Optimal Pairwise Alignment

September 8, 2014

Contents

1 Introduction

The nascent field of bioinformatics is expanding at unprecedented rate, demanding more and more efficient implementations of a variety of algorithms, especially associated with sequence alignment.

In many cases hundreds of billions of alignments have to be done to investigate relations between publicly available genes. Such an immense amount of operations require an efficient low level - possibly vectorised - implementation, for which usually C/C++ is used. However, it is usually not really convenient to develop the applications that use the alignment algorithm in these low level languages.

PyOPA makes it possible to carry out various operations on sequences with the efficiency of a vectorised C and the convenience of a Python code, in a platform independent way.

The C core, that is wrapped by using NumPy and Cython, is already broadly used from Darwin, so in order to make a platform-switching possible, all results must conform to the results coming from Darwin, which is assured by the unit tests coming with PyOPA.

In this document, we concisely describe the main functionalities offered by PyOPA as well as a brief comparison to PyCogent. In Section 2 we show how to install PyOPA, while Section 3 outlines the main features of the package, followed by the comparison to PyCogent in Section 4. Finally, a short conclusion can be found in Section 5.

2 Installing PyOPA

PyOPA draws on features offered by *NumPy* and *Cython* and it is strongly recommended to properly install the *latest* version of these packages before installing PyOPA.

There are two simple ways to install PyOPA, the first, probably more convenient, is by using *pip*:

```
pip install PyOPA
```

This simple code snippet should automatically download PyOPA from *PyPi* and compile and install the necessary files according to your operating system. This means, that on UNIX-based systems *gcc*, and on Windows-based systems *Visual Studio Compiler* must be available.

If you are able to install *NumPy* and *Cython* without any problems by using *pip*, which means that you have the necessary means and permissions to download, compile and install a package from *PyPi*, no errors should occur during the installation of PyOPA.

Another possibility is to download, extract and install the package manually. After extracting the project into a separate folder, you should navigate into the directory and use

```
python install setup.py
```

to install PyOPA. This should compile and install all the necessary files without any problems. To check whether the unit tests produce correct results on your computer, you can simply run them by a singe function call:

```python
import PyOPA

PyOPA.run_tests()
```

Unless you have your own data files (where you define, for example, the distance matrix, gap opening and extension costs), you have to use one of the matrices coming with PyOPA. All the data files that is necessary to run the unit tests and future operations on sequences are automatically installed into a separate folder (*<sys.prefix>/PyOPA_test/*) during installation. If you cannot locate or access the data files, you can download PyOPA and extract all of the files manually. The data files should be available under *test/data/*, which you can use for future alignments.

Under *test/* and *examples/* you can find a variety of well-commented examples for all of the features offered by PyOPA.

3 Features

3.1 Normalizing Sequences

The C core does not operate on sequences consist of upper-case letters of the English alphabet, but instead it uses byte arrays generated from the original sequences by deducting *chr('A')* from every single character in the given sequences. Because of this, for every single operation the participant sequences should be transformed, which is clearly a waste of time.

Therefore, every function and method that operates on sequences has an optional (false by default) parameter, where you can specify whether the given parameter is already transformed. When we align multiple consecutive sequences, like an all against all, we should normalize the sequences beforehand:

```
s1_norm = cython_swps3.normalize_sequence(s1)
s2_norm = cython_swps3.normalize_sequence(s2)

#normalizing sequences in the background
print cython_swps3.align_double(s1, s2, generated_env)

#passing normalized sequences and setting the is_normalized
# flag, a more efficient solution
print cython_swps3.align_double(s1_norm, s2_norm, generated_env,
    True)
```

3.2 Creating AlignmentEnvironments

In order to operate on sequences, the distance matrix, the gap opening and extension costs and a threshold of interest must be provided. These data are stored in a so-called *AlignmentEnvironment* class in Python, which can be used for future alignments.

The *AlignmentEnvironment* also stores different matrices and gap costs calculated from the original, provided ones. To create an environment you should provide the necessary data in a dictionary:

```
data = {'gap_open': -20.56,
        'gap_ext': -3.37,
        'pam_distance': 150.87,
        'scores': [[10.0]],
        'column_order': 'A',
        'threshold': 50.0}

env = cython_swps3.create_environment(**data)

#prints [30.0, 2, 2, 0, 0], the first element is the score
print cython_swps3.align_double('AAA', 'AAA', env)

#prints [0.0, -1, -1, 0, 0], the score is 0
# since the score for 'A -> T' is undefined
print cython_swps3.align_double('TTT', 'AAA', env)
```

In this simple example, we used a really simple distance matrix with a single element, which means only the $A \to A$ score is defined. When you create an *AlignmentEnvironment* the dimension of the matrix and the length of the columns must be the same, however, during the construction this provided distance matrix is always extended into a 26x26 one, which consists of the scores for [A-Z]. The undefined cells of the matrix are filled with zeros (in this case every cell, except for the first one).

