2023

Algorithms and Variations on the Positional Burrows-Wheeler Transform and Their Applications

Ahsan Sanaullah
University of Central Florida

ALGORITHMS AND VARIATIONS ON THE POSITIONAL BURROWS-WHEELER TRANSFORM AND THEIR APPLICATIONS

by

AHSAN SANAULLAH
B.S. Florida Atlantic University, 2019

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2023

Major Professor: Shaojie Zhang

ABSTRACT

In this dissertation, we develop algorithms and variations on the Positional Burrows-Wheeler Transform (PBWT). The PBWT is a data structure that stores M binary strings of length N while allowing efficient search. We develop the dynamic-PBWT (d-PBWT). The d-PBWT is a variation of the PBWT that allows its relevant algorithms to run with unchanged time complexity, but also allows efficient insertion and deletion of haplotypes. We provide insertion and deletion algorithms on the PBWT with average case $O(N)$ time complexity. We also improve upon the query algorithms for the PBWT. Durbin described a set maximal match query algorithm on the PBWT and claimed $O(N)$ time complexity. Naseri et al. described a long match query algorithm on the PBWT using additional data structures (LEAP arrays) in claimed $O(N+c)$ time complexity, where c is the number of matches outputted. We showed these bounds to be incorrect in the worst case and provided set maximal match and long match query algorithms that do have these time complexities. Furthermore, we develop a new formulation of haplotype threading, the Minimal Positional Substring Cover (MPSC). We solve the MPSC in $O(N)$ time. Then, we solve variants of the MPSC problem: leftmost MPSC, rightmost MPSC, and set maximal match only MPSC. Using these variants to bound the solution space, we are able to represent all possible MPSCs in efficiently. Then, we solve variants that may be more biologically useful: length maximal MPSC, h-MPSC, and L-MPSC. All the MPSC problems are solved in $O(N)$ time given a PBWT of the reference panel. Finally, we show the biological usefulness of the MPSC formulation using an imputation benchmark.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Shaojie Zhang, for his guidance. I would also like to thank Dr. Degui Zhi for insightful questions. Thank you to Dr. Charlie Hughes and Dr. Shibu Yooseph for their helpful comments and questions.

Furthermore, this work was supported in part by the National Institutes of Health grants R01 HG010086 and R56 HG011509. It has also made use of the UK Biobank Resource under Application Number 24247.

Lastly, thank you to all who have collaborated with and supported me throughout my PhD including friends, family, and colleagues.

Chapters 1 to 3 are in part a reprint of "d-PBWT: dynamic positional Burrows–Wheeler transform," co-authored with Dr. Degui Zhi and Dr. Shaojie Zhang, which has been accepted by RECOMB 2020 [23] and published in Bioinformatics [24]. The dissertation author was the primary investigator and author of the paper.

Chapters 1, 2 and 4 are in part a reprint of "Haplotype threading using the positional Burrows-Wheeler transform," co-authored with Dr. Degui Zhi and Dr. Shaojie Zhang, which has been published in WABI 2022 Proceedings [22]. The dissertation author was the primary investigator and author of the paper.

Chapters 1, 2 and 4 are in part a reprint of "Minimal Positional Substring Cover: A Haplotype Threading Alternative to Li & Stephens Model," co-authored with Dr. Degui Zhi and Dr. Shaojie Zhang, which has been accepted by RECOMB 2023 [25]. A preprint [26] of the submitted manuscript is available at `https://doi.org/10.1101/2023.01.04.522803`. The dissertation author was the primary investigator and author of the paper.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Richard Durbin introduced the PBWT in 2014 [8]. The PBWT is an index on X, a set M binary strings of length N. When applied to haplotype datasets, X is sometimes referred to as the haplotype panel. The PBWT can be constructed in time linear to the size of the input panel, $O(MN)$. The PBWT focuses on matches between strings where the position of the match is the same in both strings. It can output all such matches between strings in the input panel in $O(MN+c)$ time where c is the number of matches outputted. This can be done for all long matches or all set maximal matches. These matches where position is the same in both strings are biologically useful since matches along long sections of the genome can be used to infer shared ancestry over that section of the genome. In fact the PBWT has been applied to many tasks in computational genetics. These methods can scale to biobank scale datasets due to their utilization of the PBWT.

