An Introduction to *IRanges*

Michael Lawrence, Patrick Aboyoun, Hervé Pagès

June 10, 2009

1 Introduction

The *IRanges* package is designed to represent sequences, ranges representing indices along those sequences, and data related to those ranges. In this vignette, we will rely on simple, illustrative example datasets, rather than large, real-world data, so that each data structure and algorithm can be explained in an intuitive, graphical manner. We expect that packages that apply *IRanges* to a particular problem domain will provide vignettes with relevant, realistic examples.

2 Sequences

The *IRanges* package is primarily interested in the representation, manipulation and analysis of sequences. A *sequence* is mathematically defined as an ordered list of objects, or elements. It differs from a *set* in that order matters and the same element can appear multiple times.

IRanges supports the use of R vectors to represent sequences, and we also formally define a virtual class *Sequence*, the derivatives of which convey the sequence semantic of ordered elements. There are currently two *Sequence* implementations in *IRanges*: *Rle*, which compresses a sequence through run-length encoding, and *XSequence*, which refers to its data through an external pointer. *XSequence* and its derivatives are considered low-level infrastructure and, as such, will not be covered by this vignette.

We begin our demonstration by loading the *IRanges* package.

```
> library(IRanges)
```

2.1 Run Length Encoding

The *Rle* class represents a run-length encoded (compressed) sequence of *logical*, *integer*, *numeric*, *complex*, *character*, or *raw* values.

```
> set.seed(0)
> lambda <- c(rep(0.001, 4500), seq(0.001, 10, length = 500),
+     seq(10, 0.001, length = 500))
> x <- Rle(rpois(1e+07, lambda))
> y <- Rle(rpois(1e+07, lambda[c(251:length(lambda),
+     1:250)]))
> x

  'numeric' Rle instance of length 10000000 with 1510219 runs
  Lengths:  780 1 208 1 1599 1 883 1 1038 1 ...
  Values :  0 1 0 1 0 1 0 1 0 1 ...

> y
```

```
  'numeric' Rle instance of length 10000000 with 1511351 runs
  Lengths:  1003 1 413 1 896 1 161 1 1788 3 ...
  Values :  0 1 0 1 0 1 0 1 0 1 ...

> head(runValue(x))

[1] 0 1 0 1 0 1

> head(runLength(x))

[1]  780    1  208    1 1599    1

> x > 0

  'logical' Rle instance of length 10000000 with 197127 runs
  Lengths:  780 1 208 1 1599 1 883 1 1038 1 ...
  Values :  FALSE TRUE FALSE TRUE FALSE TRUE ...

> x + y

  'numeric' Rle instance of length 10000000 with 1957707 runs
  Lengths:  780 1 208 1 13 1 413 1 896 1 ...
  Values :  0 1 0 1 0 1 0 1 0 1 ...

> x > 0 | y > 0

  'logical' Rle instance of length 10000000 with 210711 runs
  Lengths:  780 1 208 1 13 1 413 1 896 1 ...
  Values :  FALSE TRUE FALSE TRUE FALSE TRUE ...

> range(x)

[1]  0 26

> sum(x > 0 | y > 0)

[1] 2105185

> log1p(x)

  'numeric' Rle instance of length 10000000 with 1510219 runs
  Lengths:  780 1 208 1 1599 1 883 1 1038 1 ...
  Values :  0 0.693 0 0.693 0 ...

> cor(x, y)

[1] 0.5739224

> shiftApply(249:251, y, x, FUN = function(x, y) var(x,
+     y)/(sd(x) * sd(y)))

[1] 0.8519138 0.8517324 0.8517725
```

2.2 Sequence Extraction

It is often necessary to extract a sequence from another, in analogous manner to ordinary vector extraction or subsetting. A simple operation is to select a list of consecutive elements from a sequence. This is the purpose of the `subseq` function.

Mathematically, a subsequence is slightly more general: selected elements need only to be in the same order, not consecutive. We can generalize this further to sequence extraction, where the order of the elements is no longer fixed. As the order constraint is rarely broken, we will use the term *subsequence* to represent the result of sequence extraction. The most general way to describe such a subsequence would be a vector of indices into the sequence. As it is common to extract consecutive values from the sequence, the indices are usually more efficiently encoded as a list of ranges, i.e. a vector of start positions and a parallel vector of widths.

The general interface for extracting subsequences is `seqextract`, which is supported by all *Sequence* objects, as well as ordinary vectors.

3 Lists

3.1 Basic operations

One often wants to organize and manipulate multiple sequences simultaneously. We could place multiple *Rle* instances, for example, into a list. However, a list is too generic; it does not confer any information about the specific class of its elements. There is no type safety, and it is not possible to define methods specifically for homogeneous lists with elements of a particular class. For example, for a list of *Rle* objects, we may wish to define a method that retrieves the run values for each element, without special type checking. To enable this, we define a specific collection class, *RleList*, for storing *Rle* objects.

In fact, we have done the same for many of the other classes in *IRanges*, as well as the base atomic vectors (raw, logical, integer, numeric, complex and character). All of the collection classes derive from the virtual class *ListLike*, the derivatives of which are all obligated to support two basic list operations: element extraction and length retrieval.

