The Biostrings 2 classes (work in progress)

Hervé Pagès

November 7, 2006

Contents

1 Introduction 1
2 The BString class and its subsetting operator [ 1
3 The == binary operator for BString objects 3
4 The BStringViews class and its subsetting operators [ and [[ 4
5 A few more BStringViews objects 5
6 The == binary operator for BStringViews objects 6
7 The first, last 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 BString class used to represent a single string, and (2) the BStringViews class used to represent a set of views on the same BString object, and by introducing new implementations and new interfaces for these 2 classes.

2 The BString class and its subsetting operator [ 

A first BString object:
> library(Biostrings)
> b <- BString("I am a BString object")
> b

21-letter "BString" object
Value: I am a BString object

> length(b)
[1] 21

The DNAString and RNAString classes are direct extensions of the BString class (no additional slot):

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

13-letter "DNAString" object
Value: 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:

> d[3]

1-letter "DNAString" object
Value: G

> d[7:12]

6-letter "DNAString" object
Value: A-CTC-

> d[]

13-letter "DNAString" object
Value: TTGAAAA-CTC-N

> b[length(b):1]

21-letter "BString" object
Value: tcejbo gnirtSB a ma I
Only *in bounds* positive numeric subscripts are supported. In fact the subsetting operator for `BString` (or `DNASTring`) objects is not efficient and one should always use the `subBString` method to extract a substring from a big string:

```r
> bb <- subBString(b, 3, 6)
> dd1 <- subBString(d, last = 7)
> dd2 <- subBString(d, first = 8)
```

To *dump* a `BString` (or `DNASTring`) 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 `BString` objects. It will be essentially doing `toString(subBString())`.]

### 3 The `==` binary operator for `BString` objects

The 4 following expressions are **TRUE**:

```r
> bb == "am a"
> dd2 != DNASTring("TG")
> "-CTC-N" == dd2
> DNASTring("AC") == BString("AC")
```

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 `BString` (or `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 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
```

A `DNASTring` object and a `RNAString` object are considered equals when they are complementary:

```r
> dAlphabet <- DNASTring("ACGT-MRSVWYHKDBN")
> rAlphabet <- RNAString("UGCA-KYSBWRDMHVN")
> dAlphabet == rAlphabet
[1] TRUE
```

Note that this behaviour is consistent with the fact that `RNAString(dAlphabet)` is equal to `rAlphabet` and `DNASTring(rAlphabet)` is equal to `dAlphabet`. 
4 The BStringViews class and its subsetting operators [ and [[]

A BStringViews object contains a set of views on the same BString (or DNAString, or RNAString) object called the subject string. Here is a BStringViews object with 4 views:

> v4 <- views(dd2, 3:0, 5:8)
> v4

Views on a 6-letter DNAString subject
Subject: -CTC-N
Views:
  first last 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 a BStringViews object:

> v4[4:2]

Views on a 6-letter DNAString subject
Subject: -CTC-N
Views:
  first last width
[1]  0  8  9 [ -CTC-N ]
[2]  1  7  7 [-CTC-N ]
[3]  2  6  5 [CTC-N]

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

> v4[[2]]

5-letter "DNAString" object
Value: CTC-N

You can’t extract a view that is out of limits:

> v4[[3]]

Error in v4[[3]] : view is out of limits
Note that, when first and last are numeric vectors and i is a single integer, views(b, first, last)[i] is equivalent to subBString(b, first[i], last[i]).

The following is the same as doing subject(v4) (provided as a convenience):

```r
> v4[[0]]
```

6-letter "DNAString" object
Value: -CTC-N

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>owner</th>
<th>first</th>
<th>last</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>[2]</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>[3]</td>
<td>0</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>owner</th>
<th>first</th>
<th>last</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>[2]</td>
<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 BStringViews objects

12 views (all of the same width):

```r
> v12 <- views(DNAString("TAATAATG"), -2:9, 0:11)
```

This is the same as doing views(d, 1, length(d)):

```r
> views(d)
```

Hence the following will always return the d object itself:

```r
> views(d) [[1]]
```

3 BStringViews objects with no view:

```r
> v12[0]
> v12[FALSE]
> views(d, integer(0), integer(0))
```
6 The == binary operator for BStringViews objects

This operator is the vectorized version of the == operator defined previously for BString (or DNAString) objects:

```
> v12 == "TAA"
[1] FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
```

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

```
> v12[v12 == v12[4]]
```

Views on a 8-letter DNAString subject
Subject: TAATAATG
Views:
```
   first last 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:
```
   first last width
[1] -2  0  3 [ ]
[2]  9 11  3 [ ]
```

This is FALSE (whitespace matters):

```
> v12[3] == "TA"
```

and this provokes an error:

```
> v12[3] == " TA"
```

but these are TRUE:

```
> v12[3] == views(BString("TA"), 0, 2)
> v12[3] == views(RNAString("AU"), 0, 2)
```

7 The first, last and width methods

```
> first(v4)
[1] 3 2 1 0
```

```
> last(v4)
```
[1] 5 6 7 8

> width(v4)

[1] 3 5 7 9

Note that first(v4)[i] is equivalent to first(v4[i]), except that the former will not issue an error if i is out of bounds (same for last and width methods).

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