MSARC – implementation details

Michał Modzelewski Norbert Dojer

The MSARC implementation is written in the Python language in the version 2.7 dialect. It makes heavy use of the numpy library for array operations, as well as scipy.sparse for storing the posterior probability matrices in compressed row format. The BioPython library is used for sequence input and output. A customized version of the probA program is used for the pairwise stochastic alignment of sequences. The probA implementation of the partition function was chosen because the available source code provides several substitution matrices to work with, and solves the problem of numerical overflow when aligning large sequences. The probA tool is exposed to Python as an extension module written using Cython. The consistency transformation, the main loop of the Fiduccia-Mattheyses partitioner, the gains computation methods, and the maximal expected accuracy alignment algorithm are also written as Cython extension modules for speed. Code listings below are simplified for legibility.

1 Module balibase

The balibase module is a thin wrapper around the bali_score program from BAliBASE 3.0, which is expected to be in the same directory as the calling script. It is used by the balitest and balitest-compare programs.

2 Module clusteralign

This module contains the main implementation of the MSARC method conceptually split into several files.

alignment.py

Contains the definition of the Alignment class which describes and can construct an alignment. This class is the main access point of the MSARC algorithm. An instance of Alignment is created with a string containing the name of a file containing the aligned sequences in *fasta* format, and keyword arguments containing the algorithm parameters. An instance of Alignment exposes some public methods and properties.

alignment A property for accessing the alignment as a MultipleSeqAlignment instance. This can be calculated on access from the sequences.

value A property that returns the sum of pairwise posterior probabilities value of the alignment.

```
1  @property
2  def value(self):
3      # self._columns is a list of Graph instances
4      return sum(c.value for c in self._columns)
```

output A method that prints the alignment to the screen in any format supported by the BioPython library or the *msf* format (default).

write A method that writes the alignment to the named file in any format supported by the BioPython library or the *msf* format (default).

read A method that reads an alignment from the named file.

Accessing the alignment property causes an alignment to be calculated if the object does not yet contain an alignment. To this end the alignment property getter method calls some private methods.

_relax A method that performs the consistency transformation on the posterior probabilities the desired number of times. The actual consistency transformation is delegated to the Graph class.

_align A method that recursively partitions the alignment graph (in the form of an instance of the Graph class) until a list of columns is produced.

```
def _align(self, graph_or_starts, ends=None):
    """Align the sequences."""
    if isinstance(graph_or_starts, Graph):
        graph = graph_or_starts
    else:
        graph = self._graph.slice(graph_or_starts, ends)
    graphs = [graph]
    columns = []
    while len(graphs) > 0:
        g = graphs.pop()
        # Graph depicts a single column, add to columns
        if g.is_column:
            columns.append(g)
            continue
        # Iterativly partition the graph
        l, r = g.split()
        graphs.append(r)
        graphs.append(l)
    return columns
```

_refine A method that performs iterative refinement on the alignment.

graph.py

The definition of the Graph class which is an instance of an alignment graph. An instance of Graph is created with a list of SeqRecord objects describing the aligned sequences, and keyword arguments containing the algorithm parameters. The Graph class knows how to partition itself, return a slice of itself, perform an iteration of the consistency transformation, and coarsen and uncoarsen itself for the purpose of multilevel partitioning. It populates, on demand, the values of the weights of the edges using the clusteralign.probA module. Some public properties and methods of the Graph class:

size A property returning the total number of nodes in the graph.

num_sequences A property returning the number of sequences.

lengths A property that returns an array of lengths of the sequences.

max_length A property that returns the length of the longest sequence.

value A property returning the total *value* of the graph. Only inter-sequence edges are counted, because only those can remain uncut after alignment.

```
@property
def value(self):
    return sum((m.sum() for m in self._pair_pp.flat if m is not None))
```

starts A property that returns an array of starting points of the sub-sequences within the sequences.

ends A property that returns an array of end points of the sub-sequences within the sequences.

