Efficient genome searching with Biostrings and the BSgenome data packages

Hervé Pagès

April 11, 2014

Contents

1 The Biostrings-based genome data packages

The Bioconductor project provides data packages that contain the full genome sequences of a given organism. These packages are called *Biostrings-based genome data packages* because the sequences they contain are stored in some of the basic containers defined in the Biostrings package, like the *DNAString*, the *DNAStringSet* or the *MaskedDNAString* containers. Regardless of the particular sequence data that they contain, all the Biostrings-based genome data packages are very similar and can be manipulated in a consistent and easy way. They all require the BSgenome package in order to work properly. This package, unlike the Biostrings-based genome data packages, is a software package that provides the infrastructure needed to support them (this is why the Biostrings-based genome data packages are also called *BSgenome data packages*). The BSgenome package itself requires the Biostrings package.

See the man page for the `available.genomes` function (`?available.genomes`) for more information about how to get the list of all the BSgenome data packages currently available in your version of Bioconductor (you need an internet connection so that `available.genomes` can query the Bioconductor package repositories).

More genomes can be added if necessary. Note that the process of making a BSgenome data package is not yet documented but you are welcome to ask for help on the bioc-devel mailing list (`http://bioconductor.org/docs/mailList.html`) if you need a genome that is not yet available.

2 Finding an arbitrary nucleotide pattern in a chromosome

In this section we show how to find (or just count) the occurences of some arbitrary nucleotide pattern in a chromosome. The basic tool for this is the `matchPattern` (or `countPattern`) function from the Biostrings package.

First we need to install and load the BSgenome data package for the organism that we want to look at. In our case, we want to search chromosome I of *Caenorhabditis elegans*.

UCSC provides several versions of the C. elegans genome: ce1, ce2 and ce4. These versions correspond to different *releases* from WormBase, which are the WS100, WS120 and WS170 releases, respectively. See `http://genome.ucsc.edu/FAQ/FAQreleases#release1` for the list of all UCSC genome releases and for the correspondance between UCSC versions and release names.

The BSgenome data package for the ce2 genome is BSgenome.Celegans.UCSC.ce2. Note that ce1 and ce4 are not available in Bioconductor but they could be added if there is demand for them.

See `?available.genomes` for how to install BSgenome.Celegans.UCSC.ce2. Then load the package and display the single object defined in it:

```
> library(BSgenome.Celegans.UCSC.ce2)
> ls("package:BSgenome.Celegans.UCSC.ce2")

[1] "BSgenome.Celegans.UCSC.ce2" "Celegans"

> genome <- BSgenome.Celegans.UCSC.ce2
> genome

Worm genome
|
| organism: Caenorhabditis elegans (Worm)
| provider: UCSC
| provider version: ce2
| release date: Mar. 2004
| release name: WormBase v. WS120
|
| single sequences (see '?seqnames'):
|   chrI    chrII   chrIII  chrIV   chrV    chrX    chrM
|
| multiple sequences (see '?mseqnames'):
|   upstream1000  upstream2000  upstream5000
|
| (use the '$' or '[[' operator to access a given sequence)
```

`genome` is a *BSgenome* object:

```
> class(genome)

[1] "BSgenome"
attr(,"package")
[1] "BSgenome"
```

When displayed, some basic information about the origin of the genome is shown (organism, provider, provider version, etc...) followed by the index of *single* sequences and eventually an additional index of *multiple* sequences. Methods (adequately called *accessor methods*) are defined for individual access to this information:

```
> organism(genome)
```

```
[1] "Caenorhabditis elegans"
```

```
> provider(genome)
```

```
[1] "UCSC"
```

```
> providerVersion(genome)
```

```
[1] "ce2"
```

```
> seqnames(genome)
```

```
[1] "chrI"   "chrII"  "chrIII" "chrIV"  "chrV"   "chrX"   "chrM"
```

```
> mseqnames(genome)
```

```
[1] "upstream1000" "upstream2000" "upstream5000"
```

See the man page for the *BSgenome* class (`?BSgenome`) for a complete list of accessor methods and their descriptions.