The PBWT has been applied to haplotype threading, haplotype phasing, haplotype imputation, identical-by-descent (IBD) segment detection, and genome graphs. Haplotype threading is the process of representing a query haplotype as a sequence of copied segments from haplotypes in a reference panel. The Li & Stephens model has been the state of the art haplotype threading method for many years [12]. Haplotype threading has found applications in haplotype phasing and imputation. Haplotype phasing is the process of taking a genotype and inferring two haplotypes from it given a reference panel. A genotype can be seen as a string over an alphabet of size 3, at some variant, a value of 0 implies both haplotypes don't have the variant, 1 implies both haplotypes have the variant, and 2 implies one haplotype has the variant and one does not. Haplotype imputation is the inference of missing variants of a haplotype given some of its variants and a reference panel. Typically, microarray data of the haplotype to be imputed is provided and whole genome sequencing data is being imputed. The PBWT has been applied to haplotype phasing in widely used tools such as Beagle 5.2 [3] and Eagle2 [13]. Similarly, it has seen use in haplotype impu-

tation tools such as Beagle 5.2, IMPUTE5 [21], and SHAPEIT4 [7]. Identical-by-Descent (IBD) segments are segments of two haplotypes that match exactly and are at the same position on the genome. Furthermore, they match due to a shared ancestor through which the haplotypes obtained this segment. Many tools attempt to infer IBD segments through the use of exact or inexact match, typically using the PBWT. Some examples are RaPID [18], hap-IBD [29], and templated PBWT (TPBWT) [10]. Lastly, genome graphs have been proposed as an alternative to a single reference genome to more completely represent the possible variation in the human genome [9]. Variations of the PBWT have been built for this purpose, see the gPBWT and gBWT [19, 27]. In this dissertation, we develop a variation of the PBWT, d-PBWT, and algorithms related to IBD detection, haplotype threading, haplotype phasing, and haplotype imputation.

The PBWT as described by the original paper relies on arrays as the underlying data structure. However, arrays do not support efficient dynamic updates. If new haplotypes are to be added to, or some haplotypes are to be deleted from an existing PBWT structure, one has to rebuild the entire PBWT, an expensive effort linear to the number of haplotypes. This will be inefficient for large databases hosting millions of haplotypes as they may face constant update requests per changing consent of data donors in biobanks or customer growth in direct-to-consumer genetic companies. Moreover, lack of dynamic updates limits PBWT to be applied to large-scale genotype imputation and phasing, which typically go through the panel multiple times and update individual haplotypes in turn. It is much more efficient to allow updating the PBWT with an individual's new haplotypes while keeping others intact.

The classic Li and Stephens model parameterized haplotype threading as a hidden Markov model which can take into account the uncertainties regarding mismatches (emission probabilities) and template switching (transition probabilities) [12]. This model is quite sufficient for moderate sample sizes (hundreds to thousands) as it scales linearly with sample size. As a result, this model has served as a foundation for haplotype phasing and genotype imputation for the past two decades.

However, in the biobank era when the panel size is large, the standard Li and Stephens model may not be efficient enough.

The PBWT has been leveraged for speeding up the Li and Stephens model. Lunter proposed a representation of the PBWT using the BWT [14]. On this representation, they perform a search for a maximum likelihood Viterbi path through the Li and Stephens Hidden Markov Model. While their algorithm can compute the optimal score in $O(N)$ time, outputting of the haplotype threading requires $O(N \log M)$ time. Rubinacci et al. use the PBWT to select closely related individuals efficiently. These closely related individuals are then imputed using IMPUTE5, an imputation based on the Li and Stephens model [21]. Loh et al. use the PBWT similarly for EAGLE2, their phasing algorithm. They use the PBWT to obtain representative data of a haplotype and then thread it using a haplotype copying model similar to the Li and Stephens model [13]. Lastly, Delanueau et al. use the PBWT to speed up phasing through the Li and Stephens model among other methodologies in their phasing method SHAPEIT4 [7].