Because of this, aligning *AAA* to *AAA* gives a result of 30.0, whereas aligning the same sequence to *TTT* results in zero.

Alternatively, you can either read a single *AlignmentEnvironment* or multiple ones from a JSON-formatted file:

```
env_list = cython_swps3.read_all_env_json(
    os.path.join(data_dir, 'all_matrices.json'))
log_pam1 = cython_swps3.read_env_json(
    os.path.join(data_dir, 'logPAM1.json'))
```

Precomputed matrices stored in JSON are also coming with the package, see in Section 2.

Another possibility is to generate the distance matrices and gap costs from a single *AlignmentEnvironment*:

```
#generates a signle AlignmentEnvironment
# with a pam distance of 250
generated_env = cython_swps3.generate_env(log_pam1_env, 250)

#generates 1000 environments for different pam distances
gen_env_list = cython_swps3.generate_all_env(log_pam1_env, 1000)
```

For each generation a log_pam1 *AlignmentEnvironment* should be used.

3.3 Byte and short estimations

Instead of calculating the actual double score for two given sequences, we can rapidly compute an upper bound of the score. If this upper bound score is lower than the threshold of interest, we can omit the calculation of the double precision score for these particular sequences.

If the upper bound score is over the threshold, however, we have do another alignment to get the actual score, which can still be lower than the threshold. Since the vast majority of the scores is usually lower than the threshold, a substantial amount of time can be saved by using the byte or short estimation, because an estimation runs much faster than a double alignment.

To produce an upper bound by the short (or the byte) method, we have to transform the double distance matrix and the gap costs to shorts (or bytes). The transformation has to be done in a way that the future calculations will surely produce an upper bound score. Such a transformation could be $ceil(x)$ for every element, but this would result in a really imprecise upper bound score. Instead, we use $ceil(x \cdot factor)$ with $factor = UNSIGNED\_SHORT\_MAX/threshold$ for the short version, and a bit more complicated formula for the byte version. The factor should be designed in a way that it reduces the relative rounding errors, for a more precise estimation.

After the transformations, we can cram multiple elements into a single register (8 for the short version and 16 for the byte version, assuming 128-bit SSE registers) and perform the calculation. Of course the result score has to be scaled back by the used $factor$.

To hasten the estimation further, we can create so-called profiles for a given S sequence and matrix combination. After a profile is created, we can use it for multiple alignments to the same S and matrix combination (the gap scores can be changed, by redefining in the provided *AlignmentEnvironment*):

```python
s1 = 'AATCGGA'
s2 = 'AAAA'
s3 = 'CATACCTGGTGTGATGCC'

#not optimal, two hidden profile generation in the background
print cython_swps3.align_short(s1, s2, generated_env)
print cython_swps3.align_short(s1, s3, generated_env)
print cython_swps3.align_byte(s1, s2, generated_env)
print cython_swps3.align_byte(s1, s3, generated_env)

#one profile generation
profile = cython_swps3.AlignmentProfile()
profile.create_profiles(s1, generated_env)

#the following code produces the exact same result
#but is more efficient since it's
#using the same profile for multiple alignments
print profile.align_short(s2, generated_env)
print profile.align_short(s3, generated_env)
print profile.align_byte(s2, generated_env)
print profile.align_byte(s3, generated_env)
```

The scaled short and byte matrices, which are used for profile generation, and the gap costs are automatically computed during the creation of an *AlignmentEnvironment*. However, if you change the threshold, the element of the original matrix or the gap costs the short and byte ones have to be recomputed:

```
generated_env.threshold = 50.0
#WRONG: the short and byte matrices are not recomputed!!!
# the matrices and gap costs for the old threshold
# will be used
profile.create_profiles(s1, generated_env)

#recompute the byte/short matrices and gap costs
generated_env.create_scaled_matrices()

#and then create the profile
profile.create_profiles(s1, generated_env)

#now we can do alignments with the new profile:
print profile.align_short(s3, generated_env)
```

3.4 Computing the actual score

Whenever the estimation is over the threshold, we have to compute the actual score, which can still be below the threshold. In PyOPA there are two possible ways to do so, however, only one of them is an efficient vectorised approach.

