The *Biostrings* 2 classes (work in progress)

Hervé Pagès

October 3, 2009

Contents

1 Introduction 1
2 The *XString* class and its subsetting operator [ 2
3 The == binary operator for *XString* objects 3
4 The *XStringViews* class and its subsetting operators [ and [[ 4
5 A few more *XStringViews* objects 5
6 The == binary operator for *XStringViews* objects 6
7 The start, end and width methods 6

1 Introduction

This document briefly presents the new set of classes implemented in the *Biostrings* 2 package. Like the *Biostrings* 1 classes (found in *Biostrings* v 1.4.x), they were designed to make manipulation of big strings (like DNA or RNA sequences) easy and fast. This is achieved by keeping the 3 following ideas from the *Biostrings* 1 package: (1) use R external pointers to store the string data, (2) use bit patterns to encode the string data, (3) provide the user with a convenient class of objects where each instance can store a set of views on the same big string (these views being typically the matches returned by a search algorithm).

However, there is a flaw in the *BioString* class design that prevents the search algorithms to return correct information about the matches (i.e. the views) that they found. The new classes address this issue by replacing the *BioString* class (implemented in *Biostrings* 1) by 2 new classes: (1) the *XString* class used to represent a single string, and (2) the *XStringViews* class used to represent a set of views on the same *XString* object, and by introducing new implementations and new interfaces for these 2 classes.
2 The XString class and its subsetting operator

The XString is in fact a virtual class and therefore cannot be instanciated. Only subclasses (or subtypes) BString, DNAString, RNAString and AAString can. These classes are direct extensions of the XString class (no additional slot).

A first BString object:

```r
> library(Biostrings)
> b <- BString("I am a BString object")
> b

21-letter "BString" instance
seq: I am a BString object

> length(b)
[1] 21
```

A DNAString object:

```r
> d <- DNAString("TTGAAAA-CTC-N")
> d

13-letter "DNAString" instance
seq: TTGAAAA-CTC-N

> length(d)
[1] 13
```

The differences with a BString object are: (1) only letters from the IUPAC extended genetic alphabet + the gap letter (-) are allowed and (2) each letter in the argument passed to the DNAString function is encoded in a special way before it’s stored in the DNAString object.

Access to the individual letters:

```r
> d[3]

1-letter "DNAString" instance
seq: G

> d[7:12]

6-letter "DNAString" instance
seq: A-CTC-

> d[]

13-letter "DNAString" instance
seq: TTGAAAA-CTC-N
```
21-letter "BString" instance
 seq: tcejbo gnirtSB a ma I

Only in bounds positive numeric subscripts are supported.

In fact the subsetting operator for XString objects is not efficient and one should always use the subseq method to extract a substring from a big string:

```r
> bb <- subseq(b, 3, 6)
> dd1 <- subseq(d, end = 7)
> dd2 <- subseq(d, start = 8)
```

To dump an XString object as a character vector (of length 1), use the toString method:

```r
> toString(dd2)
[1] "-CTC-N"
```

Note that length(dd2) is equivalent to nchar(toString(dd2)) but the latter would be very inefficient on a big DNAString object.

[TODO: Make a generic of the substr() function to work with XString objects. It will be essentially doing toString(subseq)).]

### 3 The == binary operator for XString objects

The 2 following comparisons are TRUE:

```r
> bb == "am a"
> dd2 != DNAString("TG")
```

When the 2 sides of == don’t belong to the same class then the side belonging to the “lowest” class is first converted to an object belonging to the class of the other side (the “highest” class). The class (pseudo-)order is character < BString < DNAString. When both sides are XString objects of the same subtype (e.g. both are DNAString objects) then the comparison is very fast because it only has to call the C standard function memcmp() and no memory allocation or string encoding/decoding is required.