Now we are ready to display chromosome I:

```
> genome$chrI
```

```
  15080483-letter "DNAString" instance
seq: GCCTAAGCCTAAGCCTAAGCCTAAGCCTAAGCCTAA...TTAGGCTTAGGCTTAGGCTTAGGTTTAGGCTTAGGC
```

Note that this chrI sequence corresponds to the *forward* strand (aka *direct* or *sense* or *positive* or *plus* strand) of chromosome I. UCSC, and genome providers in general, don't provide files containing the nucleotide sequence of the *reverse* strand (aka *indirect* or *antisense* or *negative* or *minus* or *opposite* strand) of the chromosomes because these sequences can be deduced from the *forward* sequences by taking their reverse complements. The BSgenome data packages are no exceptions: they only provide the *forward* strand sequence of every chromosome. See `?reverseComplement` for more details about the reverse complement of a *DNAString* object. It is important to remember that, in practice, the *reverse* strand sequence is almost never needed. The reason is that, in fact, a *reverse* strand analysis can (and should) always be transposed into a *forward* strand analysis. Therefore trying to compute the *reverse* strand sequence of an entire chromosome by applying `reverseComplement` to its *forward* strand sequence is almost always a bad idea. See the *Finding an arbitrary nucleotide pattern in an entire genome* section of this document for how to find arbitrary patterns in the *reverse* strand of a chromosome.

The number of bases in this sequence can be retrieved with:

```
> chrI <- genome$chrI
> length(chrI)
```

```
[1] 15080483
```

Some basic stats:

```
> afI <- alphabetFrequency(chrI)
> afI
```

```
      A       C       G       T       M       R       W       S       Y       K
4838561 2697177 2693544 4851201       0       0       0       0       0       0
      V       H       D       B       N       -       +       .
      0       0       0       0       0       0       0       0
```

```
> sum(afI) == length(chrI)

[1] TRUE
```

Count all *exact* matches of pattern `"ACCCAGGGC"`:

```
> p1 <- "ACCCAGGGC"
> countPattern(p1, chrI)

[1] 0
```

Like most pattern matching functions in Biostrings, the `countPattern` and `matchPattern` functions support *inexact* matching. One form of inexact matching is to allow a few mismatching letters per match. Here we allow at most one:

```
> countPattern(p1, chrI, max.mismatch=1)

[1] 235
```

With the `matchPattern` function, the locations of the matches are stored in an *XStringViews* object:

```
> m1 <- matchPattern(p1, chrI, max.mismatch=1)
> m1[4:6]

  Views on a 15080483-letter DNAString subject
subject: GCCTAAGCCTAAGCCTAAGCCTAAGCCTAAGCCT...AGGCTTAGGCTTAGGCTTAGGTTTAGGCTTAGGC
views:
     start    end width
[1] 187350 187358     9 [ACCCAAGGC]
[2] 213236 213244     9 [ACCCAGGGG]
[3] 424133 424141     9 [ACCCAGGAC]

> class(m1)

[1] "XStringViews"
attr(,"package")
[1] "Biostrings"
```

The `mismatch` function (new in Biostrings 2) returns the positions of the mismatching letters for each match:

```
> mismatch(p1, m1[4:6])

[[1]]
[1] 6

[[2]]
[1] 9

[[3]]
[1] 8
```

Note: The `mismatch` method is in fact a particular case of a (vectorized) *alignment* function where only "replacements" are allowed. Current implementation is slow but this will be addressed.

It may happen that a match is *out of limits* like in this example:

```
> p2 <- DNAString("AAGCCTAAGCCTAAGCCTAA")
> m2 <- matchPattern(p2, chrI, max.mismatch=2)
> m2[1:4]

  Views on a 15080483-letter DNAString subject
subject: GCCTAAGCCTAAGCCTAAGCCTAAGCCT...AGGCTTAGGCTTAGGCTTAGGTTTAGGCTTAGGC
views:
    start end width
[1]    -1  18    20 [  GCCTAAGCCTAAGCCTAA]
[2]     5  24    20 [AAGCCTAAGCCTAAGCCTAA]
[3]    11  30    20 [AAGCCTAAGCCTAAGCCTAA]
[4]    17  36    20 [AAGCCTAAGCCTAAGCCTAA]

> p2 == m2[1:4]

[1] FALSE  TRUE  TRUE  TRUE

> mismatch(p2, m2[1:4])

[[1]]
[1] 1 2

[[2]]
integer(0)

[[3]]
integer(0)

[[4]]
integer(0)
```

The list of exact matches and the list of inexact matches can both be obtained with:

```
> m2[p2 == m2]
> m2[p2 != m2]
```

Note that the length of `m2[p2 == m2]` should be equal to `countPattern(p2, chrI, max.mismatch=0)`.