However, these algorithms [7, 13, 14, 21] are designed within the Li and Stephens HMM framework and the PBWT is used as a subroutine. These algorithms mostly focus on the Viterbi path that gives the maximum likelihood solution. We argue that in the biobank-scale panel, a query may have a large number of high-quality matches, and thus outputting the single best Viterbi solution may not be informative to reveal the overall high probability possible paths. We formulate the haplotype threading problem as a combinatorial optimization problem: given a set of haplotypes X and a query z, represent z as segments of haplotypes in X and optimize a certain objective scoring function. There are a number of possible scoring functions for threading, however there are common themes between them. Usually, one wants to represent z using a small amount of haplotypes in X or a small amount of distinct segments. In this dissertation, we minimize the number of segments we use to represent z. We formulate the Minimum Positional Substring Cover problem (MPSC), given a query z and a set of strings X, find a smallest set of positional substrings

contained in z and a string in X that cover all characters of z. While this formulation simplifies the original haplotype threading by ignoring potential mismatches in the Li and Stephens model flavor, it enables efficient enumeration of all possible solutions, leveraging the structure of PBWT. Augmenting our algorithms with mismatch-tolerating methods such as random projection [18] or PBWT-smoothing [28], our formulation can capture the bulk of the high-probability threading paths, and thus provide flexibility for designing variations of downstream tasks such as genotype imputation and haplotype phasing.

In Chapter 3, we develop query algorithms on the PBWT and a variation of the PBWT, dynamic-PBWT (d-PBWT). The query algorithms output all matches between a query haplotype and haplotypes in the PBWT. Durbin described a set maximal match query algorithm and Naseri et al. described a long match query algorithm. They claimed $O(N)$ and $O(N+c)$ time complexity respectively, where c is the number of matches outputted. We show that these algorithms do not have these time complexities in the worst case. Furthermore, we provide query algorithms that do have this optimal time complexity in the worst case. (A set maximal match query in $O(N)$ time and a long match query algorithm with $O(N+c)$ time complexity). The d-PBWT is an extension of the PBWT. It has most of the same capabilities of the d-PBWT while allowing average case $O(N)$ insertion and deletion algorithms. We show the previous query algorithms run with the same time complexity on the d-PBWT. This data structure makes the maintenance of a large database of a dynamically changing set of strings feasible while allowing queries for aligned matches.

In Chapter 4, we develop a new formulation of haplotype threading. We describe and solve the Minimal Positional Substring Cover (MPSC) problem using the PBWT. The MPSC problem is, given a query string z and a set of strings X, find a smallest set of positional substrings that cover z and are in X. A positional substring is a substring with a position in a string. A set of positional substrings cover a string if every positional substring is contained in the string and every index of the string is contained in a positional substring in the set. A positional substring is contained in a

set of strings if it contained in any string in the set. We show that this problem can be solved in $O(N)$ time given a PBWT of X (where N is the length of z). We also introduce and solve variants of the problem: Leftmost MPSC, Rightmost MPSC, and Set Maximal Match only MPSCs. We also solve biologically useful variants of the MPSC problem: h-MPSC, Length Maximal MPSC, and L-MPSC. In the h-MPSC problem, every positional substring in the cover is contained in at least h strings in X. A Length Maximal MPSC is an MPSC with maximum total length out of all MPSCs. An L-MPSC is a smallest cover of the query haplotype with segments that are all at least length L. Lastly, we represent the solution space of all MPSCs in $O(N)$ time and space. We provide solutions to all of these algorithms in $O(N)$ time. Finally, we benchmark our results on real genetics data and test the MPSC's performance on haplotype imputation.