The first available Python function is a reference implementation of the Smith-Waterman algorithm, and should run on every system without problems. It does not use any vectorisation, therefore it is clearly not an efficient solution. This reference implementation is only capable of local alignments and provides a single double precision score as a result but not the ranges of the local alignment:

```
#always a local alignment
print cython_swps3.align_scalar_reference_local(s1, s2,
    generated_env)
```

The second, more efficient and vectorised solution is also capable of global alignments and can also provide the ranges of the local alignment on demand. By setting the *stop_at_threshold* argument true, we can terminate the score computation if it reaches the provided threshold:

```
s1 = 'AATCGGA'
s3 = 'CATACCTGGTGTGATGCC'
#operating on normalized sequences, does not stop at threshold,
#  it is NOT a global alignment, and computes the ranges
```

```python
#returns [19.946195452221108, 6, 8, 0, 2], the first element is the score
# the 4 other elements are [max1, max2, min1, min2] the ranges
print cython_swps3.align_double(s1_norm, s3_norm, generated_env, True, False, False, True)

#returns [score, max1, max2]
print cython_swps3.align_double(s1_norm, s3_norm, generated_env, True, False, False, False)

generated_env.threshold = 10.0
#no generated_env.create_scaled_matrices() is needed
# because we do not operate on short/byte matrices

#results in [11.499268729503227, 3, 0, 3, 0], not the best possible
# local alignment, but still over the threshold of 10.0
print cython_swps3.align_double(s1_norm, s3_norm, generated_env, True, True, False, True)

#global alignment, stop at threshold is ignored
print cython_swps3.align_double(s1, s3, generated_env, False, True, True, True)
```

In the example given above, *s1[0:6]* = *AATCGGA* has been aligned to *s3[2:8]* = *TACCTGG*. If we do not require the full ranges, the *max1* and *max2* values are still provided, so in this case the result is an array with three elements.

3.5 Getting the concrete alignment

In some cases we might be interested not only in the score, but also in a concrete alignment of the two sequences. With PyOPA this can be computed by using a single function, however, this function requires a huge amount of stack to work correctly. On UNIX-based systems you can unlimit your stack size by using the following Python code snippet:

```python
import resource

resource.setrlimit(resource.RLIMIT_STACK, (resource.RLIM_INFINITY, resource.RLIM_INFINITY))
```

Another, more general approach is to start the operation in a new thread with a given stack size:

```
def nt_align(s1, s2, env, is_global, aligned_strs):
    print 'Concrete %s alignment:' % ('global' if is_global else 'local')
    tmp_aligned_strings = cython_swps3.align_strings(s1, s2, env, False, is_global)
    print '\taligned_s1: %s' % tmp_aligned_strings[0]
    print '\taligned_s2: %s' % tmp_aligned_strings[1]
    aligned_strs.extend(tmp_aligned_strings)

s1 = 'PISRIDNNKITTTLGNTGIISVTIGVIIFKDLHAKVHGF'
s2 = 'PIERIENNKILANTGVISVTIGVIIYQDLHADTVMTSDY'
threading.stack_size(100000000)

# aligned_s1: PISRIDNNKITTTLGNTGIISVTIGVIIFKDLHAKV
# aligned_s2: PIERIENNKI___LANTGVISVTIGVIIYQDLHADT
aligned_strings = []
t = threading.Thread(None, nt_align,
                          'Aligning Thread', (s1, s2, generated_env, False, aligned_strings))
t.start()
t.join()
print aligned_strings
```

3.6 EstimatePam

For two given aligned sequences, the *EstimatePam* function computes maximum likelihood estimates on the score and pam distance, and the variance.

From a list of *AlignmentEnvironments* and a *log_pam1* environment we can create the necessary data structure which can be used for computing the estimations.

```
dms = cython_swps3.MutipleAlEnv(gen_env_list, log_pam1_env)

#returns an array: [similarity, pam_distance, variance]
print dms.estimate_pam(aligned_strings[0], aligned_strings[1])
```

The first element of the array is the estimated similarity score, which is always higher than the actual score, the second element is the estimated evolutionary distance and the third one is the variance.

The input strings (*s1* and *s2*) must have the same length, since they have to be a concrete alignment pair produced by *align_strings* (see in Section 3.5).

4 Comparison with PyCogent

5 Conclusion

With its core written in C, PyOPA offers and efficient vectorized implementation of the Smith-Waterman algorithm as well as many other features including byte and short estimations and computing the concrete strings.

All of the operations can be performed with the convenience of Python, in a platform independent way, which makes it an enticing choice for applications operating on immense amount of sequences, where efficiency is of paramount importance.