Implementing the pREC methods in C

R. Diaz-Uriarte

October 22, 2008

Contents

1 Summary

These are notes on how we wrote pREC_A and pREC_S in C, and how we modified the algorithm for finding pREC_S. Up to RJaCGH v. 1.2-5, pREC_A and pREC_S were essentially all R code (that made calls to C to obtain the Viterbi path). Now, all of pREC_A and pREC_S is done in C. In addition, the algorithm for pREC_S has been modified substantially, compared to the algorithm as in the first verision of the common regions paper (up to May 2008).

We explain the major steps, design decissions, etc, and provide references to the C and R code. Please read these notes as guided pointers to the code, not as a complete description of what is done. For that, look at the code.

2 Introduction

In the previous code, there were two main reasons the implementation was slow. First, after the complete run of RJaCGH, a new call to C was needed to obtain the Viterbi sequences. Second, the algorithms themselves were implemented in R, not C. In addition, some algorithms operating on the sequences read them repeatedly and had to do several intermediate operations.

Since we are having to run Viterbi as a routine part of the main RJaCGH calculation, we can store those sequences in that first Viterbi run, without a need for second call. Storing, however, could take up lots of space, so the sequences are compressed (using the zip library) after being coded in a slightly more compact format (the same used in the RJaCGH v.1.2-5).

Next, when we want to do either pREC_A or pREC_S, we will need to transfer those (compressed) sequences to C, read and manipulate them, and run the algorithms themselves.

Thus, the major steps in all cases are:

1. Obtain the viterbi paths in the first run of RJaCGH. Move them from C to R.

2. Store the paths in R.

3. When any pREC is desired move the Viterbi paths from R to C.

4. In C, compute on those paths and apply the appropriate algorithm.

3 Viterbi paths: C $\rightarrow$ R

Each Viterbi path in C is stored in a compact representation: for a segment of probes with identical state we only store the state and the position of the last probe with that state. Thus, for a given array we will have several of these paths. Viterbi paths are stored as a linked list (see `struct Sequence`). In addition, we keep track of the number of hidden states that sequence corresponds to.

Storing Viterbi paths this way decreases storage demands and makes it faster to assess if a given sequence is equal to a previous one (see next). The smaller the number of sequences we have, the faster the subsequent pREC computations will be (since we need to loop over all sequences).

We record the sequences when we call the `viterbi` function. The main function is `viterbi_to_Sequence`, called from within `viterbi`. It takes a Viterbi sequence, produces the compact representation above and updates the linked list of compact sequences (`Compare_Add_Seq`). However, before adding a sequence to the linked list (`addSeq`), we verify if that sequence has already been observed and, thus, stored (`CompareSeq_tmp`). If it has, we do not store again the sequence (only update the counter, `MCMC_num`).

At the very end of the run (at the end of `MetropolisSweep`), the linked list is written, as a gzipped file, by `Sequence_to_gzfile`. If the R function RJaCGH was called with `model=''none''`, only one file per array is written. Otherwise (i.e., if there are chromosomes), one file per array * chromosome is written.

The (gzipped) files are read by R via readBin at the end of the function `MetropolisSweep.C` and their content (plus some extra stuff) stored as the list component `viterbi`. By default, the gzipped files are deleted. (This explains that the `viterbi` list component is present at the same level as the mu, k, etc, in an RJaCGH object).

We can examine the files with the Viterbi sequences by setting `.__DELETE_GZIPPED <- FALSE` in R. Each file has the Viterbi sequences corresponding to one array (or to one array by chromosome). In addition to the compact representation above, the first fields are:

k The hidden state.

count_viterbi The `MCMC_num` mentioned above. The number of times we have seen this sequence (of course, this sequence AND this k).

sum_viterbi $\sum count\_viterbi$ for a given k. In other words, the total number of Viterbi sequences for a given k.

count_k Number of times state k is visited.

sum_k $\sum count\_k$.

These fields will be used next.

4 Viterbi paths: R → C

Finding common regions involves repeatedly examining the Viterbi sequences (over different possible combinations of arrays and probes). Thus, we want to have the Viterbi sequences in such a way that traversing and computing on them will be as fast as possible. The best point where this is done is when transferring the sequences from R to C. This is done only once (for a single call of any of the pREC functions).

