Supplementary Material for "Minimal Positional Substring Cover is a haplotype threading alternative to Li and Stephens Model"

Ahsan Sanaullah[1], Degui Zhi[2,⋆], and Shaojie Zhang[1,⋆]

[1] Department of Computer Science, University of Central Florida,
Orlando, FL 32816, USA

[2] Center for AI and Genome Informatics, School of Biomedical Informatics,
University of Texas Health Science Center at Houston,
Houston, TX 77030, USA

Proofs of Claims and Lemmas

Claim 1 *Every $i+1$-th positional substring in any MPSC of z by X contains index $\pi_2(\mathcal{C}[i])+1$. Where $\mathcal{C}$ is a rightmost MPSC of z by X.*

Proof. Suppose there existed an $i \in \{0,\ldots,|\mathcal{C}|-2\}$ and $\mathcal{D}$, MPSC of z by X, s.t. $\pi_2(\mathcal{C}[i])+1 < \pi_1(\mathcal{D}[i+1])$. Then, since $\mathcal{D}$ is a cover of z by X, it contains a positional substring that contains index $\pi_1(\mathcal{C}[i])+1$, call it $\mathcal{D}[j]$. If $j < i+1$, then by definition of i-th positional substring, $\mathcal{C}$ is not rightmost since $\pi_2(\mathcal{D}[i]) > \pi_2(\mathcal{C}[i])$. If $j > i+1$, the definition of i-th positional substring is contradicted since $\pi_1(\mathcal{D}[j]) < \pi_1(\mathcal{D}[i+1])$ and $j > i+1$. Therefore, no such i and $\mathcal{D}$ exist.

Suppose there existed an $i \in \{0,\ldots,|\mathcal{C}|-2\}$ and $\mathcal{D}$, MPSC of z by X, s.t. $\pi_2(\mathcal{C}[i])+1 > \pi_2(\mathcal{D}[i+1])$. Then, the set $\{\mathcal{D}[j] : i+2 \leq j < |\mathcal{C}|\}$ covers the indices $[\pi_2(\mathcal{D}[i+1])+1, |z|-1]$ with $|\mathcal{C}|-i-2$ positional substrings and the set $\{\mathcal{C}[j] : 0 \leq j \leq i\}$ covers the indices $[0, \pi_2(\mathcal{D}[i+1])]$ with $i+1$ positional substrings. Their union is a positional substring cover of z by X with $|\mathcal{C}|-1$ positional substrings. This contradicts the fact that $\mathcal{C}$ is an MPSC of z by X. Therefore, no such i and $\mathcal{D}$ exist. □

Lemma 1 (Required Regions). *There exists a contiguous nonempty range of sites for every $i \in \{0,\ldots,|\mathcal{C}|-1\}$, such that $\mathcal{C}[i]$ contains this range of sites for all MPSCs $\mathcal{C}$ of z by X. Call the largest such range the i-th required region. For $i \in \{1,\ldots,|\mathcal{C}|-2\}$, this range is $[\pi_2(\mathcal{C}_r[i-1])+$*

⋆ Corresponding authors: shzhang@cs.ucf.edu, degui.zhi@uth.tmc.edu

$1, \pi_1(\mathcal{C}_l[i+1]) - 1]$, *where* $\mathcal{C}_r$ *and* $\mathcal{C}_l$ *are rightmost and leftmost MPSCs of* z *by* X *respectively. The required region for* $\mathcal{C}[0]$ *is* $[0, \pi_1(\mathcal{C}_l[1]) - 1]$, *and for* $\mathcal{C}[|\mathcal{C}| - 1]$, $[\pi_2(\mathcal{C}_r[|\mathcal{C}| - 2]), |z| - 1]$.

Proof. By Claim 1, every i-th substring must contain index $\pi_2(\mathcal{C}_r[i-1]) + 1$ for $i \in \{1, \ldots, |\mathcal{C}| - 1\}$. The 0-th substring must contain index 0 by definition of MPSC and i-th substring (every site is covered and $\mathcal{C}[0]$ has the smallest starting point of all positional substrings in $\mathcal{C}$). Therefore, the required region exists and is nonempty for all i.