CHAPTER 2: POSITIONAL BURROWS-WHEELER TRANSFORM

The following is a review of Durbin's PBWT paper and notation [8]. PBWT is a data structure that groups similar strings by sorting the reverse prefixes at each length. Say we have a PBWT data structure of a set X of M haplotype sequences $x_i \in X, i \in \{0 \dots M-1\}$. Each sequence has N sites indexed by $k \in \{0 \dots N-1\}$, values at a site are 0 or 1, $x_i[k] \in \{0,1\}$. For some haplotype sequence s we use $s[k_1,k_2)$ to represent the substring of s beginning at k_1 and ending at k_2-1. The length of this substring is k_2-k_1. Sequences s and t have a *match* from k_1 to k_2 if $s[k_1,k_2) = t[k_1,k_2)$. This match is *locally maximal* if it can't be extended, i.e. $(s[k_1-1] \neq t[k_1-1]$ or $k_1 = 0)$ and $(s[k_2] \neq t[k_2]$ or $k_2 = N)$. A match is a *long match* if it is locally maximal and at least length L for some constant $L > 0$. A match is a *set maximal* match from a sequence s to X if it is locally maximal and there is no longer match between s and any other sequence from X that covers the matching region.

The *prefix array a* contains $N+1$ sorted orderings of the sequences, one for each $k \in \{0 \dots N\}$. The k-th sorted ordering is a_k, the ordering of a_k is based on the reversed prefixes $x[0,k)$, if the prefixes are the same they are ordered according to their index i in X. a_k can also be thought of as the sorted ordering of the reversed prefixes of length k. In any a_k, adjacent sequences are maximally matching until k. In the following, let y_i^k be the i-th sequence in a_k, $y_i^k = x_{a_k[i]}$. The *divergence array* keeps track of the start position of locally maximal matches ending at k between a sequence and the sequence above it in a_k, i.e., $d_k[i]$ is the smallest value j such that $y_i^k[j,k) = y_{i-1}^k[j,k)$. The *extension function* $w_k(i,h), h \in \{0,1\}$ gives the a_{k+1} index of the first sequence after $a_k[i]$ ($a_k[i]$

inclusive) that has h at site k, i.e.,

$$w_k(i,h) = g \text{ s.t. } a_{k+1}[g] = a_k[j] \text{ and}$$
$$j = \min\{i \leq o < N \mid y_o^k[k] = h\}.$$

In implementation, the extension function is fully specified by two arrays: u and v. $w_k(i,0)$ is stored at $u_k[i]$ and $w_k(i,1)$ is stored at $v_k[i]$.

$$w_k(i,h) = \begin{cases} u_k[i] & \text{if } h = 0 \\ v_k[i] & \text{otherwise } (h = 1). \end{cases}$$

v is redefined here to make the extension function more intuitive. It is changed from Durbin's original definition [8].

CHAPTER 3: DYNAMIC POSITIONAL BURROWS-WHEELER TRANSFORM

Introduction

The scalability of the PBWT has attracted the use of its basic concepts in many variations. Graph PBWT (gPBWT) represents the population haplotype data in a genome graph using positional prefix arrays, it allows multiple alleles and subhaplotype queries (read alignments) [19]. The graph BWT (gBWT) is a version of the gPBWT with new features such as merging two indexes and removing paths from the genome graph [27]. Lunter proposed a data structure that reduces PBWT to a special case of BWT [14]. However, these variants only use the positional prefix array to index multiple genomes. While this allows exploitation of the PBWT's strong compression and efficient subhaplotype queries, it ignores two of its important aspects: The divergence values and u and v pointers of the PBWT. These data structures enable additional capabilities including long match search and block search [1, 15, 17]. Furthermore, they allow the traversal of the PBWT through a query haplotype that is not even in the PBWT (out-of-sample query). Therefore, gPBWT, gBWT, and Lunter's extension of PBWT do not yet realize the full power of the PBWT.

The original PBWT paper described an array version of the PBWT, and a set of basic algorithms: Algorithms 1 and 2 for construction, Algorithms 3 and 4 for reporting all vs all long matches and set maximal matches, and Algorithm 5 for reporting set maximal matches between an out of panel query against a constructed PBWT panel. Recently, [17] presented a new algorithm, L-PBWT-Query, that reports all long matches between an out of panel query against a constructed PBWT panel in time complexity linear to the length of the haplotypes and constant to the size of the panel. They introduced Linked Equal/Alternating Positions (LEAP) arrays, an additional data structure

that allows direct jumping to boundaries of matching blocks. This algorithm offers efficient long matches, a more practical target for genealogical search. Arguably, L-PBWT-Query makes PBWT search more practical as it returns all long enough matches rather than merely the best matching ones. We believe that L-PBWT-Query represents a missing piece of the PBWT algorithms.

