18: Biological Sequence Alignment

Last Update: June 25, 2013.

The Global Alignment Library [xalgoalign:include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

The library contains C++ classes encapsulating global pairwise alignment algorithms frequently used in computational biology.

- CNWAligner is the base class for the global alignment algorithm classes. The class provides an implementation of the generic Needleman-Wunsch for computing global alignments of nucleotide and amino acid sequences. The implementation uses an affine scoring scheme. An optional end-space free variant is supported, which is useful in applications where one sequence is expected to align in the interior of the other sequence, or the suffix of one string to align with a prefix of the other.
  The classical Needleman-Wunsch algorithm is known to have memory and CPU requirements of the order of the sequence lengths' product. If consistent partial alignments are available, the problem is split into smaller subproblems taking fewer operations and less space to complete. CNWAligner provides a way to specify such partial alignments (ungapped).
- CBandAligner encapsulates the banded variant of the global alignment algorithm which is applicable when the number of differences in the target alignment is limited ('the band width'). The computational cost of the algorithm is of the order of the band width multiplied by the length of the query sequence.
- CMMAligner follows Hirschberg's divide-and-conquer approach under which the amount of space required to align two sequences globally becomes a linear function of the sequences' lengths. Although the latter is achieved at a cost of up to twice longer running time, a multithreaded version of the algorithm can run even faster than the classical Needleman-Wunsch algorithm in a multiple-CPU environment.
- CSplicedAligner is an abstract base for algorithms computing cDNA-to-genome, or spliced alignments. Spliced alignment algorithms specifically account for splice signals in their dynamic programming recurrences resulting in better alignments for these particular but very important types of sequences.

Chapter Outline

The following is an outline of the chapter topics:

- Computing pairwise global sequence alignments
  - Initialization
  - Parameters of alignment
  - Computing
  - Alignment transcript

- Computing multiple sequence alignments
- Aligning sequences in linear space
  - The idea of the algorithm
  - Implementation
- Computing spliced sequences alignments
  - The problem
  - Implementation
- Formatting computed alignments
  - Formatter object

Demo Cases [src/app/nw_aligner] [src/app/splign/]

Computing pairwise global sequence alignments

Generic **pairwise** global alignment functionality is provided by CNWAligner.

NOTE: CNWAligner is not a multiple sequence aligner. An example of using CNWAligner can be seen here.

This functionality is discussed in the following topics:

- Initialization
- Parameters of alignment
- Computing
- Alignment transcript

Initialization

Two constructors are provided to initialize the aligner:

```
CNWAligner(const char* seq1, size_t len1,
 const char* seq2, size_t len2,
 const SNCBIPackedScoreMatrix* scoremat = 0);
CNWAligner(void);
```

The first constructor allows specification of the sequences and the score matrix at the time of the object's construction. Note that the sequences must be in the proper strands, because the aligners do not build reverse complementaries. The last parameter must be a pointer to a properly initialized SNCBIPackedScoreMatrix object or zero. If it is a valid pointer, then the sequences are verified against the alphabet contained in the SNCBIPackedScoreMatrix object, and its score matrix is further used in dynamic programming recurrences. Otherwise, sequences are verified against the IUPACna alphabet, and match/mismatch scores are used to fill in the score matrix.

The default constructor is provided to support reuse of an aligner object when many sequence pairs share the same type and alignment parameters. In this case, the following two functions must be called before computing the first alignment to load the score matrix and the sequences:

```
void SetScoreMatrix(const SNCBIPackedScoreMatrix* scoremat = 0);
void SetSequences(const char* seq1, size_t len1,
 const char* seq2, size_t len2,
 bool verify = true);
```

where the meaning of scoremat is the same as above.

Parameters of alignment

CNWAligner realizes the affine gap penalty model, which means that every gap of length L (with the possible exception of end gaps) contributes Wg+L*Ws to the total alignment score, where Wg is a cost to open the gap and Ws is a cost to extend the gap by one basepair. These two parameters are always in effect when computing sequence alignments and can be set with:

```
void SetWg(TScore value); // set gap opening score
void SetWs(TScore value); // set gap extension score
```

To indicate penalties, both gap opening and gap extension scores are assigned with negative values.

Many applications (such as the shotgun sequence assembly) benefit from a possibility to avoid penalizing end gaps of alignment, because the relevant sequence's ends may not be expected to align. CNWAligner supports this through a built-in end-space free variant controlled with a single function:

```
void SetEndSpaceFree(bool Left1, bool Right1, bool Left2, bool Right2);
```

The first two arguments control the left and the right ends of the first sequence. The other two control the second sequence's ends. True value means that end spaces will not be penalized. Although an arbitrary combination of end-space free flags can be specified, judgment should be used to get plausible alignments.

The following two functions are only meaningful when aligning nucleotide sequences:

```
void SetWm(TScore value); // set match score
void SetWms(TScore value); // set mismatch score
```

The first function sets a bonus associated with every matching pair of nucleotides. The second function assigns a penalty for every mismatching aligned pair of nucleotides. It is important that values set with these two functions will only take effect after SetScoreMatrix() is called (with a zero pointer, which is the default).