In their Claim 5, Sanaullah et al. showed that every i-th substring must contain index $\pi_1(\mathcal{C}_l[i + 1]) - 1$ for $i \in \{0, \ldots, |\mathcal{C}| - 2\}$ (Sanaullah et al. 2022). Furthermore, the $|\mathcal{C}| - 1$-th positional substring must contain index $|z| - 1$ by definition of MPSC and i-th substring (every site is covered and $\mathcal{C}[|\mathcal{C}| - 1]$ has the largest ending point of all positional substrings in $\mathcal{C}$).

Positional substrings cover a contiguous range of sites. Therefore the i-th required regions for $i \in \{1, \ldots, |\mathcal{C}| - 2\}$ contains indices $[\pi_2(\mathcal{C}_r[i - 1]) + 1, \pi_1(\mathcal{C}_l[i + 1]) - 1]$. $\mathcal{C}[0]$ and $\mathcal{C}[|\mathcal{C}| - 1]$ must contain indices 0 and $|z| - 1$ respectively by definition of MPSC and i-th positional substring.

Lastly, these ranges are the complete required regions. For $i \in \{1, \ldots, |\mathcal{C}| - 1\}$, there exists an MPSC in which the i-th positional substring doesn't contain index $\pi_2(\mathcal{C}_r[i-1])$, namely a rightmost MPSC with its i-th positional substring trimmed to only include sites not covered by its $i - 1$-th substring. For $i \in \{0, \ldots, |\mathcal{C}| - 2\}$, there exists an MPSC in which the i-th positional substring doesn't contain $\pi_1(\mathcal{C}_l[i+1])$, namely a leftmost MPSC with its i-th positional substring trimmed to only include sites not covered by its $i + 1$-th substring. Finally, -1 is not part of the 0-th required region and $|z|$ is not part of the $|\mathcal{C}| - 1$-th required region because it is impossible for either of these indices to be contained in an MPSC of z by X. □

Claim 2 *Every non-empty positional substring* $m = (i, j, z)$ *contained in* z *and a string in* X *is contained in a set maximal match from* z *to* X.

Proof. If m is a set maximal match, it is contained in a set maximal match (itself). If m is not a set maximal match, then there exists a larger match, n, between z and $s \in X$ that contains m. If n is set maximal, we are done, otherwise, there exists a match larger than n that contains it (and therefore m). We can repeat this logic until a set maximal match containing m is found. This process is guaranteed to stop because there are a finite amount of matches from z to X (since z and X are finite) and each match is considered at most once. □

Claim 3 *For any two set maximal matches from z to X, $m = (i, j, s)$ and $n = (k, l, t)$, $i = k \iff j = l$.*

Proof. For two set maximal matches with the same starting position, if they have different ending positions, the smaller one is contained in the larger one and is not a set maximal match. Similarly, for two set maximal matches with the same ending position, if the have different starting positions, the smaller one is contained in the larger one and is not set maximal. Therefore, two set maximal matches have the same starting position if and only if they have the same ending position. □

Claim 4 *For every positional substring that is contained in z and a string in X, if it contains a full required region, it doesn't contains sites from any other required region.*

Proof. If there existed a positional substring m that fully contains i-th required region and a site from another required region, then m must contain sites from either the $i-1$-th or the $i+1$-th required region. If it contains sites from the $i-1$-th required region, we can show that these sites do not belong in the $i-1$-th required region by replacing $\mathcal{C}_l[i]$ with m and removing any overlap of $\mathcal{C}_l[i-1]$ and $\mathcal{C}_l[i]$ from $\mathcal{C}_l[i-1]$ (for leftmost MPSC of z by X, $\mathcal{C}_l$).

