BioPython Problems

By: Anmol Panchal

1- Sequence Alignment Problem

Pairwise sequence alignment using a dynamic programming algorithm.

This provides functions to get global and local alignments between two sequences. A global alignment finds the best concordance between all characters in two sequences. A local alignment finds just the subsequences that align the best.

When doing alignments, you can specify the match score and gap penalties. The match score indicates the compatibility between an alignment of two characters in the sequences. Highly compatible characters should be given positive scores, and incompatible ones should be given negative scores or 0. The gap penalties should be negative.

The names of the alignment functions in this module follow the convention <alignment type>XX where <alignment type> is either "global" or "local" and XX is a 2 character code indicating the parameters it takes. The first character indicates the parameters for matches (and mismatches), and the second indicates the parameters for gap penalties.

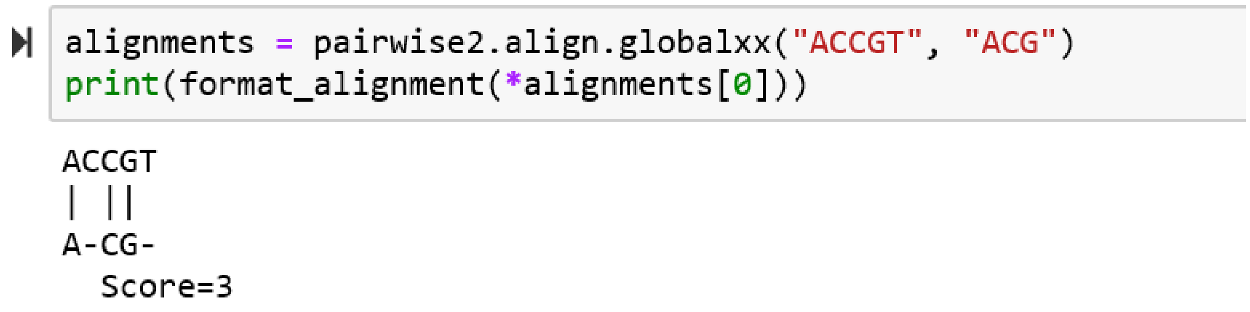
The match parameters are:

CODE	DESCRIPTION
x	No parameters. Identical characters have score of 1, otherwise 0.
m	A match score is the score of identical chars, otherwise mismatch score.
d	A dictionary returns the score of any pair of characters.
c	A callback function returns scores.

The gap penalty parameters are:

CODE	DESCRIPTION
x	No gap penalties.
s	Same open and extend gap penalties for both sequences.
d	The sequences have different open and extend gap penalties.
c	A callback function returns the gap penalties.

All the different alignment functions are contained in an object align. For example:



```
alignments = pairwise2.align.globalxx("ACCGT", "ACG")
print(format_alignment(*alignments[0]))
```

```
ACCGT
| ||
A-CG-
  Score=3
```

All alignment functions have the following arguments:

Two sequences: strings, Biopython sequence objects or lists. Lists are useful for supplying sequences which contain residues that are encoded by more than one letter.

penalize_extend_when_opening: boolean (default: False). Whether to count an extension penalty when opening a gap. If false, a gap of 1 is only penalized an "open" penalty, otherwise it is penalized "open+extend".

penalize_end_gaps: boolean. Whether to count the gaps at the ends of an alignment. By default, they are counted for global alignments but not for local ones. Setting penalize_end_gaps to (boolean, boolean) allows you to specify for the two sequences separately whether gaps at the end of the alignment should be counted.

gap_char: string (default: '-'). Which character to use as a gap character in the alignment returned. If your input sequences are lists, you must change this to ['-'].

force_generic: boolean (default: False). Always use the generic, non-cached, dynamic programming function (slow!). For debugging.

score_only: boolean (default: False). Only get the best score, don't recover any alignments. The return value of the function is the score. Faster and uses less memory.

one_alignment_only: boolean (default: False). Only recover one alignment.

The other parameters of the alignment function depend on the function called. Some examples:

Find the best global alignment between the two sequences. Identical characters are given 1 point. No points are deducted for mismatches or gaps.