However, all above algorithms are based on arrays, which do not support dynamic updates. That means, if new haplotypes are to be added to, or some haplotypes are to be deleted from an existing PBWT structure, one has to rebuild the entire PBWT, an expensive effort linear to the number of haplotypes. This will be inefficient for large databases hosting millions of haplotypes as they may face constant update requests per changing consent of data donors in biobanks or customer growth in direct-to-consumer genetic companies. Moreover, lack of dynamic updates limits PBWT to be applied to large-scale genotype imputation and phasing, which typically go through the panel multiple times and update individual haplotypes in turn. It is much more efficient to allow updating the PBWT with an individual's new haplotypes while keeping others intact.

We introduce d-PBWT, a dynamic version of the PBWT data structure. We show that the d-PBWT allows efficient insertion and deletion of haplotypes. Furthermore, unlike other variants of the PBWT, including gPBWT, gBWT, and Lunter's proposed generalization, the d-PBWT keeps the divergence values and u and v pointers from the PBWT. The key difference between the PBWT and d-PBWT is that, at each position k, instead of keeping track of sequence order using an array, we use a linked list. The nodes of this linked list encapsulate all pointers needed for traversing PBWT data structures. Our main results are: we developed efficient insertion and deletion algorithms that dynamically update all d-PBWT data structures (Algorithms 1 and 2). In addition, we will show that the d-PBWT can do Durbin's Algorithms 1-5 and L-PBWT-Query with the same time complexity as the PBWT. While Durbin's Algorithm 5 and L-PBWT-Query are practically independent of the number of haplotypes in practice, we show that they are not in the worst case. We show search algorithms for set maximal and long matches with worst case linear time complexity,

Table 3.1: Algorithms on the PBWT and d-PBWT

	Structure	Functions	Algorithm	Time
Durbin [8]	PBWT	Construction	Alg. 1 & 2	$O(MN)$
		All vs. All LM & SMM	Alg. 3 & 4	$O(MN)$
		SMM Query	Alg. 5	$O(MN)^*$
Naseri et al. [17]		Long Match Query	Alg. 2	$O(MN)^*$
This Chapter	d-PBWT	Insertion	Alg. 1	Avg. $O(N)$
		Deletion	Alg. 2	Avg. $O(N)$
		SMM Query	Alg. 5	$O(N)$
		Long Match Query	Alg. 6	$O(N)$
		Long Match Query	Alg. 7	$O(MN)^*$
		Construction	-	$O(MN)$
		All vs. All LM & SMM	-	$O(MN)$
		Conversion with PBWT	Alg. 3 & 4	$O(MN)$

M is the number of sequences. N is the number of sites. Time complexities assume $|\text{output}| < N$. LM is long match. SMM is set maximal match. For construction of the d-PBWT, a modification of Durbin's Algorithm 2 should be used. For All vs. All long and set maximal matches, modifications of Durbin's Algorithms 3 and 4 should be used respectively. * In practice, the worst case behavior of these algorithms is almost never encountered. In practice, they usually behave like $O(N)$ time algorithms.

but requiring multiple passes (Algorithms 5 and 6), and one search algorithm for long matches with average case linear time complexity with a single pass (Algorithm 7). The long match query algorithms we describe do not require additional LEAP arrays data structures. These three new search algorithms can also be applied to the static PBWT. Table 3.1 summarizes the major contributions of this chapter.

d-PBWT

Our main observation is that PBWT algorithms are not necessarily array algorithms. The essence of PBWT is that, at each site, sequences are ordered by their reverse prefix, and the updates of the ordering across adjacent sites are tracked by pointers. However, the ordering of sequences is not