3 Finding an arbitrary nucleotide pattern in an entire genome

Now we want to extend our analysis to the *forward* and *reverse* strands of all the C. elegans chromosomes. More precisely, here is the analysis we want to perform:

- The input dictionary: Our input is a dictionary of 50 patterns. Each pattern is a short nucleotide sequence of 15 to 25 bases (As, Cs, Gs and Ts only, no Ns). It is stored in a FASTA file called `"ce2dict0.fa"`. See the *Finding all the patterns of a constant width dictionary in an entire genome* section of this document for a very efficient way to deal with the special case where all the patterns in the input dictionary have the same length.
- The target: Our target (or subject) is the *forward* and *reverse* strands of the seven C. elegans chromosomes (14 sequences in total). We want to find and report all occurences (or hits) of every pattern in the target. Note that a given pattern can have 0, 1 or several hits in 0, 1 or 2 strands of 0, 1 or several chromosomes.

- Exact or inexact matching? We are interested in exact matches only (for now).
- The output: We want to put the results of this analysis in a file so we can send it to our collaborators for some post analysis work. Our collaborators are not necessarily familiar with R or Bioconductor so dumping a high-level R object (like a list or a data frame) into an .rda file is not an option. For maximum portability (one of our collaborators wants to use Microsoft Excel for the post analysis) we choose to put our results in a tabulated file where one line describes one hit. The columns (or fields) of this file will be (in this order):
  - seqname: the name of the chromosome where the hit occurs.
  - start: an integer giving the starting position of the hit.
  - end: an integer giving the ending position of the hit.
  - strand: a plus (`+`) for a hit in the positive strand or a minus (`-`) for a hit in the negative strand.
  - patternID: we use the unique ID provided for every pattern in the `"ce2dict0.fa"` file.

Let's start by loading the input dictionary with:

```
> ce2dict0_file <- system.file("extdata", "ce2dict0.fa", package="BSgenome")
> ce2dict0 <- readDNAStringSet(ce2dict0_file, "fasta")
> ce2dict0

  A DNAStringSet instance of length 50
     width seq                                               names
 [1]    18 GCGAAACTAGGAGAGGCT                                pattern01
 [2]    25 CTGTTAGCTAATTTTAAAAATAAAT                         pattern02
 [3]    24 ACTACCACCCAAATTTAGATATTC                          pattern03
 [4]    24 AAATTTTTTTTGTTGCAAATTTGA                          pattern04
 [5]    25 TCTTCTTGGCTTTGGTGGTACTTTT                         pattern05
 ...   ... ...
[46]    24 TTTTGAACAAAGCATGTCTAACTA                          pattern46
[47]    20 TAAACGAATTTAGGATATAT                              pattern47
[48]    19 AAGGACCAGGATTGGCACG                               pattern48
[49]    24 AAATAACTGCGTAAAAACACAATA                          pattern49
[50]    22 AAAATGCCGGAGCATTTTAAAG                            pattern50
```

Here is how we can write the functions that will perform our analysis:

```
> writeHits <- function(seqname, matches, strand, file="", append=FALSE)
+ {
+     if (file.exists(file) && !append)
+         warning("existing file ", file, " will be overwritten with 'append=FALSE'")
+     if (!file.exists(file) && append)
+         warning("new file ", file, " will have no header with 'append=TRUE'")
+     hits <- data.frame(seqname=rep.int(seqname, length(matches)),
+                        start=start(matches),
+                        end=end(matches),
+                        strand=rep.int(strand, length(matches)),
+                        patternID=names(matches),
+                        check.names=FALSE)
+     write.table(hits, file=file, append=append, quote=FALSE, sep="\t",
+                 row.names=FALSE, col.names=!append)
+ }
```

```
> runAnalysis1 <- function(dict0, outfile="")
+ {
+     library(BSgenome.Celegans.UCSC.ce2)
+     genome <- BSgenome.Celegans.UCSC.ce2
+     seqnames <- seqnames(genome)
+     seqnames_in1string <- paste(seqnames, collapse=", ")
+     cat("Target:", providerVersion(genome),
+         "chromosomes", seqnames_in1string, "\n")
+     append <- FALSE
+     for (seqname in seqnames) {
+         subject <- genome[[seqname]]
+         cat(">>> Finding all hits in chromosome", seqname, "...\n")
+         for (i in seq_len(length(dict0))) {
+             patternID <- names(dict0)[i]
+             pattern <- dict0[[i]]
+             plus_matches <- matchPattern(pattern, subject)
+             names(plus_matches) <- rep.int(patternID, length(plus_matches))
+             writeHits(seqname, plus_matches, "+", file=outfile, append=append)
+             append <- TRUE
+             rcpattern <- reverseComplement(pattern)
+             minus_matches <- matchPattern(rcpattern, subject)
+             names(minus_matches) <- rep.int(patternID, length(minus_matches))
+             writeHits(seqname, minus_matches, "-", file=outfile, append=append)
+         }
+         cat(">>> DONE\n")
+     }
+ }
```