Thus, for every Viterbi sequence in compact form in R we will do the following when transferring to C:

1. "Stretch it": go from the compact representation to the representation of one entry per probe (i.e., each sequence has length equal to the number of probes).

2. Replace the state of every probe in the previous step by its probability of being altered. This probability is the one given by the component `state.labels` in the R object.

3. Obtain the probability of a sequence. Using the components stored above this is: $(count\_viterbi * count\_k)/(sum\_viterbi * sum\_k)$

4. Multiply the entry for each probe by the probability just computed.

This way, all subsequent calculations will only involve computing the minimum, or summing over arrays (optionally weighted by array weights).

The conversion from the gzipped and compacted sequence stored in R to the stretched and premultiplied sequences in C is done inside the C function `wrap_pREC`. This is the very function called by R. The main work of conversion is done by the function `read_convert_prob_seq`, which is called once for every array. Going from the state of a probe to its probability (and multiplying by the probability of the sequence) is done by `state_to_prob`.

`state_to_prob` has to use, thus, the mappings of state to probability of alteration stored in the `$state.labels` component. Note that, because of the algorithm used by `state_to_prob`, and in contrast to previous versions of RJaCGH, even models that have not been visited have a non-null `$state.labels` component. To catch possible errors, the `$state.labels` component when that k has not been visited is set to -9. Extraction and "stretching" of the `$state.labels` component, in R, is done by function `getStretchedStateProbs`.

Note: if there is a Chromosome component in the RJaCGH object, the call to the pREC functions is done once for each chromosome. Only the corresponding Viterbi sequences are transferred to C, and only for those is the stretching, etc, done; look at the loops in the `pREC` function in R:

```
####################################################
## Main loop: each chromosome is done separately.
## For each chrom, do all arrays.
####################################################

## [other stuff: deleted here]...
for(chromNum in 1:nchrom) {
    ## do stuff
    res <- .C("wrap_pREC",
    ...
}
```

So, after transferring the Viterbi sequences to C, in C we have an array with number of columns = number of probes and number of rows = sum of the number of sequences over all arrays (for the given chromosome). This is the array we call `stretched` in C. Which of the rows in `stretched` correspond to each array is something we know from the vector `starting_indices_sequences`. The entries in this vector we use as offsets (to increase the pointer to the right place in the array) when calling `read_convert_prob_seq`, both to access the correct probabilities of a sequence, and to write to the correct rows in `stretched` (and analogoyusly with the `state.labels` entries), as is done in `wrap_pREC`:

```
for(int i = 0; i < (*numarrays); i++) {
  index_seq = starting_indices_sequences[i];
  index_state_probs = starting_indices_state_probs[i];
```

```
    read_convert_prob_seq(prob_row_seq + index_seq,
                          stretched + index_seq,
                          filenames[i],
                          state_probs + index_state_probs,
                          *alteration,  num_sequences[i]);
  }
```

5 pREC: probability of a sequence

This is a big, major change. Up to now, a given state, say, “Gain-1” had a probability of being gained of 1, and 0 of being everything else. This is now changed, so there is (potentially) non-zero probability of being anything and, as well, the probability of being gained is not exactly 1. Yes, we can recover the old behavior by using, in the R function `relabelStates singleState = TRUE`. But for several reasons, in general, having the smoothness is good. But this raises an issue. Suppose we are looking for common regions of “Gain” and we have the sequence:

```
G1 G1 G1 G1 G1 N
```

where “G1” means, say, the “Gain-1” state, and “N” normal. Up to now, in that sequence it is obvious that $P(S1 = 1, S2 = 1, \ldots, S5 = 1) = 1$ but $P(S1 = 1, S2 = 1, \ldots, S6 = 1) = 0$, as the sixth probe is not gained. But suppose we replace the “G1” and “N” by the probabilities that state “G1” is gained and “N” is gained. For example, suppose we have, in the `state.labels` the matrix:

```
        Loss Normal Gain
Normal   0.1   0.7  0.2
Gain-1   0.01  0.1  0.89
Gain-2   0     0.01 0.99
```

We replace the entries above with the probs of being gained:

```
0.89 0.89 0.89 0.89 0.89 0.2
```

We can easily argue that we want $P(S1 = 1, S2 = 1, \ldots, S5 = 1) = 0.89$. We do not want to multiply the entries, for suppose the sequence had been

```
G1 G1 G1 G1 G1 G2
```

It makes no sense to have $P(S1 = 1, S2 = 1, \ldots, S6 = 1) < P(S1 = 1, S2 = 1, \ldots, S5 = 1)$, when the sixth probe has a prob of being gained much larger than any of the previous probes.

