic profile (PWM or PSSM);
- indivs: list that represents the individuals.

2.3.7.1 Usage example

An example of the *printBestSolution* function (that runs the evolutionary algorithm cycle until finding the best solution or the number of iterations is reached) of the **class EvolAlgorithm** using the **module EAMotifs** is:

```
ea = EAMotifsInt(100, 1000, 50, "exemploMotifs.txt")
ea.run()
ea.printBestSolution()

ea = EAMotifsReal(100, 2000, 50, "exemploMotifs.txt")
ea.run()
ea.printBestSolution()
```

An output example of the function is:

```
Best solution:  [1, 1, 38, 5, 0]
Best fitness: 26

Best solution:  [8.516736066370482, 45.09360628965298,
18.76870217784353, 1.8943195756811626, 44.755318800730294,
57.01685444072013, 30.058588338437982, 57.57018724789731,
54.953566477848234, 4.4991871772483645, 51.06728076392951,
35.70291467955955, 54.79780268609979, 36.96251035963968,
46.41430272653992, 26.973657742880054, 26.695411467911214,
22.413997024635908, 32.78657491199706, 39.69118829409969]
Best fitness: 34
```

2.4 Graphs

2.4.1 Class MyGraph

The class MyGraph allows the construction of graphs through a set of strategies:

- Incremention of nodes and edges construction;
- Obtain successors, predecessors and adjacent nodes;
- Degrees calculation;
- Reachables nodes obtainment;
- Path discovery between different locations (nodes);
- Calculation of clustering coefficients;
- Hamiltonian path discovery;
- Eulerian path discovery.

The parameters are:

- g: dictionary given to represent the graph;
- v: element to add in the graph;
- o: element to add in the beggining of the graph and edge with “d”;
- d: element to edge with “o”;
- v: element from which successors are returned;
- v: element from which predecessors are returned;
- v: element from which adjacents are obtained;
- v: node to obtain number of degree;

- deg_type: informs which type of degree to compute and return;
- v: node to search for the reachables ones;
- graph: path to analyze;
- s: node where the path begins;
- d: node where the path ends;
- s: node to get the reachable ones;
- v: node from which it is checked if a cycle is detected;
- v: node to calculate the clustering coefficient;
- p: path to take in the graph;
- p: path given to verify if is an hamiltonian path;
- v: current node to obtain the possibilities;
- path: current path built;
- visited: previously visited nodes;
- start: node to start the hamilton path;
- tl: list/tuple to check if element "val" exists;
- val: element to check if present in "tl".

The parameters of the **subclass OverlapGraph**, that includes diverse strategies that uses Hamiltonian methods for faster sequence analysis, sequence obtainment from graphs, and suffix/prefix iteration, are:

- MyGraph: essential class to construct the graph and access to the hamiltonian methods;
- frags: list of elements to create the graph
- reps: if True the graph will add to each element an identifier to avoid repetitions, otherwise they will not be considered;
- frags: list of elements to add to the graph;
- frags: list of elements to add to the graph;
- seq: sequence to encounter the occurrences of this in the graph;
- node: node to obtain the sequence;
- path: list elements of a path;
- k: size of each element to retrieve
- seq: original sequence.

2.4.1.1 Usage example

An example of the *all_clustering_coefs* (that collects all clustering coefficients of the nodes in the graph) function of the **class MyGraph** is:

```
gr = MyGraph({1:{2: None}, 2:{3: None}, 3:{2: None, 4: None},
4:{2: None}})
print(gr.all_clustering_coefs())
```

An output example of the functions is:

```
{1: 0.0, 2: 0.3333333333333333, 3: 1.0, 4: 1.0}
```

An example of the *check_if_valid_path*, *check_if_hamiltonian_path* and *seq_from_path* functions is:

```
ovgr = OverlapGraph([["ATA",  "ACC", "ATG", "ATT", "CAT", "CAT",
"CAT", "CCA" , "GCA", "GGC", "TAA", "TCA", "TGG", "TTC", "TTT"],
True])
frags = ['ACC-2', 'CCA-8', 'CAT-5', 'ATG-3', 'TGG-13', 'GGC-10',
'GCA-9', 'CAT-6', 'ATT-4', 'TTT-15', 'TTC-14', 'TCA-12', 'CAT-7',
'ATA-1', 'TAA-11']

print(ovgr.check_if_valid_path(frags))
print(ovgr.check_if_hamiltonian_path(frags))
print(ovgr.seq_from_path(frags))
print(ovgr.search_hamiltonian_path())
```