Some important notes about the implementation of the `runAnalysis1` function:

- `subject <- genome[[seqname]]` is the code that actually loads a chromosome sequence into memory. Using only one sequence at a time is a good practice to avoid memory allocation problems on a machine with a limited amount of memory. For example, loading all the human chromosome sequences in memory would require more than 3GB of memory!
- We have 2 nested `for` loops: the outer loop walks thru the target (7 chromosomes) and the inner loop walks thru the set of patterns. Doing the other way around would be very inefficient, especially with a bigger number of patterns because this would require to load each chromosome sequence into memory as many times as the number of patterns. `runAnalysis1` loads each sequence only once.
- We find the matches in the minus strand (`minus_matches`) by first taking the reverse complement of the current pattern (with `rcpattern <- reverseComplement(pattern)`) and NOT by taking the reverse complement of the current subject.

Now we are ready to run the analysis and put the results in the `"ce2dict0_ana1.txt"` file:

```
> runAnalysis1(ce2dict0, outfile="ce2dict0_ana1.txt")

Target: ce2 chromosomes chrI, chrII, chrIII, chrIV, chrV, chrX, chrM
>>> Finding all hits in chromosome chrI ...
>>> DONE
>>> Finding all hits in chromosome chrII ...
>>> DONE
>>> Finding all hits in chromosome chrIII ...
```

```
>>> DONE
>>> Finding all hits in chromosome chrIV ...
>>> DONE
>>> Finding all hits in chromosome chrV ...
>>> DONE
>>> Finding all hits in chromosome chrX ...
>>> DONE
>>> Finding all hits in chromosome chrM ...
>>> DONE
```

Here is some very simple example of post analysis:

- Get the total number of hits:

```
> hits1 <- read.table("ce2dict0_ana1.txt", header=TRUE)
> nrow(hits1)

[1] 79
```

- Get the number of hits per chromosome:

```
> table(hits1$seqname)

  chrI  chrII chrIII  chrIV   chrM   chrV   chrX
    11      5     16      8      8     15     16
```

- Get the number of hits per pattern:

```
> hits1_table <- table(hits1$patternID)
> hits1_table

pattern01 pattern02 pattern03 pattern04 pattern06 pattern07 pattern08 pattern09
        1         1         1         1         2         1         1         1
pattern10 pattern11 pattern12 pattern13 pattern14 pattern15 pattern16 pattern17
        1         1         1         1         1         1         1         1
pattern18 pattern19 pattern20 pattern21 pattern22 pattern23 pattern24 pattern25
        1         9         1        10         2         1         1         1
pattern26 pattern27 pattern28 pattern29 pattern30 pattern31 pattern32 pattern33
        1         1         1         1         1         1         1         1
pattern34 pattern35 pattern36 pattern37 pattern38 pattern39 pattern40 pattern41
        2         1         1         7         1         1         1         1
pattern42 pattern43 pattern44 pattern45 pattern46 pattern47 pattern48 pattern49
        1         5         1         1         1         1         1         1
pattern50
        1
```

- Get the pattern(s) with the higher number of hits:

```
> hits1_table[hits1_table == max(hits1_table)] # pattern(s) with more hits

pattern21
       10
```

- Get the pattern(s) with no hits:

```
> setdiff(names(ce2dict0), hits1$patternID) # pattern(s) with no hits

[1] "pattern05"
```

- And finally a function that can be used to plot the hits:

```
> plotGenomeHits <- function(bsgenome, seqnames, hits)
+ {
+     chrlengths <- seqlengths(bsgenome)[seqnames]
+     XMAX <- max(chrlengths)
+     YMAX <- length(seqnames)
+     plot.new()
+     plot.window(c(1, XMAX), c(0, YMAX))
+     axis(1)
+     axis(2, at=seq_len(length(seqnames)), labels=rev(seqnames), tick=FALSE, las=1)
+     ## Plot the chromosomes
+     for (i in seq_len(length(seqnames)))
+         lines(c(1, chrlengths[i]), c(YMAX + 1 - i, YMAX + 1 - i), type="l")
+     ## Plot the hits
+     for (i in seq_len(nrow(hits))) {
+         seqname <- hits$seqname[i]
+         y0 <- YMAX + 1 - match(seqname, seqnames)
+         if (hits$strand[i] == "+") {
+             y <- y0 + 0.05
+             col <- "red"
+         } else {
+             y <- y0 - 0.05
+             col <- "blue"
+         }
+         lines(c(hits$start[i], hits$end[i]), c(y, y), type="l", col=col, lwd=3)
+     }
+ }
```

Plot the hits found by `runAnalysis1` with:

```
> plotGenomeHits(genome, seqnames(genome), hits1)
```

4 Some precautions when using *matchPattern*

Improper use of *matchPattern* (or *countPattern*) can affect performance.