Similarly, if m contains sites from the $i+1$-th required region, we can show that they are not required by replacing $\mathcal{C}_r[i]$ with m and removing any overlap between $\mathcal{C}_r[i]$ and $\mathcal{C}_r[i+1]$ from $\mathcal{C}_r[i+1]$ (for rightmost MPSC of z by X, $\mathcal{C}_r$). These are contradictions, therefore no such positional substring m exists. □

Lemma 2. *The set of nodes $R_i[j]$ has an edge to is a subset of the set of nodes $R_i[j+1]$ has an edge to.*

Proof. All of the nodes $R_i[j]$ has an edge to are in R_{i+1}. The set maximal match of every node in R_{i+1} starts after the i-th required region by Claim 4. $R_i[j]$ ends before $R_i[j+1]$, and both start at or before the start of the i-th required region. Therefore, every set maximal match in R_{i+1} that overlaps or is directly after $R_i[j]$ overlaps $R_i[j+1]$. Therefore, the set of nodes $R_i[j]$ has an edge to is a subset of the set of nodes $R_i[j+1]$ has an edge to. □

Lemma 3. *Every positional substring in any Length Maximal MPSC of z by X is a set maximal match from z to X.*

Proof. Suppose there existed a length maximal MPSC of z by X, $\mathcal{C}$, that contained a positional substring that is not a set maximal match. Then, replacing it with a positional substring that is a set maximal match that contains it yields another MPSC $\mathcal{C}'$ with a larger length. Such a positional substring always exists by Claim 2. $\mathcal{C}'$ is an MPSC because it covers all the sites $\mathcal{C}$ covered and has the same size. Its length is larger because the only difference between $\mathcal{C}$ and $\mathcal{C}'$ is a positional substring in $\mathcal{C}$ was replaced by a larger one $\mathcal{C}$. This contradicts the fact $\mathcal{C}$ is a length maximal MPSC of z by X. Therefore, every length maximal MPSC of z by X is composed of only set maximal matches. □

Length Maximal MPSC Algorithm

Two versions of the LongestPaths subroutine are provided. The LongestPaths subroutine calculates the length of the longest path from each node to t in the graph. There is an in-place and out-of-place version of the subroutine. The out-of-place version is provided because it is easier to understand. Both versions have time complexity $O(S) \subseteq O(N)$.

Algorithm 1: Output a Length Maximal MPSC of z by X given a PBWT of X

```
t, d_min = VirtualInsert(z, PBWT);
C_l = LeftmostMPSC(d_min, z);
if |C_l| = 0 then
    output "No MPSC exists";
    return;
C_r = RightmostMPSC(d_min, z);
S = SetMaximalMatchPositions(d_min, z);
L = LongestPaths(S, C_l, C_r);
C = Backtracking(L, S);
return C;
```

Algorithm 2: VirtualInsert: Calculate Prefix and Divergence values of query z in a PBWT

$t[0] = 0$;
for $j = 0 \to N - 1$ **do**
  if $t[j] \neq M$ **then**
    $t[j+1] = w[t[j]][j][z[j]]$;
  else
    $t[j+1] = w[M-1][j][z[j]]$;
    if $z[j] = x_{a[M-1][j]}[j]$ **then**
      $t[j+1]++$;
$d_z[N+1] = d_b[N+1] = d_{min}[N+1] = N$;
for $j = N \to 0$ **do**
  $d_z[j] = \min(d_z[j+1], j)$;
  $d_b[j] = \min(d_b[j+1], j)$;
  if $t[j] \neq 0$ **then**
    while $d_z[j] > 0$ *and* $z[d_z[j]-1] = x_{a[t[j]-1][j]}[d_z[j]-1]$ **do**
      $d_z[j]--$;
  if $t[j] \neq M$ **then**
    while $d_b[j] > 0$ *and* $z[d_b[j]-1] = x_{a[t[j]][j]}[d_b[j]-1]$ **do**
      $d_b[j]--$;
  $d_{min}[j] = \min(d_z[j], d_b[j])$;
return t, d_{min};

Algorithm 3: LeftmostMPSC: Output Leftmost MPSC of z by X

$k = d_{min}[N]$;
$k_{old} = N$;
while $k \neq k_{old}$ **do**
  $\mathcal{C}_l$.enqueue($(k, k_{old} - 1, z)$);
  $k_{old} = k$;
  $k = d_{min}[k]$;
if $k \neq 0$ **then**
  return $\emptyset$;
return $\mathcal{C}_l$;