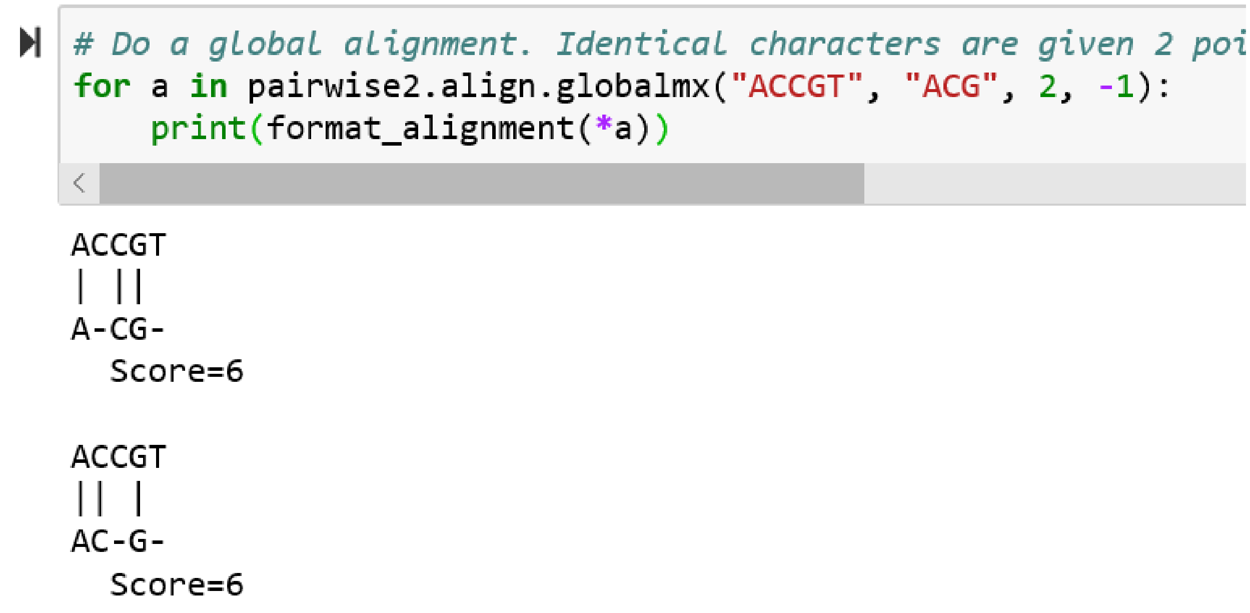
```python
for a in pairwise2.align.globalxx("ACCGT", "ACG"):
    print(format_alignment(*a))
```

```
ACCGT
| ||
A-CG-
  Score=3

ACCGT
|| |
AC-G-
  Score=3
```

```python
# Same thing as before, but with a local alignment.
for a in pairwise2.align.localxx("ACCGT", "ACG"):
    print(format_alignment(*a))
```

```
ACCGT
| ||
A-CG-
  Score=3

ACCGT
|| |
AC-G-
  Score=3
```

Next, we will try to do for a global alignment. Identical characters are given 2 points, 1 point is deducted for each non-identical character. Don't penalize gaps.



```
# Do a global alignment. Identical characters are given 2 poi
for a in pairwise2.align.globalmx("ACCGT", "ACG", 2, -1):
    print(format_alignment(*a))
```

```
ACCGT
| ||
A-CG-
  Score=6

ACCGT
|| |
AC-G-
  Score=6
```

Compiling the code to do a global sequence alignment is shown below:

Code:

```
# Define two sequences to be aligned

X = "ACGGGT"

Y = "ACG"


# Get a list of the global alignments between the two sequences ACGGGT and ACG

# No parameters. Identical characters have score of 1, else 0.

# No gap penalties.

alignments = pairwise2.align.globalxx(X, Y)


# Use format_alignment method to format the alignments in the list

for a in alignments:

  print(format_alignment(*a))
```

Output:

```
ACGGGT
||  |
AC--G-
  Score=3

ACGGGT
|| |
AC-G--
  Score=3

ACGGGT
|||
ACG---
  Score=3
```