Table 3.2: PBWT and d-PBWT equivalencies

Entity	PBWT	d-PBWT
Column k	a_k, d_k, u_k, v_k	linked list, head $= (k,0)$
Index i at Column k	$a_k[i], d_k[i], u_k[i], v_k[i]$	(k,i)
Sequence name	$a_k[i]$	n.ID
Match length	$d_k[i]$	$n.d$
Previous sequence in k-th sorting	$(a,d,u,v)_k[i-1]$	$n.above$
Next sequence in k-th sorting	$(a,d,u,v)_k[i+1]$	$n.below$
Next site by extension function	$u_k[i]$, $v_k[i]$, $w_k(i,h)$	$n.u$, $n.v$, $n.w(h)$
Substring in original haplotype	$y_k[j,k)$, $x_{a_k[i]}[j,k)$	$n[j,k)$, $x_{n.\mathrm{ID}}[j,k)$

necessarily tracked by prefix arrays. This fact might not be obvious as the original BWT [4] was based on arrays and all Durbin's PBWT algorithms and previous PBWT algorithms are written in the array language. Here, we propose using a doubly linked list at each site to track the sorting. In doing so, we can enable PBWT for dynamic updates, while still maintaining all basic operations of PBWT. Below we formally describe the dynamic version of PBWT, d-PBWT, and all its algorithms.

Like PBWT, the d-PBWT consists of N columns[1], each corresponds to one site. While *column* is an array-biased term, we abuse it for convenience of corresponding back to an array-based PBWT. Column k is a doubly linked list of M nodes that represents the reverse prefix sorting of all M sequences at site k. A node n in column k is noted as (k,i) *iff* it takes i node traversals (using node.below) to reach n from the top node of column k. It turns out that we can encapsulate all necessary PBWT pointers at (k,i), including $a_k[i], d_k[i], u_k[i],$ and $v_k[i]$ inside individual nodes: A node n has one function, w, and six properties. The properties are *above, below,* ID*, d, u, and v*. $n.above$ represents $(k,i-1)$ and $n.below$ represents $(k,i+1)$. n.ID is an integer $\in \{0 \ldots M-1\}$ that is unique to the sequence n represents, i.e. n.ID is $a_k[i]$. $n[j,k)$ is equivalent to $y_i^k[j,k)$ and

[1]It is OK to use an array for indexing columns as long as the sites of a genome are stable. However, it may be possible to extend the columns to be non-linearly sorted, as in *variant graph* [11].

$x_{n.\mathrm{ID}}[j,k)$. $n.d$ is equivalent to $d_k[i]$, i.e., $n.d = \min\{0 \leq j \leq k \mid n[j,k) = n.above[j,k)\}$. Each node also has u and v pointers that make up the extension function, these are equivalent to the u and v arrays as well. This means that they point to the node in the next column of the first sequence below them (self included) that has 0 (for u) or 1 (for v). $n.w(h)$ gets/sets $n.u$ if $h = 0$, otherwise $n.v$. Lastly, the haplotype panel of d-PBWT is a dynamic array of M haplotypes. The equivalencies between data structures of PBWT and d-PBWT are summarized in Table 3.2.

Insertion

The insertion algorithm inserts a new haplotype z into the d-PBWT. It works by first inserting the nodes of z in the correct position in each column and then calculating the divergence values after. This is analogous to first updating the prefix arrays and then updating the divergence arrays. This is done by first sweeping forwards through the data to insert the nodes, and then sweeping backwards through the data to calculate the divergence values. z is inserted into the dynamic haplotype panel in the forward sweep.

We update the prefix panel by keeping track of the node that z should be above and then inserting z above said node. We define t_k as the node that z should be above in column k. If we have t_k, then we can get t_{k+1} using the extension function. The sequence that will be below z at column $k+1$ is the first sequence below z (not inclusive) that has the same value as z at k, i.e., $t_{k+1} = t_k.w(z[k])$. We can use this to calculate all t_ks and insert z above them. See Fig. 3.1.