If needed, the *matchPattern* and *countPattern* methods convert their first argument (the pattern) to an object of the same class than their second argument (the subject) before they pass it to the subroutine that actually implements the fast search algorithm.

So if you need to reuse the same pattern a high number of times, it's a good idea to convert it *before* to pass it to the *matchPattern* or *countPattern* method. This way the conversion is done only once:

```
> library(hgu95av2probe)
> tmpseq <- DNAStringSet(hgu95av2probe$sequence)
> someStats <- function(v)
+ {
+     GC <- DNAString("GC")
+     CG <- DNAString("CG")
```

```
+     sapply(seq_len(length(v)),
+            function(i) {
+                y <- v[[i]]
+                c(alphabetFrequency(y)[1:4],
+                  GC=countPattern(GC, y),
+                  CG=countPattern(CG, y))
+            }
+     )
+ }
> someStats(tmpseq[1:10])

   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
A     1    5    6    4    4    2    4    5    9     2
C    10    5    4    7    5    7   10    8    7    10
G     6    5    3    8    8    6    4    5    4     4
T     8   10   12    6    8   10    7    7    5     9
GC    2    1    1    4    3    2    2    2    1     1
CG    0    0    0    2    1    1    0    0    0     0
```

5 Masking the chromosome sequences

Starting with Bioconductor 2.2, the chromosome sequences in a *BSgenome data package* can have built-in masks. Starting with Bioconductor 2.3, there can be up to 4 built-in masks per sequence. These will always be (in this order): (1) the mask of assembly gaps, (2) the mask of intra-contig ambiguities, (3) the mask of repeat regions that were determined by the RepeatMasker software, and (4) the mask of repeat regions that were determined by the Tandem Repeats Finder software (where only repeats with period less than or equal to 12 were kept).

For a given package, all the sequences will always have the same number of masks.

```
> library(BSgenome.Hsapiens.UCSC.hg19.masked)
> genome <- BSgenome.Hsapiens.UCSC.hg19.masked
> chrY <- genome$chrY
> chrY

  59373566-letter "MaskedDNAString" instance (# for masking)
seq: ####################################...####################################
masks:
  maskedwidth maskedratio active names                                desc
1    33720000 0.567929506   TRUE AGAPS                         assembly gaps
2           0 0.000000000   TRUE   AMB   intra-contig ambiguities (empty)
3    16024357 0.269890426  FALSE    RM                          RepeatMasker
4      587815 0.009900281  FALSE   TRF Tandem Repeats Finder [period<=12]
all masks together:
  maskedwidth maskedratio
     49783032   0.8384713
all active masks together:
  maskedwidth maskedratio
     33720000   0.5679295

> chrM <- genome$chrM
> chrM
```

```
  16571-letter "MaskedDNAString" instance (# for masking)
seq: GATCACAGGTCTATCACCCTATTAACCACTCACGGG...AGCCCACACGTTCCCCTTAAATAAGACATCACGATG
masks:
  maskedwidth maskedratio active names
1           0   0.0000000   TRUE AGAPS
2           0   0.0000000   TRUE   AMB
3         373   0.0225092  FALSE    RM
4           0   0.0000000  FALSE   TRF
                                      desc
1                     assembly gaps (empty)
2           intra-contig ambiguities (empty)
3                              RepeatMasker
4 Tandem Repeats Finder [period<=12] (empty)
all masks together:
  maskedwidth maskedratio
         373   0.0225092
all active masks together:
  maskedwidth maskedratio
           0           0
```

The built-in masks are named consistenly across all the BSgenome data packages available in Bioconductor:

Name	Active by default	Short description	Long description
AGAPS	yes	assembly gaps	Masks the big N-blocks that have been placed between the cont
AMB	yes	intra-contig ambiguities	Masks any IUPAC ambiguity letter that was found in the conti
RM	no	RepeatMasker	Masks the repeat regions determined by the RepeatMasker soft
TRF	no	Tandem Repeats Finder	Masks the tandem repeat regions that were determined by the

Table 1: The built-in masks provided by the BSgenome data packages.