Algorithm 4: RightmostMPSC: Output Rightmost MPSC of z by X

```
// Assumes a positional substring cover of z by X exists
j = N;
k = N;
while j > 0 do
    while d_min[k] < j do
        j − −;
        b[j] = k − 1;
    k − −;
i = 0;
while i ≠ N do
    C_r.push_back((i, b[i], z));
    i = b[i] + 1;
return C_r;
```

Algorithm 5: SetMaximalMatchPositions: Output positions of set maximal matches from z to X

```
// Assumes a positional substring cover of z by X exists
for i = 0 → N − 1 do
    if d_min[i] < d_min[i + 1] then
        S.push_back((d_min[i], i − 1, z));
return S;
```

Algorithm 6: Backtracking: Output Length Maximal MPSC by backtracking through graph

```
// Assumes a positional substring cover of z by X exists
C = {S[0]};
l = L[0];
s = |S[0]|;
for i = 1 → |S| − 1 do
    if L[i] = l − s then
        C = C ∪ S[i];
        l = L[i];
        s = |S[i]|;
return C;
```

Algorithm 7: LongestPaths: Calculate length of Longest Path from every node (out-of-place)

```
// Assumes a positional substring cover of z by X exists and |C_l| > 1
R_0[0] = 0;
R_{|C_l|-1}[0] = |S| - 1;
j = 1;
for i = |C_l| - 2 → 1 do
    reqReg = (π_2(C_r[i - 1]) + 1, π_1(C_l[i + 1] - 1, z);
    while π_2(S[j]) < π_2(reqReg) do
        j++;
    while π_1(S[j] ≤ π_1(reqReg) do
        R_i.push_back(j);
        j++;
for i = |C_l| - 2 → 1 do
    L[R_i[0]] = |S[R_i[0]]| + L[R_{i+1}[0]];
    l = 0;
    while l + 1 < |R_{i+1}| and π_2(S[R_i[0]]) ≥ π_1(S[R_{i+1}[l + 1]]) - 1 do
        L[R_i[0]] = max(L[R_i[0]], L[R_{i+1}[l + 1]] + |S[R_i[0]]|);
        l++;
    for k = 1 → |R_i| - 1 do
        L[R_i[k]] = L[R_i[k - 1]] - |S[R_i[k - 1]]| + |S[R_i[k]]|;
        while l + 1 < |R_{i+1}| and π_2(S[R_i[k]]) ≥ π_1(S[R_{i+1}[l + 1]]) - 1 do
            L[R_i[k]] = max(L[R_i[k]], L[R_{i+1}[l + 1]] + |S[R_i[k]]|);
            l++;
return L;
```

Algorithm 8: LongestPaths: Calculate length of Longest Path from every node (in-place)

```
// Assumes a positional substring cover of z by X exists and |C_l| > 1
j_old = k_old = |S| − 1;
L[j_old] = |S[j_old]|;
for i = |C_l| − 2 → 1 do
    reqReg = (π_2(C_r[i − 1]) + 1, π_1(C_l[i + 1] − 1, z);
    k = j_old − 1;
    while π_1(S[k]) > π_1(reqReg) do
        L[k] = −1;
        k − −;
    j = k;
    while π_2(S[j − 1]) ≥ π_2(reqReg) do
        j − −;
    j_next = j;
    L[j] = |S[j]| + L[j_old];
    j_old + +;
    while j_old ≤ k_old and π_2(S[k]) ≥ π_1(S[j_old]) − 1 do
        L[j] = max(L[j], |S[j]| + L[j_old]);
        j_old + +;
    j + +;
    while j ≤ k do
        L[j] = L[j − 1] − |S[j − 1]| + |S[j]|;
        while j_old ≤ k_old and π_2(S[j]) ≥ π_1(S[j_old]) − 1 do
            L[j] = max(L[j], |S[j]| + L[j_old]);
            j_old + +;
        j + +;
    j_old = j_next;
    k_old = k;
L[0] = |S[0]| + L[j_old];
j_old + +;
while j_old ≤ k_old and π_2(S[0]) ≥ π_1(S[j_old]) − 1 do
    L[0] = max(L[0], |S[0]| + L[j_old]);
    j_old + +;
return L;
```