Most collection types, including *RleList*, derive from *TypedList*, a *ListLike* derivative that implements the requisite operations, as well as a number of additional features. These include familiar list functions, such as `c` and `lapply`.

3.2 Annotated Lists

Often when one has a collection of objects, there is a need to attach metadata that describes the collection in some way. The *AnnotatedList* class extends *TypedList* to add two metadata components: an ordinary *list* to hold arbitary objects, called `metadata`, and `elementMetadata`, a data frame with one row per element and any number of columns. Many of the high-level list objects in *IRanges* (described later) are *AnnotatedList* objects.

3.3 Advanced Notes

TypedList also provides some more advanced features. First, as its name suggests, subclasses of *TypedList* can specify a type from which the class of each of its elements must inherit. For example, *RleList* directs *TypedList* to ensure that all of its elements are *Rle* objects. One benefit of this type safety is that it allows methods dispatching on *RleList* to assume that all of the elements are really of class *Rle*.

A second advanced feature of *TypedList* relates to its internal representation. Behind the scenes, *TypedList* is simply an R *list*. By default, there is a one-to-one mapping between elements of the *TypedList* and elements in the *list*. This is a simple design; however, it is not always ideal. If one has many, small objects, the storage overhead, especially for S4 objects, would be relatively high. Our solution is to *compress* the list by

concatenating the elements together, if possible. This forms a single, long element that is virtually split by the *TypedList* interface. A beneficial side effect of this approach is that unlisting (concatenating all of the elements) is cheap, as it reduces to returning the internal representation.

4 Sequence Ranges

When analyzing sequences, we are often interested in specific segments, or consecutive subsequences, of the sequence. It is not uncommon for an analysis task to focus only on the segments themselves, while ignoring the underlying sequence values. A consecutive list of indices would be a simple way to select a consecutive subsequence. However, a sparser representation would be a a range, a pairing of a start position and a width, as used when extracting sequences with `subseq` and `seqextract`, above.

When analyzing subsequences in *IRanges*, each range is treated as an observation. The virtual *Ranges* class represents lists of ranges, or, equivalently, sequences of consecutive integers. The most commonly used implementation of *Ranges* is *IRanges*, which stores the starts and widths as ordinary integer vectors. To construct an *IRanges* instance, we call the `IRanges` constructor. Ranges are normally specified by passing two out of the three parameters: start, end and width (see /tmp/Rinst1032195089/IRanges/help/IRanges-constructor for more information).

Accessing the starts, widths and ends is supported by every *Ranges* implementation.

For *IRanges* and some other *Ranges* derivatives, subsetting is also supported.

4.1 Normality

NormalIRanges

4.2 Lists of Ranges

RangesList IRangesList MaskCollection

4.3 Sequence Extraction

As *Ranges* objects encode subsequences, they may be used directly in sequence extraction. Note that when a *normal Ranges* is given as the index, the result is a true subsequence, in the mathematical sense.

4.4 Transforming Ranges

Making ranges normal `reduce`

Making ranges disjoint `disjoin`, `disjointBins`

Adjusting starts, ends and widths `shift` `*` `narrow` `threebands`

Other transformations `restrict` `reflect` `flank`

4.5 Set Operations

`gaps`, `pgaps` `setdiff`, `psetdiff` `union`, `punion` `intersect`, `pintersect`

4.6 Finding Overlapping Ranges

RangesMatching RangesMatchingList IntervalTree

`overlap`, `%in%`

```
> toLatex(sessionInfo())
```

- R version 2.9.0 (2009-04-17), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US;LC_NUMERIC=C;LC_TIME=en_US;LC_COLLATE=en_US;LC_MONETARY=C;LC_MESSAGES=en_US;LC_`
- Base packages: base, datasets, graphics, grDevices, methods, stats, tools, utils
- Other packages: IRanges 1.2.3

Table 1: The output of `sessionInfo` on the build system after running this vignette.

4.7 Finding Neighboring Ranges

`nearest`, `precede`, `follow`

4.8 Mapping Ranges Between Sequences

RangesMapping `map`

5 Sequence Views

When we extract a sequence with `seqextract`, we can pass multiple ranges, each selecting a single consecutive subsequence. Those subsequences are extracted and concatenated into a single sequence. There are many cases where the user wishes to avoid the concatenation step and instead treat each consecutive subsequence as a separate element in a list.

While one could simply store each extracted sequence as an element in a list object like a *TypedList*, this is undesirable for a couple of reasons. First, the user often wants to preserve the original sequence and declare a set of interesting regions as an overlay. This allows retrieving sequence values even after the ranges have been adjusted. Another benefit of an overlay approach is performance: the sequence values need not be copied.

For representing such an overlay, the *IRanges* package provides the virtual *Views* class, which derives from *Ranges* but also stores a sequence. Each range is said to represent a *view* onto the sequence.

Here, we will demonstrate the *RleViews* class, where the sequence is of class *Rle*. Other *Views* implementations exist, such as *XStringViews* in the *Biostrings* package.

5.1 Manipulating Views

5.2 Aggregating Views

6 Data Sets

XDataFrame XDataFrameList SplitXDataFrameList

7 Sequence Ranges with Data Sets

RangedData RangedDataList

7.1 Applying Over Spaces

RDApplyParams FilterRules