Further Improvements for SP 800-90B Tuple Counts

Aaron Kaufer

April 10, 2019

1 Overview

Recently, UL improved the SP 800-90B [2] code by implementing a Longest Common Prefix (LCP) array to compute the tuple counts for both the *t-tuple* and *LRS* estimates (see [1] for details). We give further improvements to the performance time by aggregating the tuple counts across multiple tuple sizes, resulting in only one traversal of the LCP array.

2 Count Aggregation

The main observation is that every time we see a non-zero value in the LCP array, we can efficiently keep track of all of the smaller tuple size counts. We define `A` as the array that keeps track of tuple repeat counts. Let `L` be an LCP array. Assuming we are currently at index `i` ($1 \leq i \leq n$) of the LCP array, `A` is updated as $A[L[i]] = A[L[i]] + 1$. Now, if $L[i] < L[i-1]$, the tuple counts for tuple sizes $L[i] < t \leq L[i-1]$ are computed as follows:

```
for(t = L[i-1]; t > L[i]; t--){
     A[t]  += A[t+1]
     A[t+1] = 0
     /* t-tuple count is A[t]+1 */
}
```

Note that the `t`-tuple count is $A[t] + 1$ since the LCP array counts the number of tuple repeats, which is one less than the number of occurrences.

To facilitate applying this algorithm, we define $L[0] = L[n] = 0$. This makes it so that the counts are correctly updated for the boundary cases, since $L[0] \leq L[1]$ and $L[n-1] \geq L[n]$. Furthermore, let `v` be the maximum value in `L`. The full algorithm to compute each of the tuple counts is then:

```
A[1] = A[2] = ... = A[v+1] = 0 /* v+1 avoids a buffer overflow for A[t+1] */
for(i = 1; i <= n; i++){
    if(L[i] < L[i-1]){
          for(t = L[i-1]; t > L[i]; t--){
               A[t]  += A[t+1]
               A[t+1] = 0
              /* do something with t-tuple count A[t]+1 ... */
          }
          if(L[i] > 0) A[L[i]] += A[L[i]+1] /* carry over count for t = L[i] */
          A[L[i]+1] = 0
     }
     if(L[i] > 0) A[L[i]]++ /* update count for t = L[i] */
}
```

As an example, we will apply this algorithm to the string "banana" in order to get the tuple counts. We ignore the character '$', which is typically used to denote the smallest character. The suffix and LCP array for this string are as follows:

i	suffix
0	a
1	ana
2	anana
3	banana
4	na
5	nana

$$\Longrightarrow$$

i	LCP
0	0
1	1
2	3
3	0
4	0
5	2
6	0

Note that $\mathtt{L}[0] = \mathtt{L}[6] = \mathtt{0}$, where $\mathtt{n} = \mathtt{6}$. Also, $\mathtt{v} = \mathtt{3}$.

The following table depicts the steps of the algorithm and corresponding counts ($\mathtt{A}[\mathtt{t}] + \mathtt{1}$) for each t-tuple that occurs more than once.

i	t	A	A[t]+1	t-tuple
0		[0 0 0 0]		
1		[1 0 0 0]		
2		[1 0 1 0]		
3	3	[1 0 1 0]	2	ana
3	2	[1 1 0 0]	2	an
3	1	[2 0 0 0]	3	a
4		[0 0 0 0]		
5		[0 1 0 0]		
6	2	[0 1 0 0]	2	na
6	1	[1 0 0 0]	2	n

3 t-Tuple Counts

Since the t-tuple estimate only requires finding the maximum t-tuple occurrence, we can ignore tuple counts that are at most the current maximum count. To achieve this, we just need to keep track of the indices (tuples) for the non-zero entries of A. In other words, we keep track of the indices $\mathtt{L}[\mathtt{i}]$ of A for which $\mathtt{L}[\mathtt{i}] > \mathtt{0}$. Letting I be the array that keeps track of these indices, and Q be the array that stores the maximum tuple counts, the following algorithm will calculate Q:

```
A[1] = A[2] = ... = A[v+1] = 0 /* v+1 not necessary here (for consistency) */
Q[1] = Q[2] = ... = Q[v] = 1
j = 0
for(i = 1; i <= n; i++){
     c = 0
     if(L[i] < L[i-1]){
          t = L[i-1]
          j--
          while(t > L[i]){
               if((j > 0) && (I[j] == t)){
                    /* update count for non-zero entry of A */
                    A[I[j]]  += A[I[j+1]]
                    A[I[j+1]] = 0
                    j--
               }

               if(Q[t] >= A[I[j+1]]+1){
                    /*
                     * Q[t] is at least as large as current count,
                     * and since Q[t] <= Q[t-1] <= ... <= Q[1],
                     * there is no need to check zero entries of A
                     * until next non-zero entry
                     */
                    if(j > 0)
                         /* skip to next non-zero entry of A */
                         t = I[j]
                    else
                         /*
                          * no more non-zero entries of A,
                          * so skip to L[i] (terminate while loop)
                          */
                         t = L[i]
               }
               else
                    /* update Q[t] with new maximum count */
                    Q[t--] = A[I[j+1]]+1
          }

          c = A[I[j+1]] /* store carry over count */
          A[I[j+1]] = 0
     }

     if(L[i] > 0){
          if((j < 1) || (I[j] < L[i]))
               /* insert index of next non-zero entry of A */
               I[++j] = L[i]
          A[I[j]] += c+1 /* update count for t = I[j] = L[i] */
     }
}
```