When displaying a masked sequence (here a *MaskedDNAString* object), the *masked width* and *masked ratio* are reported for each individual mask, as well as for all the masks together, and for all the active masks together. The *masked width* is the total number of nucleotide positions that are masked and the *masked ratio* is the *masked width* divided by the length of the sequence.

To activate a mask, use the *active* replacement method in conjonction with the *masks* method. For example, to activate the RepeatMasker mask, do:

```
> active(masks(chrY))["RM"] <- TRUE
> chrY

  59373566-letter "MaskedDNAString" instance (# for masking)
seq: ####################################...####################################
masks:
  maskedwidth maskedratio active names                                desc
1    33720000 0.567929506   TRUE AGAPS                        assembly gaps
2           0 0.000000000   TRUE   AMB   intra-contig ambiguities (empty)
3    16024357 0.269890426   TRUE    RM                           RepeatMasker
4      587815 0.009900281  FALSE   TRF Tandem Repeats Finder [period<=12]
all masks together:
  maskedwidth maskedratio
```

```
    49783032   0.8384713
all active masks together:
  maskedwidth maskedratio
    49744357   0.8378199
```

As you can see, the *masked width* for all the active masks together (i.e. the total number of nucleotide positions that are masked by at least one active mask) is now the same as for the first mask. This represents a *masked ratio* of about 83%.

Now when we use a function that is *mask aware*, like `alphabetFrequency`, the masked regions of the input sequence are ignored:

```
> active(masks(chrY)) <- FALSE
> active(masks(chrY))["AGAPS"] <- TRUE
> alphabetFrequency(unmasked(chrY))

      A       C       G       T       M       R       W       S
 7667625 5099171 5153288 7733482       0       0       0       0
      Y       K       V       H       D       B       N       -
      0       0       0       0       0       0 33720000       0
      +       .
      0       0

> alphabetFrequency(chrY)

      A       C       G       T       M       R       W       S       Y       K
7667625 5099171 5153288 7733482       0       0       0       0       0       0
      V       H       D       B       N       -       +       .
      0       0       0       0       0       0       0       0
```

This output indicates that, for this chromosome, the assembly gaps correspond exactly to the regions in the sequence that were filled with the letter N. Note that this is not always the case: sometimes Ns, and other IUPAC ambiguity letters, can be found inside the contigs.

When coercing a *MaskedXString* object to an *XStringViews* object, each non-masked region in the original sequence is converted into a view on the sequence:

```
> as(chrY, "XStringViews")

  Views on a 59373566-letter DNAString subject
subject: NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
views:
        start      end   width
 [1]    10001    44821   34821 [CTAACCCTAACCCTAACCCTAA...TAGGTCTCATTGAGGACAGATA]
 [2]    94822   181384   86563 [GATCCACCCATCTCGGTCTCCC...TCGATCTCGTGACCTCGTGATC]
 [3]   231385   997557  766173 [GATCGGGGTATCCCAGCTGCTA...TGTGATCCAAGGGACTGAATTC]
 [4]  1047558  1084113   36556 [ATTAAAGAAGGAGAGAGACTGG...GGTGTGTGTGTGCATGCATGCT]
 [5]  1134114  1214234   80121 [GATTGAACCAGCCCACTCCACG...CCTGGCCAACATGGGGAAACCC]
 ...      ...      ...     ... ...
[13] 20193886 22369679 2175794 [GAATTCAGCATTTTCATGGAAT...GTTTCTGTCTTCTGGCAGGATC]
[14] 22419680 23901428 1481749 [GAATTCAGTGGTGAGTGTTACA...TTTCAAAAACTTTATGGAATTC]
[15] 23951429 28819361 4867933 [AAGCTTTGGCTAATATATCTCT...GGAGTGGTGCAGAGTGGAATTC]
[16] 58819362 58917656   98295 [GAATTCCATTCCATTCCAATCC...TCCCTTCCATTCCAATGAATTC]
[17] 58967657 59363566  395910 [GAATTCAACATTATTCTTGTTT...GGGTGTGGTGTGTGGGTGTGGT]
```

This can be used in conjonction with the *gaps* method to see the gaps between the views i.e. the masked regions themselves:

```
> gaps(as(chrY, "XStringViews"))
```

To extract the sizes of the assembly gaps:

```
> width(gaps(as(chrY, "XStringViews")))
```

```
 [1]    10000    50000    50000    50000    50000    50000    50000    50000
 [9]    50000  3000000    50000    50000    50000    50000    50000 30000000
[17]    50000    10000
```