We also have to maintain the u and v pointers. The contiguous group of sequences directly above z at k that have the opposite of $z[k]$ at k need to have their corresponding pointer updated to point to z_{k+1}. If $z[k]$ is 0, the pointer to be updated is u, otherwise the pointer to be updated is v. Haplotypes that have similar prefixes will have the same allele at the next site with high probability. We expect that the number of haplotypes above the inserted haplotype with the opposite allele value

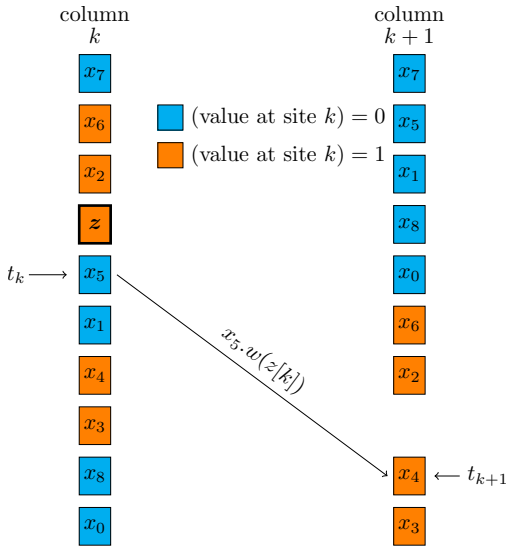

Figure 3.1: Finding the insertion location of z at next column. The first sequence below x_5 (also known as t_k) that has the same value as z at k is x_4. $t_k.w(z[k])$ points to the $k+1$ node of the first sequence below t_k (t_k inclusive) that has the same value as z at k, which is the $k+1$ node of x_4.

will usually be zero, and is probably a small constant on average. Our testing has shown that in the UK Biobank Chromosome 21 data, the average number of u and v pointers updated per column is 1, see Fig. 3.6. Therefore, the number of pointers updated per site is a small constant on average. Furthermore, $z_k.w(z[k])$ is equal to z_{k+1} and $z_k.w(\text{opposite of } z[k])$ is equivalent to $t_k.w(\text{opposite of } z[k])$ (z_k is the node of z in column k). Therefore u and v pointers of column k are updated after insertion of z_{k+1} into column $k+1$. See Fig. 3.2.

The only thing left to do is update the divergence values. For each column k, only 2 divergence values need to be set, the divergence of z and the divergence of $z.below$, all other divergence values remain unchanged because the sequence above all other sequences remain unchanged. We will update the divergence values by going backwards through the columns and keeping track of the minimum divergence value (longest match) found so far. A key observation is that at any column k, the divergence value of z must be at least the divergence value of z at $k+1$, i.e., $z_k.d \leq z_{k+1}.d$.

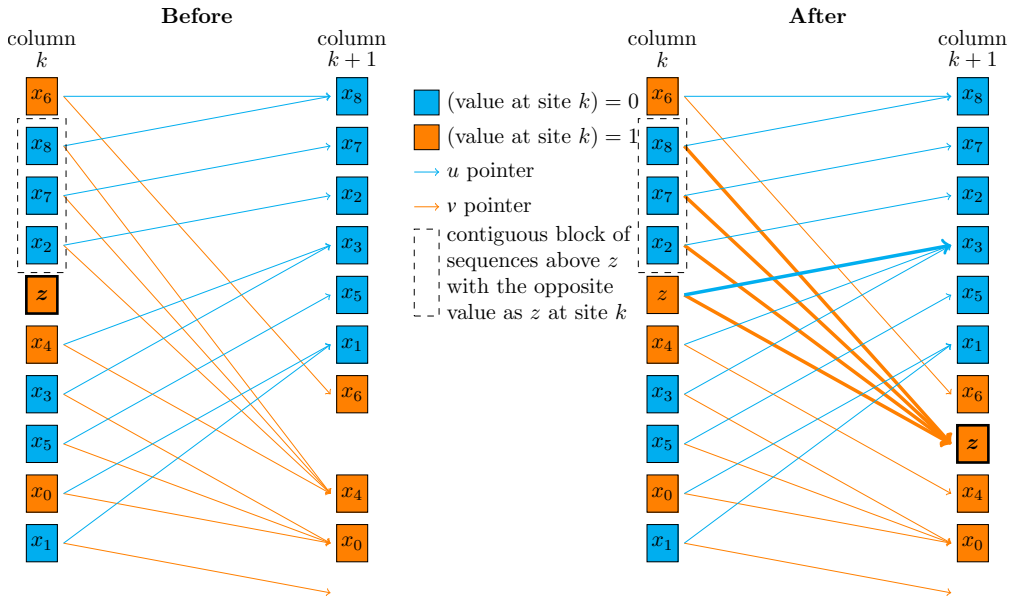

Figure 3.2: Updating the u and v pointers when inserting the z node into column $k+1$. Updated items are bold, including z, four v pointers, and one u pointer. It also shows the update of the $w.(\text{opposite of } z[k])$ pointers of the contiguous block of sequences above z with the opposite value as z at site k.