The 2 following expressions provoke an error because the right member can’t be “upgraded” (converted) to an object of the same class than the left member:

```r
> bb == ""
> d == bb
```

When comparing an RNAString object with a DNAString object, U and T are considered equals:

```r
> r <- RNAString(d)
> r
```
An `XStringViews` object contains a set of views on the same `XString` object called the subject string. Here is an `XStringViews` object with 4 views:

```r
> v4 <- Views(dd2, start = 3:0, end = 5:8)
> v4

Views on a 6-letter DNAString subject
subject: -CTC-N
views:
  start end width
  [1]  3  5  3 [TC-]
  [2]  2  6  5 [CTC-N]
  [3]  1  7  7 [-CTC-N]
  [4]  0  8  9 [-CTC-N]

> length(v4)

[1] 4
```

Note that the 2 last views are out of limits.

You can select a subset of views from an `XStringViews` object:

```r
> v4[4:2]

Views on a 6-letter DNAString subject
subject: -CTC-N
views:
  start end width
  [1]  0  8  9 [-CTC-N]
  [2]  1  7  7 [-CTC-N]
  [3]  2  6  5 [CTC-N]
```

The returned object is still an `XStringViews` object, even if we select only one element. You need to use double-brackets to extract a given view as an `XString` object:

```r
> v4[[2]]

5-letter "DNAString" instance
seq: CTC-N
```
You can’t extract a view that is out of limits:

```r
> v4[[3]]
Error in v4[[3]] : view is out of limits
```

Note that, when start and end are numeric vectors and i is a single integer, Views(b, start, end)[[i]] is equivalent to subseq(b, start[i], end[i]).

Subsetting also works with negative or logical values with the expected semantic (the same as for R built-in vectors):

```r
> v4[-3]
```

Views on a 6-letter DNAString subject
subject: -CTC-N
views:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>3</td>
<td>[TC-]</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>5</td>
<td>[CTC-N]</td>
</tr>
<tr>
<td>0</td>
<td>8</td>
<td>9</td>
<td>[-CTC-N ]</td>
</tr>
</tbody>
</table>

```r
> v4[c(TRUE, FALSE)]
```

Views on a 6-letter DNAString subject
subject: -CTC-N
views:

<table>
<thead>
<tr>
<th>start</th>
<th>end</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Note that the logical vector is recycled to the length of v4.

## 5 A few more XStringViews objects

12 views (all of the same width):

```r
> v12 <- Views(DNAString("TAATAATG"), start = -2:9, end = 0:11)
```

This is the same as doing Views(d, start=1, end=length(d)):

```r
> as(d, "Views")
```

Hence the following will always return the d object itself:

```r
> as(d, "Views")[[1]]
```

3 XStringViews objects with no view:

```r
> v12[0]
> v12[FALSE]
> Views(d)
```
6 The \( \texttt{==} \) binary operator for \textit{XStringViews} objects

This operator is the vectorized version of the \( \texttt{==} \) operator defined previously for \textit{XString} objects:

\begin{verbatim}
> v12 == DNAString("TAA")
[1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
[12] FALSE
\end{verbatim}

To display all the views in \( v12 \) that are equals to a given view, you can type R cuties like:

\begin{verbatim}
> v12[v12 == v12[4]]

Views on a 8-letter DNAString subject
subject: TAATAATG
views:
  start end width
[1]  1  3  3 [TAA]
[2]  4  6  3 [TAA]

> v12[v12 == v12[1]]

Views on a 8-letter DNAString subject
subject: TAATAATG
views:
  start end width
[1] -2  0  3 [  ]
[2]  9 11  3 [  ]
\end{verbatim}

This is \texttt{TRUE}:

\begin{verbatim}
> v12[3] == Views(RNAString("AU"), start = 0, end = 2)
\end{verbatim}

7 The \texttt{start}, \texttt{end} and \texttt{width} methods

\begin{verbatim}
> start(v4)
[1]  3  2  1  0

> end(v4)
[1]  5  6  7  8

> width(v4)
[1]  3  5  7  9
\end{verbatim}

Note that \texttt{start(v4)[i]} is equivalent to \texttt{start(v4[i])}, except that the former will not issue an error if \( i \) is out of bounds (same for \texttt{end} and \texttt{width} methods).

Also, when \( i \) is a \texttt{single} integer, \texttt{width(v4)[i]} is equivalent to \texttt{length(v4[[i]])} except that the former will not issue an error if \( i \) is out of bounds or if view \( v4[i] \) is \textit{out of limits}. 

6