One thing that could limit the scope of global alignment applications is that the classical algorithm takes quadratic space and time to evaluate the alignment. One wayto deal with it is to use the linear-space algorithm encapuslated in CMMAligner. However, when some pattern of alignment is known or desired, it is worthwhile to explicitly specify "mile posts" through which the alignment should pass. Long high-scoring pairs with 100% identity (no gaps or mismatches) are typically good candidates for them. From the algorithmic point of view, the pattern splits the dynamic programming table into smaller parts, thus alleviating space and CPU requirements. The following function is provided to let the aligner know about such guiding constraints:

```
void SetPattern(const vector<size_t>& pattern);
```

Pattern is a vector of hits specified by their zero-based coordinates, as in the following example:

```
// the last parameter omitted to indicate nucl sequences
CNWAligner aligner (seq1, len1, seq2, len2);
// we want coordinates [99,119] and [129,159] on seq1 be aligned
// with [1099,1119] and [10099,10129] on seq2.
const size_t hits [] = { 99, 119, 1099, 1119, 129, 159, 10099, 10129 };
vector<size_t> pattern ( hits, hits + sizeof(hits)/sizeof(hits[0]) );
aligner.SetPattern(pattern);
```

Computing

To start computations, call Run(), which returns the overall alignment score having aligned the sequences. Score is a scalar value associated with the alignment and depends on the parameters of the alignment. The global alignment algorithms align two sequences so that the score is the maximum over all possible alignments.

Alignment transcript

The immediate output of the global alignment algorithms is a transcript.The transcript serves as a basic representation of alignments and is simply a string of elementary commands transforming the first sequence into the second one on a per-character basis. These commands (transcript characters) are (M)atch, (R)eplace, (I)nsert, and (D)elete. For example, the alignment

```
TTC-ATCTCTAAATCTCTCTCATATATATCG
||| |||||| |||| || ||| ||||
TTCGATCTCT-----TCTC-CAGATAAATCG
```

has a transcript:

```
MMMIMMMMMMDDDDDMMMMDMMRMMMRMMMM
```

Several functions are available to retrieve and analyze the transcript:

```
// raw transcript
const vector<ETranscriptSymbol>* GetTranscript(void) const
{
 return &m_Transcript;
}
// converted transcript vector
void GetTranscriptString(vector<char>* out) const;
// transcript parsers
size_t GetLeftSeg(size_t* q0, size_t* q1,
 size_t* s0, size_t* s1,
 size_t min_size) const;
size_t GetRightSeg(size_t* q0, size_t* q1,
 size_t* s0, size_t* s1,
 size_t min_size) const;
size_t GetLongestSeg(size_t* q0, size_t* q1,
 size_t* s0, size_t* s1) const;
```

The last three functions search for a continuous segment of matching characters and return it in sequence coordinates through q0, q1, s0, s1.

The alignment transcript is a simple yet complete representation of alignments that can be used to evaluate virtually every characteristic or detail of any particular alignment. Some of them, such as the percent identity or the number of gaps or mismatches, could be easily restored from the transcript alone, whereas others, such as the scores for protein alignments, would require availability of the original sequences.

Computing multiple sequence alignments

COBALT (COnstraint Based ALignment Tool) is an experimental multiple alignment algorithm whose basic idea was to leverage resources at NCBI, then build up a set of pairwise constraints, then perform a fairly standard iterative multiple alignment process (with many tweaks driven by various benchmarks).

COBALT is available online at:

https://www.ncbi.nlm.nih.gov/tools/cobalt/

A precompiled binary, with the data files needed to run it, is available at:

ftp://ftp.ncbi.nlm.nih.gov/pub/agarwala/cobalt/

Work is being done on an improved COBALT tool.

The paper reference for this algorithm is:

J.S. Papadopoulos, R. Agarwala, "COBALT: Constraint-Based Alignment Tool for Multiple Protein Sequences". Bioinformatics, May 2007

Aligning sequences in linear space

CMMAligner is an interface to a linear space variant of the global alignment algorithm. This functionality is discussed in the following topics:

- The idea of the algorithm
- Implementation

The idea of the algorithm

That the classical global alignment algorithm requires quadratic space could be a serious restriction in sequence alignment. One way to deal with it is to use alignment patterns. Another approach was first introduced by Hirschberg and became known as a divide-and-conquer strategy. At a coarse level, it suggests computing of scores for partial alignments starting from two opposite corners of the dynamic programming matrix while keeping only those located in the middle rows or columns. After the analysis of the adjacent scores, it is possible to determine cells on those lines through which the global alignment's back-trace path will go. This approach reduces space to linear while only doubling the worst-case time bound. For details see, for example, Dan Gusfield's "Algorithms on Strings, Trees and Sequences".

Implementation

CMMAligner inherits its public interface from CNWAligner. The only additional method allows us to toggle multiple-threaded versions of the algorithm.

