Supplementary Material

Construction of a superword array

We describe an efficient algorithm for building the array SW. The algorithm is outlined as follows. Initialize $SW[i]$ to i for each i from 1 to n. Sort SW at word level $wlev$ for each $wlev$ from 1 to $wlcut$. The sorting at word level $wlev$ is performed by using a lookup table.

The lookup table consists of a bucket array $Buck$ with each index between 0 and 4^w, and a list array $List$ with each index between 1 and n. For convenience, $Buck[-1]$ is an alias for $Buck[4^w]$. The lookup table keeps the list of positions for every word code between -1 and $4^w - 1$ (16). The list of positions for word code c contains the positions of all occurrences of the word code c in $Cseq$, where c occurs at a position of $Cseq$ if c is the code of the word starting at the position. The list of positions $p_1, p_2, ..., p_s$ in increasing order for any word code c are linked backward with $Buck$ and $List$: $Buck[c] = p_s$, $List[p_i] = p_{i-1}$ for $2 \leq i \leq s$, and $List[p_1] = 0$. The special value 0, which is not a position of $Cseq$, indicates the start of the list.

The sorting is done in two phases. First, build the lookup table by processing the positions in SW in order of increasing index. Second, copy the positions from the lookup table into SW in order of decreasing index.

Consider phase 1. For each word code, set up the empty list for the code in the lookup table. Assume that the array SW is in order at word level $wlev - 1$. For each i from 1 and n, $SW[i]$ is the start position of a superword at word level $wlev - 1$. If there is a word immediately before the superword, concatenate the word and the superword to form a superword at word level $wlev$ starting at position $p = SW[i] - w$. The code of the word is $c = Code(p)$. Append the position p to the list of positions for the word code c, where $Buck[c]$ is the last position in the list. After the positions in SW are considered, the positions in the lookup table are less than or equal to $n - w$. For each p from $n - w + 1$ to n, append the position p to the list for the word code -1.

Below is a precise description of the steps in phase 1. Set Buck[c] to 0 for each c

from -1 to $4^w - 1$. For each i from 1 to n, if $p = SW[i] - w$ is positive, then let c be $Code(p)$, set $List[p]$ to $Buck[c]$, and set $Buck[c]$ to p. For each p from $n - w + 1$ to n, set $List[p]$ to $Buck[-1]$, and set $Buck[-1]$ to p.

We analyze the lists of positions in the lookup table before considering phase 2. The initialization of $SW[i]$ to i ensures that SW is in order at word level 0. For any $wlev \geq 1$, assume that SW is in order at word level $wlev - 1$ at the start of phase 1, which is true for $wlev = 1$. At the end of phase 1, for any list of positions for word code c in the lookup table, the superword starting at any position in the list at word level $wlev$ begins with a word of the code c. Thus, for any two lists of positions for different word codes in the lookup table, the superwords at word level $wlev$ starting at the positions in the list for the smaller word code are lexicographically before the superwords at word level $wlev$ starting at the positions in the list for the larger code.

Next consider a list with at least two positions in the lookup table. Let p and t be two different positions in the list for word code c. Assume that p is added to the list before t, which means that the position $p + w$ is before the position $t + w$ in SW at the start of phase 1. The superword starting at p at word level $wlev$ consists of an initial word of code c and a superword starting at $p + w$ at word level $wlev - 1$. Similarly, the superword starting at t at word level $wlev$ consists of an initial word of code c and a superword starting at $t + w$ at word level $wlev - 1$. Because the array SW is in order at word level $wlev - 1$ and the position $p + w$ is before the position $t + w$ in SW at the start of phase 1, the superword starting at $p + w$ at word level $wlev - 1$ is lexicographically before the superword starting at $t + w$ at word level $wlev - 1$. Thus, the superword starting at p at word level $wlev$ is lexicographically before the superword starting at t at word level $wlev$.

The analysis above suggests that in phase 2, the lists of positions in the lookup table be copied to SW in order of decreasing index. The lists are processed in order of decreasing word code. The positions in the current list are processed backward. Here is a precise description of the steps in phase 2, where the body of a statement is indicated by a pair of parentheses. Initialize k to $n + 1$. For each c from $4^w - 1$ to -1, { initialize p to $Buck[c]$, and if p is positive, { repeatedly decrease k by 1, set

$SW[k]$ to p, and update p to $List[p]$ until p is not positive, } and set $Buck[c]$ to k. }
Set $Buck[4^w]$ to $n+1$.