Note that, if applied directly to `chrY`, *gaps* returns a *MaskedDNAString* object with a single mask masking the regions that are not masked in the original object:

```
> gaps(chrY)
```

```
  59373566-letter "MaskedDNAString" instance (# for masking)
seq: NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
masks:
  maskedwidth maskedratio active
1    25653566   0.4320705   TRUE
```

```
> alphabetFrequency(gaps(chrY))
```

```
      A       C       G       T       M       R       W       S
      0       0       0       0       0       0       0       0
      Y       K       V       H       D       B       N       -
      0       0       0       0       0       0 33720000       0
      +       .
      0       0
```

In fact, for any *MaskedDNAString* object, the following should always be `TRUE`, whatever the masks are:

```
> af0 <- alphabetFrequency(unmasked(chrY))
> af1 <- alphabetFrequency(chrY)
> af2 <- alphabetFrequency(gaps(chrY))
> all(af0 == af1 + af2)
```

```
[1] TRUE
```

With all chrY masks active:

```
> active(masks(chrY)) <- TRUE
> af1 <- alphabetFrequency(chrY)
> af1
```

```
      A       C       G       T       M       R       W       S       Y       K
2935249 1845512 1857214 2952559       0       0       0       0       0       0
      V       H       D       B       N       -       +       .
      0       0       0       0       0       0       0       0
```

```
> gaps(chrY)
```

```
  59373566-letter "MaskedDNAString" instance (# for masking)
seq: NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
masks:
  maskedwidth maskedratio active
1     9590534   0.1615287   TRUE
```

```
> af2 <- alphabetFrequency(gaps(chrY))
> af2
```

```
       A        C        G        T        M        R        W        S
 4732376  3253659  3296074  4780923        0        0        0        0
       Y        K        V        H        D        B        N        -
       0        0        0        0        0        0 33720000        0
       +        .
       0        0
```

```
> all(af0 == af1 + af2)
```

```
[1] TRUE
```

Now let's compare three different ways of finding all the occurences of the `"CANNTG"` consensus sequence in chrY. The Ns in this pattern need to be treated as wildcards i.e. they must match any letter in the subject.

Without the mask feature, the first way to do it would be to use the `fixed=FALSE` option in the call to `matchPattern` (or `countPattern`):

```
> Ebox <- "CANNTG"
> active(masks(chrY)) <- FALSE
> countPattern(Ebox, chrY, fixed=FALSE)
```

```
[1] 33858401
```

The problem with this method is that the Ns in the subject are also treated as wildcards hence the abnormally high number of matches. A better method is to specify the *side* of the matching problem (i.e. *pattern* or *subject*) where the Ns should be treated as wildcards:

```
> countPattern(Ebox, chrY, fixed=c(pattern=FALSE,subject=TRUE))
```

```
[1] 138466
```

Finally, `countPattern` being *mask aware*, this can be achieved more efficiently by just masking the assembly gaps and ambiguities:

```
> active(masks(chrY))[c("AGAPS", "AMB")] <- TRUE
> alphabetFrequency(chrY, baseOnly=TRUE)  # no ambiguities
```

```
      A       C       G       T   other
7667625 5099171 5153288 7733482       0
```

```
> countPattern(Ebox, chrY, fixed=FALSE)
```

```
[1] 138466
```

Note that some chromosomes can have Ns outside the assembly gaps:

```
> chr2 <- genome$chr2
> active(masks(chr2))[-2] <- FALSE
> alphabetFrequency(gaps(chr2))

   A    C    G    T    M    R    W    S    Y    K    V    H    D    B    N    -
   0    0    0    0    0    0    0    0    0    0    0    0    0    0 2855    0
   +    .
   0    0
```

so it is recommended to always keep the AMB mask active (in addition to the AGAPS mask) whatever the sequence is.

Note that not all functions that work with an *XString* input are *mask aware* but more will be added in the near future. However, most of the times there is a alternate way to exclude some arbitrary regions from an analysis without having to use *mask aware* functions. This is described below in the *Hard masking* section.

6 Hard masking

coming soon...

7 Injecting known SNPs in the chromosome sequences

coming soon...

8 Finding all the patterns of a constant width dictionary in an entire genome

The `matchPDict` function can be used instead of `matchPattern` for the kind of analysis described in the *Finding an arbitrary nucleotide pattern in an entire genome* section but it will be much faster (between 100x and 10000x faster depending on the size of the input dictionary). Note that a current limitation of `matchPDict` is that it only works with a dictionary of DNA patterns where all the patterns have the same number of nucleotides (constant width dictionary). See `?matchPDict` for more information.

