Suffix sorting in EM

Using the DC3 algorithm seen in class, and based on a variation of mergesort, design an EM algorithm to build the suffix array for a text of N symbols. The I/O complexity should be the same as that of standard sorting, namely, $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$ block transfers.

Solution The main idea for solving this problem is, as Prof. Grossi said in class, to prepermute the datas in external memery in order to load them in main memory in linear time (in EM, i.e. $O(\frac{N}{B})$).

Let follow the steps DC3 algorithm is compesed of, and show in which parts a permutation is necessary(see the third section of [1]).

1. **Construct a sample:** In this step we need to pair the sample positions with the corrisponding 3-gram, since we cannot access in reasonable time to a random position in following sorts. Now we have for $k = 0, 1, 2$ the sets of pairs:

   $$B_k = \{(i, s[i]s[i+1]s[i+2]) \mid i \ mod \ 3 = k)\}$$

   Let $C = B_1 \cup B_2$ be the sample set. This can be made with linear number of block transfers by scanning the input string S.

2. **Sort sample suffixes:** First sort the suffixes using a mixture of radix and merge sort. That operation takes $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$. Now let generate the ids for the renaming steps, in sequel denoted by $id(i)$, by scanning the sorted suffixes (Note that in general $rank(i) \neq id(i)$ with $(s, i) \in C$ since there can be duplicates). Then permute the pairs $(id(i), s)$ by the congruence class of i and if equals by i, where $(i, s) \in C$. If there is no duplicates we have a total order for them, otherwise the recursion starts, constructing the input string simply concatenating the just permuted ids. In both cases at the end of this step we have the array S_{12}, which contains the correct ranks of suffixes in C.

3. **Sort nonsample suffixes:** First generate S_{12}^{-1} permuting S_{12} by thier index, i.e. generate the inverse. Then scan the input string character by character and construct the following 3 sets:

   - $S_{01} = \{(s[i], S_{12}^{-1}[i+1], i) \mid i \ mod \ 3 = 0\}$, this set is used for comparing elements in B_0 with elements in B_1
   - $S_{02} = \{(s[i], S_{12}^{-1}[i+1], S_{12}^{-1}[i+2], i) \mid i \ mod \ 3 = 0\}$, this set is used for comparing elements in B_0 with elements in B_2
   - $S_{12} = \{(s[i], S_{12}^{-1}[i+1], i) \mid i \ mod \ 3 = 1\} \cup \{(s[i], s[i+1], S_{12}^{-1}[i+2], i) \mid i \ mod \ 3 = 2\}$, this is the set of elements in C already prepared for merge comparison.

   All these sets can be constructed in linear time. Let sort thef first two sets in lexicographically order (taking the well-known time of $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$), and permute the last one by $S_{12}^{-1}[i]$. Note that the the first two sets are sorted in the same way but stored different information, both useful for the follwing merge step. The last two was already sorted by their position in step 2.

4. **Merge:** This step can be made easily using the following comparison function:

   ```
   /* input elements are taken resp. by S_12, S_01, S_02 */
   compare(e12, e01, e02)
     if ( e12.i mod 3 = 1 )
       return lex-less(e12, e01); /* return true if e1 < e2 in lexicographical order
     else
       return lex-less(e12, e02);
   ```

   The merge step can be made in I/O linear time and block transfers. The output is the array of ranks sorted by suffix's starting position i.

[1] https://www.cs.helsinki.fi/u/tpkarkka/publications/jacm05-revised.pdf