Note that the statements that set $Buck[4^w]$ to $n+1$ and $Buck[c]$ to k are included to ensure that the array $Buck$ has the following property after the algorithm terminates. For each word code c, if $Buck[c+1] - Buck[c]$ is positive, then the word of the code c occurs in the combined sequence, the positions of all occurrences are in a consecutive section of SW, $Buck[c]$ is the smallest index of the section and $Buck[c+1] - Buck[c]$ is the number of positions in the section. For any word, the array $Buck$ is used to locate quickly the section of positions in SW at which the word occurs.

If the order of positions in SW at the start of phase 1 is the same as the order of positions in SW at the end of phase 2 in the current iteration, then the construction of SW is terminated as no more work is necessary. Note that in phase 2, if $SW[k]$ is equal to p before the assignment for every k, then the order of positions in SW at the start of the current iteration is the same as that at the end of the iteration. In any iteration of the algorithm, the order of positions in SW at the end of the iteration depends only on the order of positions in SW at the start of the iteration. Thus, if the current iteration results in no change to the order of positions in SW, then any future iteration results in no change to the order either. In this case, the array SW is sorted at every word level.

The construction of the array SW requires an integer array of size $4^w + 1$ and three integer arrays of size n: SW, $List$, and $Wcode$, where $Wcode[p] = Code(p)$. The array $Wcode$ is computed once in linear time. The word size w is selected such that $4^w \leq n$. Because each iteration takes time proportional to n, the construction takes time proportional to $wlcut \times n$. In practice, the parameter $wlcut$ is often set to a number between 1 and 20.

Use of a superword array

We describe a method for locating the occurrences of the superwords from the whole data set in the subset by using the array SW for the subset. In the method, the reads in the whole data set are processed one at a time. The current read is compared with the subset by using the array SW for the subset. The positions of the current read are considered one at a time in increasing order. Let q be the current position. If a superword starting at the position q at a word level between 1 and $wlcut$ is free of the characters N and #, and is unique with respect to the subset, then the minimum word level $wlev$ and a section of SW are computed such that the section contains the positions of all unique occurrences of the superword at word level $wlev$ starting at the position q. Otherwise, the empty section is reported for the position q. A section of SW is represented by its leftmost (lt) and rightmost (rt) indices.

For the current position q, the word level $wlev$ and the section of SW for q are computed as follows. Start with an initial section with indices i that have the same first component code as the position q, that is, $Wcode[SW[i]] = Code(q)$. If the section is too large, then find, by using binary search, a subsection with indices i that have the same second component code as the the position q, that is, $Wcode[SW[i] + w] = Code(q + w)$. This is repeated until a small section of SW is located or the word level $wlcut$ is reached. A precise description of this procedure is given below.

Let c denote $Code(q)$. If $c = -1$ or $Buck[c+1] - Buck[c] = 0$, then stop with the empty section by setting lt to 1 and rt to 0. Otherwise, set $wlev$ to 1, set lt to $Buck[c]$, and set rt to $Buck[c+1] - 1$. (Loop) If $rt - lt + 1 \leq pncut$, then stop with the word level $wlev$ and the section of SW from lt to rt. Otherwise, if $wlev = wlcut$, then stop with the empty section. Otherwise, set c to $Code(q + wlev \times w)$, find, by using binary search, the smallest index i such that $lt \leq i \leq rt$ and $Wcode[SW[i] + wlev \times w] = c$, find the largest j such that $lt \leq j \leq rt$ and $Wcode[SW[j] + wlev \times w] = c$, and increase $wlev$ by 1. If $c = -1$ or the index i or j can not be found, then stop with the empty section. Otherwise, set lt to i, set rt to j, and go to the Loop step.

The method, in the worst case, takes time proportional to $wlcut \times \log k$ for the current position q of word code c, where k is the number of positions in SW with word code c. The time requirement of the method is acceptable for the computation of overlaps between the current read in the whole data set and reads in the subset because the time requirement of the method is not a major portion of the total time requirement of the application and main memory is limited. For the current read f from the whole data set and a read g from the subset with a unique superword, a banded dynamic programming algorithm is used to compute an overlap between the reads f and g. The banded dynamic programming algorithm is much slower than the method.