is_column A property that returns whether the graph represents a valid *column*.

```
@property
def is_column(self):
    return self.max_length <= 1
```

nonzero A property returning the number of non-zero length sequences.

```
@property
def nonzero(self):
    return (self.lengths > 0).sum()
```

partition A property that returns the partition of the graph as an array of partition points within the sequences.

relax A method that runs an iteration of the consistency transformation.

```
from numpy import empty_like
from clusteralign.matrix import prune
from clusteralign._graph import relax_XZ_ZY, relax_ZX_ZY, relax_XZ_YZ

def relax(self, weighted_transform=False):
    pair_pp = self._pair_pp
    self.__pair_pp = new_pair_pp = empty_like(pair_pp)
    cut = self._params['cut']
    # for every pair of sequences
    n = self.num_sequences
    weights=[[1]*n for i in xrange(n)]
    if weighted_transform:
        for x in xrange(n):
            for y in xrange(x + 1, n):
                weights[x][y]=pair_pp[x,y].sum()/min(pair_pp[x,y].shape)
    for x in xrange(n):
        for y in xrange(x + 1, n):
            # P|(z==x) + P|(z==y)
            sum_w = weights[x][y]*2
            new_pp_xy = pair_pp[x, y] * sum_w
            # P|(z!=x&&z!=y)
            for z in xrange(n):
                if z is x or z is y:
                    continue
                if z < x:
                    relax_ZX_ZY(pair_pp[z, x], pair_pp[z, y], new_pp_xy, weights[z][x]*weights[z][
                    sum_w += weights[z][x]*weights[z][y]
                elif x < z < y:
                    relax_XZ_ZY(pair_pp[x, z], pair_pp[z, y], new_pp_xy, weights[x][z]*weights[z][
                    sum_w += weights[x][z]*weights[z][y]
                else:
                    relax_XZ_YZ(pair_pp[x, z], pair_pp[y, z], new_pp_xy, weights[x][z]*weights[y][
                    sum_w += weights[x][z]*weights[y][z]
            # normalize
            new_pp_xy.data /= sum_w
            # cut-off
            prune(new_pp_xy, cut)
            new_pair_pp[x, y] = new_pp_xy
```

slice A method that returns a sub-graph corresponding to sub-sequences with the given start and end points.

split A method that splits the graph into to sub-graphs along the given partition.

```
def split(self, parts=None):
    if parts is None:
        parts = self.partition
    return (self.slice(self.starts, parts),
            self.slice(parts, self.ends))
```

matrix.py

Some helper functions that operate on numpy arrays.

prune A function that zeros all values in a matrix below a given threshold.

```
from scipy.sparse import isspmatrix_csr

def prune(array, cut):
    assert isspmatrix_csr(array)
    array.data[array.data < cut] = 0.
    array.eliminate_zeros()
```

msf.py

Unlike BioPerl, the Python library BioPython does not support the *msf* multiple sequence alignment format. Since this is the preferred alignment format of the BAliBASE benchmark, a custom implementation was required. This module provides functions for reading and writing such files.

options.py

This module implements an options parser for the alignment programs using Python's built-in argparse library.

partition.py

The Graph class delegates the heavy lifting of searching for an optimal partition to the GraphPartitioner class implemented in this module. A GraphPartitioner instance is created with an instance of Graph that is to be partitioned. The public method partition is exposed that takes an initial partition as an argument and returns a new partition optimized by an implementation of the Fiduccia-Mattheyses algorithm. Helper properties and methods implement the details of the process.

partition A method that finds a balanced minimum-cut partition of the alignment graph by iterating the Fiduccia-Mattheyses algorithm until convergence.

```
def partition(self, parts):
    self._parts = parts
    while True:
        best_parts = self._testmoves()
        if best_parts is None:
            break
        self._move([(i, best_parts[i]) for i in xrange(self._numgroups)])
    return self._parts
```