The following table shows the steps of the algorithm applied to the string "banana" resulting in Q for the t-tuple estimate:

i	t	A	I	Q	A[t]+1	t-tuple
0		[0 0 0 0]	[0 0 0]	[0 0 0]		
1		[1 0 0 0]	[1 0 0]	[0 0 0]		
2		[1 0 1 0]	[1 3 0]	[0 0 0]		
3	3	[1 0 1 0]	[1 3 0]	[0 0 2]	2	ana
3	2	[1 1 0 0]	[1 0 0]	[0 2 2]	2	an
3	1	[2 0 0 0]	[1 0 0]	[3 2 2]	3	a
4		[0 0 0 0]	[0 0 0]	[3 2 2]		
5		[0 1 0 0]	[2 0 0]	[3 2 2]		
6	2	[0 1 0 0]	[2 0 0]	[3 2 2]	2	na

Notice that for $i = 6$, $t = 1$ does not need to be considered since $Q[2] = A[2] + 1$ and $t = 2$ is the only non-zero entry for A at this point (implying that $A[1] + 1 \leq Q[2] \leq Q[1]$).

4 LRS Counts

Since the LRS estimate requires summing the count products for each of the tuple counts, we cannot use the trick of only keeping track of the non-zero entries of A, as we did to compute Q for the t-tuple counts. However, we can skip all of the tuple sizes that are less than u, where u is the smallest tuple size such that Q[u] is less than the t-tuple estimate threshold. The following algorithm computes the sums of count products S (numerator of P_W in [2, Section 6.3.6]) for the LRS estimate:

```
A[1] = A[2] = ... = A[v+1] = 0 /* v+1 avoids a buffer overflow for A[t+1] */
S[1] = S[2] = ... = S[v] = 0
for(i = 1; i <= n; i++){
    if((L[i-1] >= u) && (L[i] < L[i-1])){
          b = L[i]
          if(b < u) b = u-1

          for(t = L[i-1]; t > b; t--){
               A[t]  += A[t+1]
               A[t+1] = 0
               S[t]  += (A[t]+1) * A[t] /* update sum */
          }

          if(b >= u) A[b] += A[b+1] /* carry over count for t = L[i] */
          A[b+1] = 0
     }

     if(L[i] >= u) A[L[i]]++ /* update count for t = L[i] */
}
```

The following table gives the steps of the algorithm applied to the string "`banana`" yielding `S` for the LRS estimate:

i	t	A	A[t]+1	S	t-tuple
0		[0 0 0 0]		[0 0 0]	
1		[1 0 0 0]		[0 0 0]	
2		[1 0 1 0]		[0 0 0]	
3	3	[1 0 1 0]	2	[0 0 2]	ana
3	2	[1 1 0 0]	2	[0 2 2]	an
3	1	[2 0 0 0]	3	[6 2 2]	a
4		[0 0 0 0]		[6 2 2]	
5		[0 1 0 0]		[6 2 2]	
6	2	[0 1 0 0]	2	[6 4 2]	na
6	1	[1 0 0 0]	2	[8 4 2]	n

References

[1] Joshua E. Hill. *Algorithms for t-tuple and LRS Estimates.* Revision 1.0, June 11 2018, `https://www.untruth.org/~josh/sp80090b/NIST.SP.800.90B implementation comments 20181015.pdf`.

[2] Meltem Sönmez Turan, et al. *Recommendations for the Entropy Sources Used for Random Bit Generation*, NIST SP 800-90B, January 2018.

A Counting Single Tuple Occurrences

Although the LCP array is adept for counting repeated occurrences of tuples, it can also be used to (indirectly) count the number of tuples that occur once as follows. Let `Y` be the array that stores the number of tuples that occur once for each tuple size. Set the initial value of $\mathtt{Y}[\mathtt{t}]$ to $\mathtt{Y}[\mathtt{t}] = \mathtt{n} - \mathtt{t} + \mathtt{1}$. Then for each count $\mathtt{A}[\mathtt{t}] + \mathtt{1}$, set $\mathtt{Y}[\mathtt{t}] = \mathtt{Y}[\mathtt{t}] - (\mathtt{A}[\mathtt{t}] + \mathtt{1})$. This will eliminate all of the counts for tuples that occur multiple times, leaving the number of tuples that occur once.

An application of this is to compute collision entropy $\mathtt{H}_2$, since this requires computing all of the tuple counts, not just for the repeated occurrences.