Here is how our `runAnalysis1` function can be modified in order to use `matchPDict` instead of `matchPattern`:

```
> runOneStrandAnalysis <- function(dict0, bsgenome, seqnames, strand,
+                                  outfile="", append=FALSE)
+ {
+     cat("\nTarget: strand", strand, "of", providerVersion(bsgenome),
+         "chromosomes", paste(seqnames, collapse=", "), "\n")
+     if (strand == "-")
+         dict0 <- reverseComplement(dict0)
+     pdict <- PDict(dict0)
+     for (seqname in seqnames) {
+         subject <- bsgenome[[seqname]]
+         cat(">>> Finding all hits in strand", strand, "of chromosome", seqname, "...\n")
+         mindex <- matchPDict(pdict, subject)
+         matches <- extractAllMatches(subject, mindex)
+         writeHits(seqname, matches, strand, file=outfile, append=append)
+         append <- TRUE
```

```
+         cat(">>> DONE\n")
+     }
+ }
> runAnalysis2 <- function(dict0, outfile="")
+ {
+     library(BSgenome.Celegans.UCSC.ce2)
+     genome <- BSgenome.Celegans.UCSC.ce2
+     seqnames <- seqnames(genome)
+     runOneStrandAnalysis(dict0, genome, seqnames, "+", outfile=outfile, append=FALSE)
+     runOneStrandAnalysis(dict0, genome, seqnames, "-", outfile=outfile, append=TRUE)
+ }
```

Remember that `matchPDict` only works if all the patterns in the input dictionary have the same length so for this 2nd analysis, we will truncate the patterns in `ce2dict0` to 15 nucleotides:

```
> ce2dict0cw15 <- DNAStringSet(ce2dict0, end=15)
```

Now we can run this 2nd analysis and put the results in the `"ce2dict0cw15_ana2.txt"` file:

```
> runAnalysis2(ce2dict0cw15, outfile="ce2dict0cw15_ana2.txt")

Target: strand + of ce2 chromosomes chrI, chrII, chrIII, chrIV, chrV, chrX, chrM
>>> Finding all hits in strand + of chromosome chrI ...
>>> DONE
>>> Finding all hits in strand + of chromosome chrII ...
>>> DONE
>>> Finding all hits in strand + of chromosome chrIII ...
>>> DONE
>>> Finding all hits in strand + of chromosome chrIV ...
>>> DONE
>>> Finding all hits in strand + of chromosome chrV ...
>>> DONE
>>> Finding all hits in strand + of chromosome chrX ...
>>> DONE
>>> Finding all hits in strand + of chromosome chrM ...
>>> DONE

Target: strand - of ce2 chromosomes chrI, chrII, chrIII, chrIV, chrV, chrX, chrM
>>> Finding all hits in strand - of chromosome chrI ...
>>> DONE
>>> Finding all hits in strand - of chromosome chrII ...
>>> DONE
>>> Finding all hits in strand - of chromosome chrIII ...
>>> DONE
>>> Finding all hits in strand - of chromosome chrIV ...
>>> DONE
>>> Finding all hits in strand - of chromosome chrV ...
>>> DONE
>>> Finding all hits in strand - of chromosome chrX ...
>>> DONE
>>> Finding all hits in strand - of chromosome chrM ...
>>> DONE
```

9 Session info

```
> sessionInfo()

R version 3.1.0 (2014-04-10)
Platform: x86_64-unknown-linux-gnu (64-bit)

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] parallel  stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
 [1] BSgenome.Hsapiens.UCSC.hg19.masked_1.3.99
 [2] BSgenome.Hsapiens.UCSC.hg19_1.3.99
 [3] hgu95av2probe_2.14.0
 [4] AnnotationDbi_1.26.0
 [5] Biobase_2.24.0
 [6] BSgenome.Celegans.UCSC.ce2_1.3.99
 [7] BSgenome_1.32.0
 [8] GenomicRanges_1.16.0
 [9] GenomeInfoDb_1.0.0
[10] Biostrings_2.32.0
[11] XVector_0.4.0
[12] IRanges_1.21.45
[13] BiocGenerics_0.10.0

loaded via a namespace (and not attached):
[1] DBI_0.2-7        RSQLite_0.11.4   Rsamtools_1.16.0 bitops_1.0-6
[5] stats4_3.1.0     tools_3.1.0      zlibbioc_1.10.0
```