_testmoves Iteratively selects nodes to move between the two sides of the partition, and calculates the gain.

```
from clusteralign._partition import bestmove

def _testmoves(self):
    parts = self._parts.copy()

    gain = 0.
    bestgain = None if min(self._balance_weight) < self._minbalance else gain
    bestparts = parts

    allowed = self._new_allowed(parts)
    while True:
        if self._balance_weight[0] < self._minbalance:
            direction = 1
        elif self._balance_weight[1] < self._minbalance:
            direction = -1
        else:
            direction = 0

        move = bestmove(self._parts, self._gains, allowed, direction)
        if move is None:
            break
        group, partition, delta_gain = move
```

```
23          gain += delta_gain
24
25          self._movepartition(group, partition)
26          allowed[group][partition] = 0
27
28          if min(self._balance_weight) < self._minbalance:
29              continue
30          if bestgain is None or gain > bestgain:
31              bestparts = self._parts.copy()
32              bestgain = gain
33
34      if bestparts is parts:
35          self._parts = parts
36          return None
37      return bestparts
```

_move Performs a series of moves.

_movepartition Moves the partition boundary and recalculates balance and gains.

```
1  def _movepartition(self, group, partition):
2      if partition == self._parts[group]:
3          #no change
4          return
5
6      if self.__gains is not None:
7          self._fixgains(group, partition)
8
9      p_group = self._parts[group]
10     max_partition, min_partition = max(partition, p_group), min(partition, p_group)
11     dweights = self._weights_cumsum[group][max_partition - 1]
12     if min_partition > 0:
13         dweights -= self._weights_cumsum[group][min_partition - 1]
14
15     side = self._getside(group, partition)
16     other = 1 if side == 0 else 0
17     self._balance_weight[side] -= dweights
18     self._balance_weight[other] += dweights
19
20     self._parts[group] = partition
```

_edges_sum A property that returns the edge weight matrix.

_calcgain_all Calculates the gains of moving the partition point to any other point in the sequence.

```
1  def _calcgain_all(self, group):
2      l_group = self._lengths[group]
3      gains = zeros(l_group + 1, dtype=float)
4      if not l_group:
5          return gains
6
7      for i in xrange(self._numgroups):
8          if i == group:
9              continue
10
11         p_i = self._parts[i]    # the current point of partition
12         l_i = self._lengths[i]
13
14         edges = self._edges[group, i]
15         if p_i > 0:
16             gains[1:] += edges[:, :p_i].sum(axis=1)
17         if p_i < l_i:
18             gains[1:] -= edges[:, p_i:].sum(axis=1)
19
20     # divide by 2 here so we don't have to double the difference in
21     # _fixgain_all, which is called more frequently
22     return gains / 2.
```

_calcgains Calculate the gains for all the sequences.

```
def _calcgains(self):
    g = empty(self._numgroups, dtype=object)
    for i in xrange(self._numgroups):
        g[i] = self._calcgain_all(i)
    return g
```

_fixgain_all Adjusts the gains for the sequence after the partition has been moved in another sequence.

```
def _fixgain_all(self, group, move_group, move_partition):
    if move_group == group:
        raise ValueError
    l_group = self._lengths[group]
    p_move_group = self._parts[move_group]
    gains = zeros(l_group + 1, dtype=float)
    if not l_group or move_partition == p_move_group:
        return gains

    edges = self._edges[group, move_group]
    if move_partition > p_move_group:
        gains[1:] += edges[:, p_move_group:move_partition].sum(axis=1)
    else:   # less
        gains[1:] -= edges[:, move_partition:p_move_group].sum(axis=1)

    return gains
```

_fixgains Adjusts the gains for all the sequences.

```
def _fixgains(self, group, partition):
    p_group = self._parts[group]
    if partition == p_group:
        #no change
        return
    g = self._gains
    for i in xrange(self._numgroups):
        if i == group or not self._lengths[i]:
            continue
        g[i] += self._fixgain_all(i, group, partition)
    g[group] -= g[group][partition]
```

probA.py

This module defines functions for calculating the pairwise posterior probabilities of sequence residue alignment. The actual calculations are delegated to the clusteralign._probA module.