2- Random shuffling of DNA sequence

Background

Randomly shuffled sequences are routinely used in sequence analysis to evaluate the statistical significance of a biological sequence. For example, a common method for assessing the thermodynamic stability of an RNA sequence is to compare its folding free energy with those of a large sample of random sequences. It is known that the stability of an RNA secondary structure depends crucially on the stackings of adjacent base pairs; therefore the frequencies of distinct doublets in the random sequences are important considerations in such analysis. Besides, natural biological sequences often manifest certain nearest-neighbor patterns: both eukaryotic and prokaryotic nucleic acid sequences show a consistent hierarchy in the doublet frequencies; in coding regions, the codon usage can also be markedly nonuniform. In many cases, biologists need sophisticated shuffling tools that preserve not only the counts of distinct letters but also higher-order statistics such as doublet counts, triplet counts, and, in general, k-let counts.

Code:

```
# NOTE: One cannot use function "count(s,word)" to count the number

# of occurrences of dinucleotide word in string s, since the built-in

# function counts only nonoverlapping words, presumably in a left to

# right fashion.


import numpy as np

import sys,string,random

from random import *


def computeCountAndLists(s):

  #WARNING: Use of function count(s,'UU') returns 1 on word UUU

  #since it apparently counts only nonoverlapping words UU

  #For this reason, we work with the indices.


  #Initialize lists and mono- and dinucleotide dictionaries

  List = {} #List is a dictionary of lists

  List['A'] = []; List['C'] = [];

  List['G'] = []; List['T'] = [];
  nuclList   = ["A","C","G","T"]
  s       = s.upper()
  s       = s.replace("T","T")
  nuclCnt    = {}  #empty dictionary
  dinuclCnt  = {}  #empty dictionary
  for x in nuclList:
    nuclCnt[x]=0
    dinuclCnt[x]={}
    for y in nuclList:
      dinuclCnt[x][y]=0

  #Compute count and lists
  nuclCnt[s[0]] = 1
  nuclTotal     = 1
  dinuclTotal   = 0
  for i in range(len(s)-1):
    x = s[i]; y = s[i+1]
    List[x].append( y )
    nuclCnt[y] += 1; nuclTotal  += 1
    dinuclCnt[x][y] += 1; dinuclTotal += 1
  assert (nuclTotal==len(s))
  assert (dinuclTotal==len(s)-1)
  return nuclCnt,dinuclCnt,List


def chooseEdge(x,dinuclCnt):
  numInList = 0
  for y in ['A','C','G','T']:
    numInList += dinuclCnt[x][y]
  z = random.random()
  denom=dinuclCnt[x]['A']+dinuclCnt[x]['C']+dinuclCnt[x]['G']+dinuclCnt[x]['T']
  numerator = dinuclCnt[x]['A']
  if z < float(numerator)/float(denom):
    dinuclCnt[x]['A'] -= 1
    return 'A'
  numerator += dinuclCnt[x]['C']
  if z < float(numerator)/float(denom):
    dinuclCnt[x]['C'] -= 1
    return 'C'
  numerator += dinuclCnt[x]['G']
  if z < float(numerator)/float(denom):
    dinuclCnt[x]['G'] -= 1
    return 'G'
  dinuclCnt[x]['T'] -= 1
  return 'T'


def connectedToLast(edgeList,nuclList,lastCh):
  D = {}
  for x in nuclList: D[x]=0
  for edge in edgeList:
    a = edge[0]; b = edge[1]
    if b==lastCh: D[a]=1
  for i in range(2):
    for edge in edgeList:
      a = edge[0]; b = edge[1]
      if D[b]==1: D[a]=1
  ok = 0
  for x in nuclList:
    if x!=lastCh and D[x]==0: return 0
  return 1
```