An output example of the function is:

```
True
True
“ACCATGGCATTTCATAA”
['ACC-2', 'CCA-8', 'CAT-5', 'ATG-3', 'TGG-13', 'GGC-10', 'GCA-9',
'CAT-6', 'ATT-4', 'TTT-15', 'TTC-14', 'TCA-12', 'CAT-7', 'ATA-1',
'TAA-11']
```

2.5. Biological Networks

2.5.1 Class MetabolicNetwork

The **class MetabolicNetwork** aims to construct graphs based on metabolites and reactions and information retained. The parameters are:

- MyGraph: parent class of MetabolicNetwork;
- network_type: informs the type of graph network the algorithm is working with;
- split_rev: indicates if the reactions are considered reversible or not;
- v: vertex to be added;
- nodetype: vertex type (metabolic or reaction);
- filename: file name with the content desired to be loaded to a graph;
- gmr: current graph;
- s: node path starts;
- t: node path ends;
- init: list of reaction/metabolites initially active/present;
- f: indicates if the algorithm should filter reactions or metabolites;
- met: list of metabolites;
- reac: list of reactions;
- init_met: list of metabolites.

2.5.1.1 Usage example

An example of the *get_nodes_type* (that obtain the nodes of a given node type) and *final_met* functions (that obtains the possibly produced metabolites for a given list of initial metabolites) of the **class Metabolic Network** is:

```
mrn = MetabolicNetwork("metabolite-reaction")
mrn.load_from_file("example-net.txt")
mrn.print_graph()
print("Reactions: ", mrn.get_nodes_type("reaction"))
print("Metabolites: ", mrn.get_nodes_type("metabolite"))
print(mrn.final_met(["M5", "M2"]))
```

An output example of the function is:

```
R1  ->  {'M3': None, 'M4': None}
M1  ->  {'R1': None}
M2  ->  {'R1': None}
M3  ->  {}
M4  ->  {'R2': None, 'R3': None}
R2  ->  {'M3': None}
M6  ->  {'R2': None, 'R3': None}
R3  ->  {'M4': None, 'M5': None, 'M6': None}
M5  ->  {'R3': None}

Reactions:  ['R1', 'R2', 'R3']

Metabolites:  ['M1', 'M2', 'M3', 'M4', 'M5', 'M6']

['M4', 'M5', 'M6', 'M3']
```

2.6 Graphs and genome sequencing

2.6.1 Class DeBruijnGraph

The **class DebruijnGraph** includes diverse strategies for deBrujin graph construction to more easily obtain and deconstruct sequences. The parameters are:

- MyGraph: essential class to construct the graph and access to the eulerian methods;
- o: parent node of "d";
- d: child node of "o";
- v: node to calculate the entry degree;
- frags: list of fragments of sequence to add to the graph;
- path: eulerian path;
- seq: sequence to obtain the suffix;
- seq: sequence to obtain the prefix;
- k: size of the sequence fragments;
- seq: original sequence.

2.6.1.1 Usage example

An example of the *check_nearly_balanced_graph*, *eulerian_path* and *seq_from_path* functions is:

```
frags = ['AATG', 'ATGC', 'ATGG', 'CAAT', 'GCAA', 'GGTC', 'GTCT',
'TCTG', 'TGCA', 'TGGT']
dbgr = DeBruijnGraph(frags)

print(dbgr.check_nearly_balanced_graph())
print(dbgr.eulerian_path())
print(dbgr.seq_from_path(['ATG', 'TGC', 'GCA', 'CAA', 'AAT', 'ATG',
'TGG', 'GGT', 'GTC', 'TCT', 'CTG']))
```

An output example of the function is:

```
(‘ATG’, ‘CTG’)
[‘ATG’, ‘TGC’, ‘GCA’, ‘CAA’, ‘AAT’, ‘ATG’, ‘TGG’, ‘GGT’, ‘GTC’, ‘TCT’,
‘CTG’]
‘ATGCAATGGTCTG’
```