align A function that returns the pairwise posterior probabilities matrix for the given pair of sequences.

pairwise A function that returns a matrix of pairwise posterior probabilities matrices for the sequences provided.

sequence.py

A module implementing a subclass of BioPython's Seq class called EmptySeq that is an empty sequence. This is required when partitioning the alignment graph leaves a slice of one sequence that is of zero length.

_graph.pyx

A Cython extention module that implements efficient helper functions for the clusteralign.graph module.

relax_XZ_ZY A function that computes a sparse matrix multiplication and accumulates the result in a third sparse matrix. Only results that are non-zero in the third matrix are stored. Sparse matrices are stored in index sorted compressed row format.

```
def relax_XZ_ZY(matXZ, matZY, matXY, weight):
    for x in range(matXY.shape[0]):
        startXY = matXY.indptr[x]
        endXY = matXY.indptr[x + 1]
        for z in range(matXZ.indptr[x], matXZ.indptr[x + 1]):
            startZY = matZY.indptr[maxXZ.indices[z]]
            endZY = matZY.indptr[maxXZ.indices[z] + 1]
            valXZ = matXZ.data[z]
            iterXY = startXY
            indexZY = matZY.indices[startZY]
            indexXY = matXY.indices[iterXY]
            while startZY < endZY and iterXY < endXY:
                if indexZY < indexXY:
                    startZY += 1
                    indexZY = maxZY.indices[startZY]
                elif indexZY > indexXY:
                    iterXY += 1
                    indexXY = matXY.indices[iterXY]
                else:   # equal
                    matXY.data[iterXY] += valXZ * matZY.data[startZY] * weight
                    startZY += 1
                    iterXY += 1
                    indexZY = maxZY.indices[startZY]
                    indexXY = matXY.indices[iterXY]
```

relax_ZX_ZY Analogous to the previous function but for the case when the first matrix is transposed.

relax_XZ_YZ Analogous to the previous function but for the case when the second matrix is transposed.

_partition.pyx

A companion module for the clusteralign.partition module that efficiently implements long running code as an extension module for Python written in Cython.

bestmove A function that returns the best allowed move for the Fiduccia-Mattheyses algorithm.

```
def bestmove(parts, gains, allowed, direction):
    for i in range(num_groups):
        l = allowed[i].shape[0]
        if not l:
            continue
        if direction < 0:
            start, end = 0, parts[i]
        elif direction > 0:
            start, end = parts[i], l
        else:
            start, end = 0, l
        group_max_partition = -1
        group_max_gain = 0.
        gain = 0.
        curgain = 0.
        for j in range(min(start, curpart), max(end, curpart + 1):
            gain += gains_item[j]
            if j == curpart:
                curgain = gain
            if not allowed[i][j]:
                continue
            if group_max_partition == -1 or gain > group_max_gain:
                group_max_partition = j
                group_max_gain = gain
        if group_max_partition >= 0:
            group_max_gain -= curgain
            if max_partition == -1 or group_max_gain > max_gain:
                max_group = i
                max_partition = group_max_partition
                max_gain = group_max_gain
```

```
31
32      if max_partition == -1:
33          return None
34      return max_group, max_partition, max_gain
```

_ppalign.pyx

An extension module for Python written in Cython that implements the dynamic programing algorithm for computing the maximum expected accuracy alignment of a posterior probability matrix. This is used during iterative refinement by the clusteralign.alignment module.

align_probabilities A function that calculates the maximum expected accuracy alignment path for the supplied posterior probability matrix, and returns mappings of matches and spaces for the two sequences.

_probA.pyx

An extension module for Python written in Cython that wraps and exposes the customized probA tool.

align A function that returns the pairwise posterior probabilities matrix for the given pair of sequences and parameters, as calculated by the customized probA.