We do not want to average either: in the original sequence, we want an abrupt drop at the sixth probe, and we want that abrupt drop to happen as soon as a probe with low prob of being gained shows up. We do not want to smooth that drop because we could have strong effects of length of sequence before the probe with the small probability.

What makes sense is to compute the minimum. First, this returns the original behavior if we have probabilities of being gained only in $\{0, 1\}$. Second, it gives abrupt drops. Third, given a sequence, as we add probes, the probability of it being altered can never increase. (Which is also convenient for computations).

For instance, in

```
G1 G1 G1 G1 G1 G2
```

we would have $P(S1 = 1, S2 = 1, \ldots, S6 = 1) = 0.89 (= min(0.89, 0.99))$.

Then, given a sequence of m probes, the probability of alteration when we consider the next probe is
$min(prob\_alteration\_sequence\_up\_to\_new\_probe, prob\_alteration\_new\_probe)$. So updating probabilities of alteration is just computing the minimum of two numbers, the current one, and the prob of the new probe. This is what `update_prob_alteration_seq` does. Yes, we had previously pre-multiplied the entries in the sequence by $(count\_viterbi * count\_k)/(sum\_viterbi * sum\_k)$; see next.

Given all the sequences for an array, the probability of alteration of a sequence of m probes is simply the sum of the probabilities of alterations over the sequences. Recall we had previously pre-multiplied the entries in the sequence by $(count\_viterbi * count\_k)/(sum\_viterbi * sum\_k)$, so all we need to do is sum; had we not premultiplied, we would need to do a weighted sum (weighted by $(count\_viterbi * count\_k)/(sum\_viterbi * sum\_k)$). Adding the min of the premultiplied is the same as computing the weighted sum of the (non-premultiplied) mins. However, adding the min of the premultiplied entires is a lot faster. (Many, many, many fewer multiplications).

`prob_seq_array` computes the probability of alteration for an array. This is used by pREC_S. For pREC_A we use `prob_seq_all` which does the computation over arrays (weighting by array weights if needed).

6 wrap_pREC

This is the entry point from R. We use a single function (that later calls pREC_A or pREC_S) to avoid code duplication: the first part of the function deals with reading the gzipped and compressed Viterbi sequences and with obtaining the `stretched` array.

7 pREC_A

The function `pREC_A` is a straightforward implementation of the algorithm. Only some care is needed to properly capture events in the first, second, last, and one-before-last probes. Note that, in contrast to the former R code, we do not need to check for "i + 2" as breaks, etc, are placed in different locations in the code.

8 pREC_S

8.1 Two calls

We store the sequences in a linked list (see `struct regionS`). This is a global variable, as we have to preserve this object over to calls to C. I've tried to avoid using ".Call", but I have no idea of the size of return objects I'd need for the output from pREC_S. What we do follows somewhat what is done in rpart, specifically s_to_rp.c and rpart.s (comments in R-devel from BDR, D. Murdoch and Dirk Eddelbuettel helped me understand how to do this).

We make a first call from R and do the computations of pREC_S. We return to R the object sizes, and we do a second call from R to C to retrieve the result in the call `res <-.C("return_pREC_S",` with appropriately sized return vectors. But so that we can access the object (the linked list with the common regions) in the second call, we make it a global. Moreover, and in contrast to the other linked lists in the C code, memory allocation is done via calls to `malloc`, not `R_alloc`. The R Extensions manual says of R_alloc "Here R will reclaim the memory at the end of the call to .C.". In fact, using gc() between the first and second calls to .C (between "wrap_pREC" and "return_pREC_S") can lead to weird errors if we use R_alloc. Thus we use malloc, and free the memory as we return the vectors to R in the second call to .C, in `return_pREC_S`.