The divide-and-conquer strategy suggests natural parallelization, where blocks of the dynamic programming matrix are evaluated simultaneously. A theoretical acceleration limit imposed by the current implementation is 0.5. To use multiple-threaded versions, call

EnableMultipleThreads(). The number of simultaneously running threads will not exceed the number of CPUs installed on your system.

When comparing alignments produced with the linear-space version with those produced by CNWAligner, be ready to find many of them similar, although not exactly the same. This is normal, because several optimal alignments may exist for each pair of sequences.

Computing spliced sequences alignments

This functionality is discussed in the following topics:

- The problem
- Implementation

The problem

The spliced sequence alignment arises as an attempt to address the problem of eukaryotic gene structure recognition. Tools based on spliced alignments exploit the idea of comparing genomic sequences to their transcribed and spliced products, such as mRNA, cDNA, or EST sequences. The final objective for all splice alignment algorithms is to come up with a combination of segments on the genomic sequence that:

- makes up a sequence very similar to the spliced product, when the segments are concatenated; and
- satisfies certain statistically determined conditions, such as consensus splice sites and lengths of introns.

According to the classical eukaryotic transcription and splicing mechanism, pieces of genomic sequence do not get shuffled. Therefore, one way of revealing the original exons could be to align the spliced product with its parent gene globally. However, because of the specificity of the process in which the spliced product is constructed, the generic global alignment with the affine penalty model may not be enough. To address this accurately, dynamic programming recurrences should specifically account for introns and splice signals.

Algorithms described in this chapter exploit this idea and address a refined splice alignment problem presuming that:

- the genomic sequence contains only one location from which the spliced product could have originated;
- the spliced product and the genomic sequence are in the plus strand; and
- the poly(A) tail and any other chunks of the product not created through the splicing were cut off, although a moderate level of sequencing errors on genomic, spliced, or both sequences is allowed.

In other words, the library classes provide basic splice alignment algorithms to be used in more sophisticated applications. One real-life application, Splign, can be found under demo cases for the library.

Implementation

There is a small hierarchy of three classes involved in spliced alignment facilitating a quality/performance trade-off in the case of distorted sequences:

- CSplicedAligner - abstract base for spliced aligners.
- CSplicedAligner16 - accounts for the three conventional splices (GT/AG, GC/AG, AT/AC) and a generic splice; uses 2 bytes per back-trace matrix cell. Use this class with high-quality genomic sequences.

- CSplicedAligner32 - accounts for the three conventionals and splices that could be produced by damaging bases of any conventional; uses 4 bytes per back-trace matrix cell. Use this class with distorted genomic sequences.

The abstract base class for spliced aligners, CNWSplicedAligner, inherites an interface from its parent, CNWAligner, adding support for two new parameters: intron penalty and minimal intron size (the default is 50).

All classes assume that the spliced sequence is the first of the two input sequences passed. By default, the classes do not penalize gaps at the ends of the spliced sequence. The default intron penalties are chosen so that the 16-bit version is able able to pick out short exons, whereas the 32-bit version is generally more conservative.

As with the generic global alignment, the immediate output of the algorithms is the alignment transcript. For the sake of spliced alignments, the transcript's alphabet is augmented to accommodate introns as a special sequence-editing operation.

Formatting computed alignments

This functionality is discussed in the following topics:

- Formatter object

Formatter object

CNWFormatter is a single place where all different alignment representations are created. The only argument to its constructor is the aligner object that actually was or will be used to align the sequences.

The alignment must be computed before formatting. If the formatter is unable to find the computed alignment in the aligner that was referenced to the constructor, an exception will be thrown.

To format the alignment as a CSeq_align structure, call

```
void AsSeqAlign(CSeq_align* output) const;
```

To format it as text, call

```
void AsText(string* output, ETextFormatType type, size_t line_width = 100)
```

Supported text formats and their ETextFormatType constants follow:

- Type 1 (eFormatType1):
  ```
  TTC-ATCTCTAAATCTCTCTCATATATATCG
  TTCGATCTCT-----TCTC-CAGATAAATCG
           ^ ^
  ```
- Type 2 (eFormatType2):
  ```
  TTC-ATCTCTAAATCTCTCTCATATATATCG
  ||| ||||||     |||| || ||| ||||
  TTCGATCTCT-----TCTC-CAGATAAATCG
  ```
- Gapped FastA (eFormatFastA):
  ```
  >SEQ1
  TTC-ATCTCTAAATCTCTCTCATATATATCG
  ```

```
>SEQ2
TTCGATCTCT-----TCTC-CAGATAAATCG
```

- Table of exons (eFormatExonTable) - spliced alignments only. The exons are listed from left to right in tab-separated columns. The columns represent sequence IDs, alignment lengths, percent identity, coordinates on the query (spliced) and the subject sequences, and a short annotation including splice signals.
- Extended table of exons (eFormatExonTableEx) - spliced alignments only. In addition to the nine columns, the full alignment transcript is listed for every exon.
- ASN.1 (eFormatASN)