3 Module clusteralign.tests

Unit tests for the different parts of the clusteralign module, based on the Python unittest framework. The test suite can be run using a compatible test runner, for example using the nose test runner by running the nosetests executable in the msarc directory.

```
$ nosetests
```

4 Customized probA

The probA source was modified to ease interaction with the Python code and optimize the computations. Some small modifications were needed to allow the construction of consecutive alignments with different parameters (specifically the temperature parameter). The array of posterior probabilities is allocated as a contiguous block so that it can be exposed directly as a numpy array. The original probA source code ordered the input sequences by length, which was not required and disabled. The code for computing the forwards and backwards partition functions, and the posterior probabilities matrix was changed to minimize the amount of multiplications and exponent calculations performed.

Furthermore, the probA program has a notion of a canonical unambiguous alignment[1] that is detrimental to the multiple sequence alignment problem. Consider the following alignments

```
A - - - X X X X B        and        A X X X X - - - B
A Y Y Y - - - - B                   A - - - - Y Y Y B
```

The probA definition of a canonical alignment excludes the second alternative which leads to this modified

[1] Mückstein, U., Hofacker, I.L., Stadler, P.F.: *Stochastic pairwise alignments.* Bioinformatics 18 Suppl 2, S153–S160 (2002)

recursion for the Gotoh algorithm

$$M_{i,j} = \max \begin{cases} M_{i-1,j-1} + s(a_i, b_j) \\ E_{i-1,j-1} + s(a_i, b_j) \\ F_{i-1,j-1} + s(a_i, b_j) \end{cases} \quad (1)$$

$$E_{i,j} = \max \begin{cases} M_{i,j-1} + g_o \\ E_{i,j-1} + g_{ext} \end{cases} \quad (2)$$

$$F_{i,j} = \max \begin{cases} M_{i-1,j} + g_o \\ E_{i-1,j} + g_o \\ F_{i-1,j} + g_{ext} \end{cases} \quad (3)$$

And this modified recursion for the partition function

$$Z^M_{i,j} = \left(Z^M_{i-1,j-1} + Z^E_{i-1,j-1} + Z^F_{i-1,j-1}\right) e^{\beta s(a_i,b_j)} \quad (4)$$

$$Z^E_{i,j} = Z^M_{i,j-1} e^{\beta g_o} + Z^E_{i,j-1} e^{\beta g_{ext}} \quad (5)$$

$$Z^F_{i,j} = \left(Z^M_{i-1,j} + Z^E_{i-1,j}\right) e^{\beta g_o} + Z^F_{i-1,j} e^{\beta g_{ext}} \quad (6)$$

$$Z_{i,j} = Z^M_{i,j} + Z^E_{i,j} + Z^F_{i,j} \quad (7)$$

The probA source code was modified to use a symmetrical recursion. Either the full recursion by replacing equation 5 with

$$Z^E_{i,j} = \left(Z^M_{i,j-1} + Z^F_{i,j-1}\right) e^{\beta g_o} + Z^E_{i,j-1} e^{\beta g_{ext}} \quad (8)$$

and equation 2 with

$$E_{i,j} = \max \begin{cases} M_{i,j-1} + g_o \\ E_{i,j-1} + g_{ext} \\ F_{i,j-1} + g_o \end{cases} \quad (9)$$

or the restricted recursion by replacing equation 6 with

$$Z^F_{i,j} = Z^M_{i-1,j} e^{\beta g_o} + Z^F_{i-1,j} e^{\beta g_{ext}} \quad (10)$$

and equation 3 with the symmetrical pair of equation 2

$$F_{i,j} = \max \begin{cases} M_{i-1,j} + g_o \\ F_{i-1,j} + g_{ext} \end{cases} \quad (11)$$

Source modifications in *diff* format can be found in the patches sub-directory of the included probA source tree.