8.2 pREC_S algorithm

```
1  for $Start \leftarrow 1$ to $TotalNumberOfProbes$ do
2      $SetArrays\_A \leftarrow \phi$ ;
3      for $array \leftarrow 1$ to $TotalNumberOfArrays$ do
4          if $P(S_{Start} = 1|array) \geq p_w$ then
5              $SetArrays\_A \leftarrow SetArrays\_A \cup array$;
6      if $|SetArrays\_A| \geq freq.arrays$ then
7          $End \leftarrow Start + 1$;
8          while $End \leq TotalNumberOfProbes$ do
9              $SetArrays\_B \leftarrow \phi$;
10             foreach $candidate\_array$ in $SetArrays\_A$ do
11                 if $P(S_{Start}, \ldots, S_{End} = 1|candidate\_array) \geq p_w$ then
12                     $SetArrays\_B \leftarrow SetArrays\_B \cup candidate\_array$;
13             if $|SetArrays\_B| < |SetArrays\_A|$ then
14                 UpdateRegionS($Start, End - 1, SetArrays\_A$);
15                 if $|SetArrays\_B| \geq freq.arrays$ then
16                     $SetArrays\_A \leftarrow SetArrays\_B$;
17                 else
18                     break out of the while loop
19             $End \leftarrow End + 1$;
20             if $End = (TotalNumberOfProbes + 1)$ then
21                 UpdateRegionS($Start, End - 1, SetArrays\_B$);
```

Algorithm 1: pREC_S algorithm

Some explanations of the algorithm.

The function `UpdateRegionS` (called in lines 14 and 21) simply adds a region to the set of regions already stored. Adding a region means storing the first probe of the region (*Start*), the last probe of the region ($End - 1$), and the arrays that compose the region (those in one of the $SetArrays$). The function `UpdateRegionS`, however, must check that the region to be added is not a subset of some previously added region. Suppose in the run that started with probe $S2$ we found the region $((S2, S3, S4), (A1, A2))$. Now, in the run that starts with probe $S3$ we find the region $((S3, S4), (A1, A2))$; obviously, the newly found region is simply a completely contained subset of the previously found region, and we should not add this newly found region as a new region.

The conditions in lines 4 and 11 refer to one of the conditions of the algorithm: an array can only be considered part of a common region if the probability of the given sequence of probes (starting at $Start$ and ending at End or, in the one-probe case, starting and ending at $Start$) is larger than p_w.

Likewise, the conditions in lines 6 and 17 refer to the second condition: at least $freq.arrays$ arrays must fulfill that the sequence has a probability larger than p_w.

Line 13 represents the condition where the number of arrays that fulfill the condition when we add a probe decreases. In other words, at step t, with $End = Start + t$, we had a set of arrays that fulfilled p_w. As soon as we add a new probe (i.e., "stretch" the region by one probe, so we are at step $t + 1$ with $End = Start + t + 1$), at least one array no longer satisfies p_w. This means that at step t we had one common region over a set of arrays to which we cannot add another probe. (Note that we know this regardless of how many arrays fulfill p_w at step $t + 1$). Therefore, as soon as the number of arrays in $SetArrays\_B$ becomes smaller than $SetArrays\_A$, we know we found a common region in the previous step, and we have to update the set of regions.

Line 16 is needed to allow capturing subsequent decreases (if there were any) in the number of arrays that meet the condition as we keep stretching the region (adding probes). Strictly the condition check in line 15 is not needed, but with it in place we avoid doing an assignment operation (really as many assignments as there are elements in $SetArrays\_B$) when we know we will not keep stretching the region (because the current number of arrays is below freq.arrays).

We use a while loop (line 8) and we will often exit out of the loop (line 17) via a break statement (or whichever similar construct is provided in the specific programming language). We could avoid the "break" and rewrite the loop in a different way, but as currently written the algorithm provides for straightforward condition checks and conditional branches.

The condition in line 20 only happens when we reach the end of the probes; thus, if we have met the conditions so far (p_w and $freq.arrays$) we must add the current set of arrays (from probes $Start$ up to the last one) to the regions. (We could have made the "while" in line 8 always a true condition, and add another "break" when we reach the end of probes).