Run Time Results

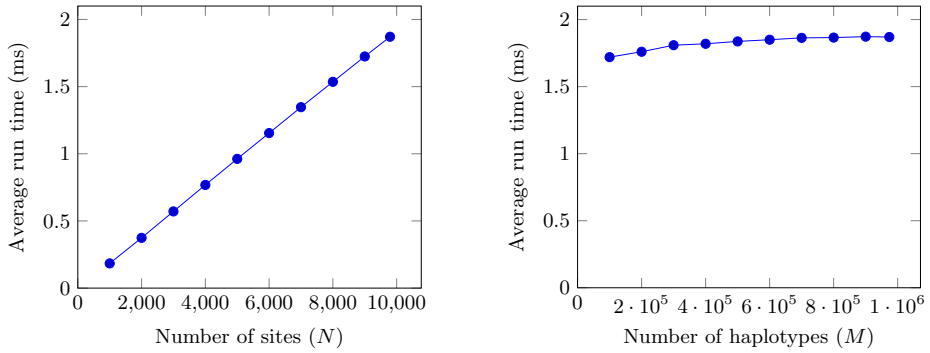

Fig. S1. Run time vs varying M and N. Average run time for the Length Maximal MPSC computation of 1,000 random haplotypes in the UKB British only dataset for varying number of sites, $N \in \{1{,}000, 2{,}000, 3{,}000, 4{,}000, 5{,}000, 6{,}000, 7{,}000, 8{,}000, 9{,}000, \text{and } 9{,}793\}$, is on the left. On the right, average run time for the Length Maximal MPSC computation of 1,000 random haplotypes in the UKB British only dataset for varying number of haplotypes in the reference panel, $M \in \{100{,}000, 200{,}000, 300{,}000, 400{,}000, 500{,}000, 600{,}000, 700{,}00, 800{,}000, 900{,}000, \text{and } 973{,}818\}$, is plotted.

Here, we estimate the speed difference between the MPSC formulation and Rosen and Patens' sublinear lazy LS solver. To do this, we use the results presented in their Fig. 8. They estimate that the lazy evaluation algorithm takes $M^{0.159}$ microseconds per site in the query haplotype to output a haplotype threading, where M is the number of haplotypes in the reference panel. Then, if their algorithm was run on our test case (UK Biobank Chromosome 21 with $1,000$ haplotypes removed). Their runtime is estimated to be $9,793 \times (973,818^{0.159}) = 87,717$ microseconds per haplotype. As can be seen in Fig. S1., the MPSC formulation outputs a haplotype threading in less than $2,000$ microseconds per haplotype ($1,871$ microseconds). The MPSC formulation is more than 45 times faster. Note that the time for the LS Lazy evaluation model does not include time taken to load the dataset into memory. Similarly, the time for the MPSC model does not include time taken to load the PBWT into memory.

h-MPSC

Algorithm 9: h-MPSC: Output h-MPSC of z by X in $O(N)$ time

```
i = f = 0;
g = M;
C = ∅;
for j = 0 → N do
    if j = N then
        f' = g' = 0;
    else
        if f ≠ M then
            f' = w[f][j][z[j]];
        else
            f' = w[M − 1][j][z[j]];
            if z[j] = x_{a[M−1][j]}[j] then
                f' + +;
        if g ≠ M then
            g' = w[g][j][z[j]];
        else
            g' = w[M − 1][j][z[j]];
            if z[j] = x_{a[M−1][j]}[j] then
                g' + +;
    if f' − g' < h then
        if i = j then
            output "No h-MPSC of z by X exists";
            return ∅;
        C = C ∪ {(i, j − 1, z)};
        i = j;
        if j ≠ N then
            j − − ;                          // to repeat current value of j in loop
        f = 0;
        g = M;
    else
        f = f';
        g = g';
return C;
```