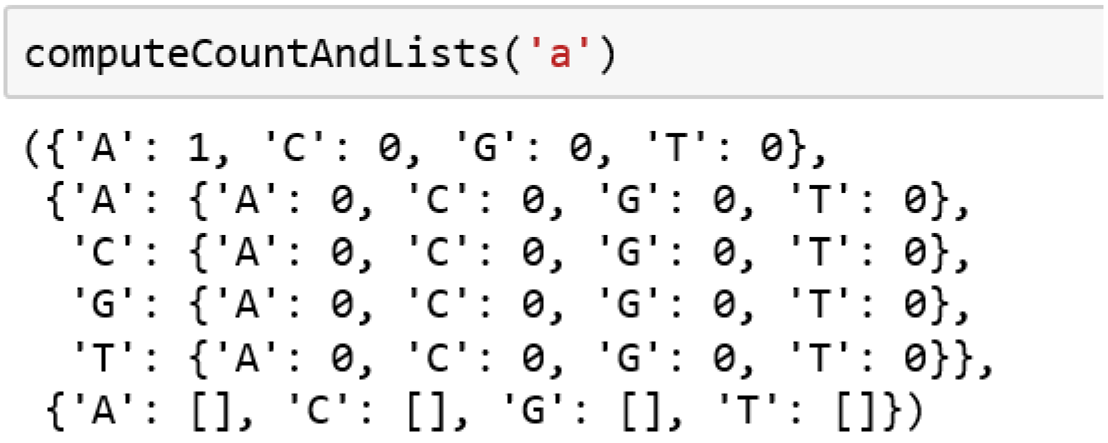
An Eulerian orientation of an undirected graph G is an assignment of a direction to each edge of G such that, at each vertex v, the indegree of v equals the outdegree of v. Such an orientation exists for any undirected graph in which every vertex has even degree, and may be found by constructing an Euler tour in each connected component of G and then orienting the edges according to the tour. Every Eulerian orientation of a connected graph is a strong orientation, an orientation that makes the resulting directed graph strongly connected.

```
def eulerian(s):
  nuclCnt,dinuclCnt,List = computeCountAndLists(s)
  #compute nucleotides appearing in s
  nuclList = []
  for x in ["A","C","G","T"]:
    if x in s: nuclList.append(x)
  #compute numInList[x] = number of dinucleotides beginning with x
  numInList = {}
  for x in nuclList:
    numInList[x]=0
    for y in nuclList:
      numInList[x] += dinuclCnt[x][y]
  #create dinucleotide shuffle L
  firstCh = s[0]  #start with first letter of s
  lastCh  = s[-1]
  edgeList = []
  for x in nuclList:
    if x!= lastCh: edgeList.append( [x,chooseEdge(x,dinuclCnt)] )
  ok = connectedToLast(edgeList,nuclList,lastCh)
  return ok,edgeList,nuclList,lastCh
```

Randomly shuffled sequences are routinely used in sequence analysis to evaluate the statistical significance of a biological sequence. In many cases, biologists need sophisticated shuffling tools that preserve not only the counts of distinct letters but also higher-order statistics such as doublet counts, triplet counts, and, in general, k-let counts.

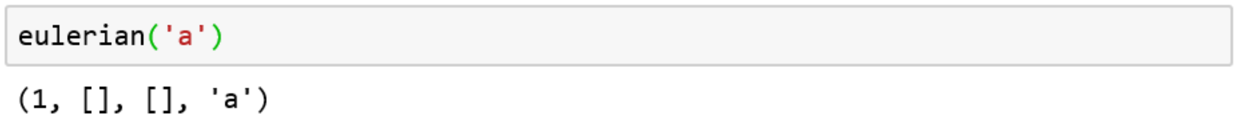
```
def shuffleEdgeList(L):
 n = len(L); barrier = n # n = randmax here in our program and we will run for loop for n iterations.
 for i in range(n):
  z = int(np.random.rand() * barrier)
  print("z",z)
  tmp = L[z]
  L[z]= L[barrier-1]
  L[barrier-1] = tmp
  barrier -= 1
  print("barrier",barrier) #barrier runs till n i.e randmax
 return L
```

Output:



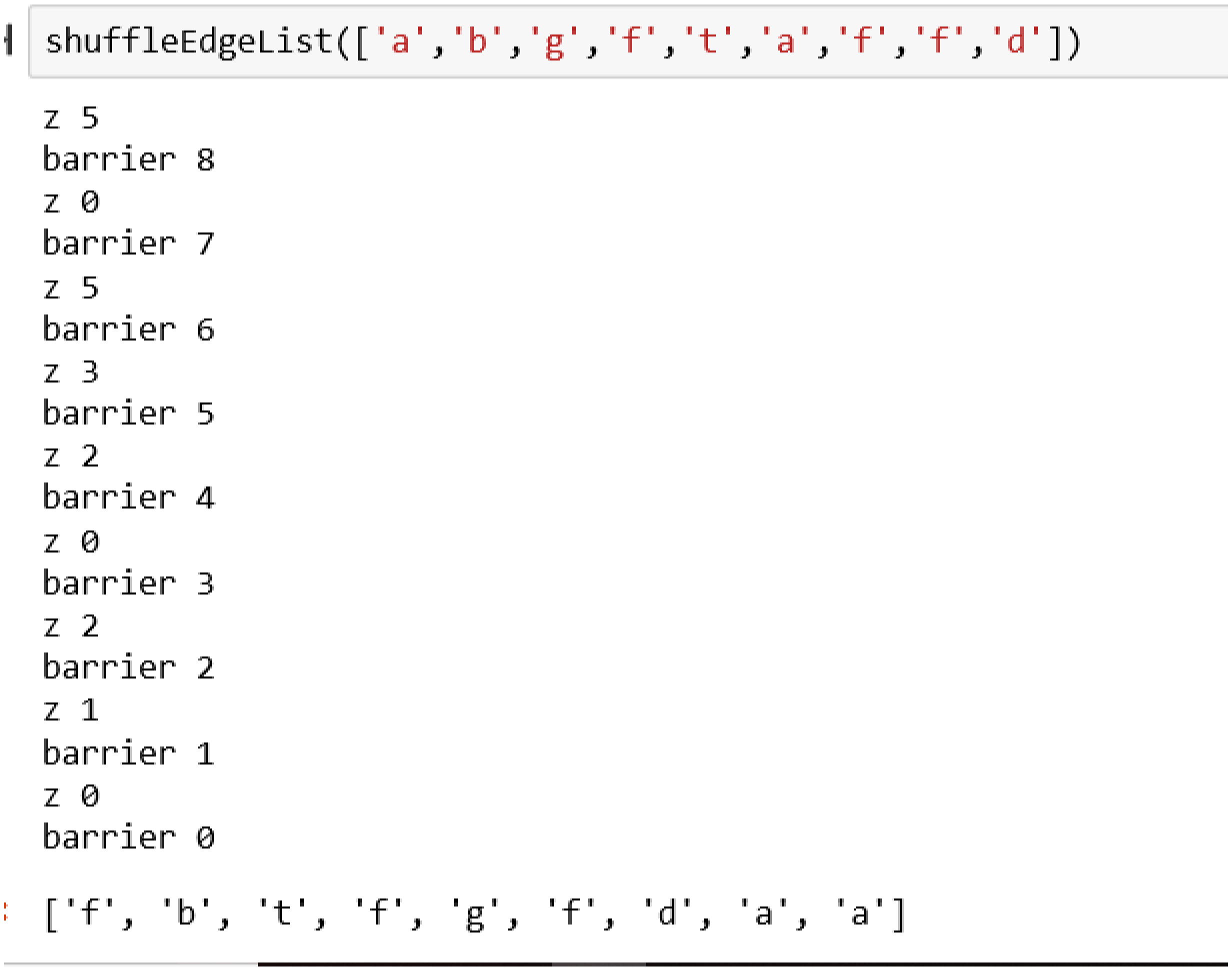
```
computeCountAndLists('a')
```

```
({'A': 1, 'C': 0, 'G': 0, 'T': 0},
 {'A': {'A': 0, 'C': 0, 'G': 0, 'T': 0},
  'C': {'A': 0, 'C': 0, 'G': 0, 'T': 0},
  'G': {'A': 0, 'C': 0, 'G': 0, 'T': 0},
  'T': {'A': 0, 'C': 0, 'G': 0, 'T': 0}},
 {'A': [], 'C': [], 'G': [], 'T': []})
```



```
eulerian('a')
```

```
(1, [], [], 'a')
```



```
shuffleEdgeList(['a','b','g','f','t','a','f','f','d'])
```

```
z 5
barrier 8
z 0
barrier 7
z 5
barrier 6
z 3
barrier 5
z 2
barrier 4
z 0
barrier 3
z 2
barrier 2
z 1
barrier 1
z 0
barrier 0
```

```
['f', 'b', 't', 'f', 'g', 'f', 'd', 'a', 'a']
```