In any specific implementation, it is not necessary to explicitly do assigments as in lines 2 and 9. In our current C implementation, we use two additional variables (one for the vector that represents $SetArrays\_A$ and one for the vector that represents $SetArrays\_B$) that tell us how many valid elements there are in each set, and we only access and use those valid elements. Likewise, the set union operation as in lines 5 and 12 can instead be implemented as simply an assignment to a specific position of a vector. Similar comments apply to line 16. For instance, we could rewritte lines 4 and 5 as:

```
$valid\_elements \leftarrow 0$;
if $P(S_{Start} = 1 | array) \geq p_w$ then
    $valid\_elements \leftarrow valid\_elements + 1$;
    $SetArrays\_A[valid\_elements] = array$;
```

valid_elements is also the cardinality of the set. (Note that, in C, and other languages that index arrays starting at 0, we would increase *valid_elements* after the assignment to *SetArrays*).

9 Miscellaneous comments

9.1 Transposing `stretched` and accumulating on it

In both pREC_A and pREC_S we do not operate on `stretched` directly, but on its transpose. As we loop over probes, by having `stretched_tr` be probe by sequence, we have all the entries for a probe contiguous in memory.

Moreover, once a given probe, say t has been considered as the origin of a sequence, that probe t is never revisited. Thus, we can accumulate the updates of the probability of a sequence in `stretched_tr[t]`.

9.2 A single pREC function

We used to have two pREC functions, one for pREC_A and one for pREC_S, with possible variants depending on the type of RJaCGH object (chromosome, array, etc). We only have one now, which leads to deleting more than 8 pages of R code. First, a lot of the R code in *pREC* is involved in preparing the files and similar functions, which are identical to the two pREC algorithms. Second, the code does not really change depending on whether the object is array, or chrom, or whatever. The only main difference is that if the object is of type "none" there is one fewer level of nesting. That is taken care of via the minor kludge:

```
} else {
    narrays <- 1
    nchrom <- length(unique(obj$Chrom))
    if(nchrom == 0) nchrom <- 1
    ## To prevent from further handling the single array case
    ## as a special one:
    obj.tmp <- obj
    obj <- list()
    obj[[1]] <- obj.tmp
    rm(obj.tmp)
    gc()
}
```

Likewise, the main difference between a "genome" and a "chromosome" type object is that, for "genome", the `state.labels` component is common to all chromosomes, whereas for the chromosome object there is one `state.labels` element per chromosome. Again, this is easy to deal with as in

```
## If Genome or None type objects probs of states common for all
## chroms.
if(!objChrom) {
    stretchedProbsList <- lapply(obj[1:narrays],
                                 getStretchedStateProbs)
}
```

There are a few other places where the type of object had to be accounted for, but this seemed better than duplicating code, or writing minor functions with lots of parameters and potentially expensive copy of objects.

9.3 Speed comparisons

The code for RJaCGH is slightly faster; for example, for the fits for Douglas and Pollack, it now takes 85% of the time it used to take (23300 vs. 27000 seconds for Pollack with the new and old code, and 23500 and 27600 for Douglas). The most impressive improvements are, of course, in the pREC computations. These are two tables of some of them.

For pREC_S:

Fit	New code	Old code
Gain4.tumors	171	6547
Loss4.tumors	159	824
Gain2	3.8	131842
Gain2.MSI	2.98	7500
Loss2.MSI	2.98	153
Gain1	2.15	50310
Loss1	2.16	46236

For pREC_A:

Fit	New code	Old code
Gain.35	9	10248
Gain.50	9	3050
Loss.50	9	3057
Loss.35	9	7573
Gain.MSI	0.85	16575
Loss.MSI	0.84	157
Gain.CIN	1.43	42702
Loss.CIN	1.39	47598
Gain.50 (Douglas)	2.1	16076
Loss.50 (Douglas)	2.1	7485
Gain.50.weights	2.1	10542
Loss.50.weights	2.14	3078

9.4 Code changes

The code has changed a lot. Just as a couple of figures: the lines of code of R have gone from 4500 to 3300, so the new code is about 75% of the size of what it used to be. The C code, however, has doubled: from about 1700 to 3600 lines (without comments).