This is true because if the sequence above z at $k+1$ matches with z r sites backwards from $k+1$, then that sequence will be above z at k and the sites will still match. The same goes for z and $z.below$. See Lemma 1 below.

Lemma 1. $z_k.d \leq z_{k+1}.d$ ***and*** $z_k.below.d \leq z_{k+1}.below.d$

Proof. If $z_{k+1}.d > k$, then $z_k.d \leq z_{k+1}.d$ because $z_k.d \leq k$. Same for $z_{k+1}.below.d$ and $z_k.below.d$.

The relative order of sequences that have the same value at site k is the same in column k and $k+1$. If $z_{k+1}.d \leq k$, then $z_k[k] = z_{k+1}.above[k]$. Therefore the relative order of z_{k+1} and $z_{k+1}.above$ is the same in column k as it was in $k+1$, i.e., $z_k.above$ is somewhere above z_k. If $z_{k+1}.below.d \leq k$, then $z_k[k] = z_{k+1}.below[k]$. Therefore the relative order of z_{k+1} and $z_{k+1}.below$ is the same in column k as it was in $k+1$, i.e., $z_k.below$ is somewhere below z_k.

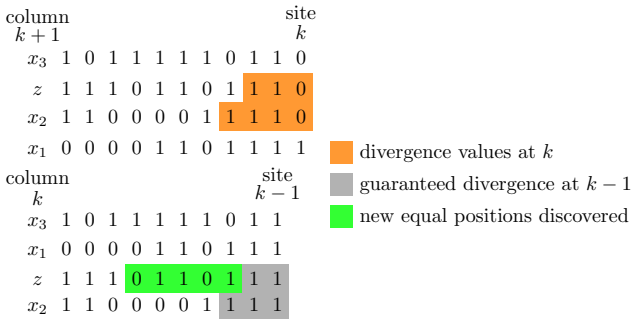

Figure 3.3: Updating the divergence value of z and $z.below$ at position k based on position $k+1$. At column k we know that there is some sequence above z that matches until the divergence value of z in column $k+1$. This is because if the sequence is above z in column $k+1$ and it matches at site k, then it is above z in column k. The relative order of sequences that have the same value at site k is the same in columns k and $k+1$. The same goes for $z.below$ and the divergence value of $z.below$.

If there is a sequence above z_k that matches longer than $z_{k+1}.above$, it will be directly above z and $z_k.d < z_{k+1}.d$. If there is no sequence above z_k that matches longer than $z_{k+1}.above$, then $z_{k+1}.above$ will be directly above z_k and $z_k.d = z_{k+1}.d$. Therefore $z_k.d \leq z_{k+1}.d$ If there is a sequence below z_k that matches longer than $z_{k+1}.below$, it will be directly below z and $z_k.d < z_{k+1}.d$. If there is no sequence below z_k that matches longer than $z_{k+1}.below$, then $z_{k+1}.below$ will be directly below z_k and $z_k.below.d = z_{k+1}.below.d$. Therefore $z_k.below.d \leq z_{k+1}.below.d$ □

Therefore, to calculate the divergence values we go from $k = N \rightarrow 1$ keeping track of divergence value of previous column. The divergence value of $z_k.below$ and z_k is calculated by decrementing from divergence of $z_{k+1}.below$ and z_{k+1} until the first site that is different is found. See Fig. 3.3.

The time complexity of the Insertion algorithm (pseudocode in Algorithm 1) is average case $O(N)$. This is average case instead of worst case solely because of updating the u and v pointers and insertion of z into the dynamic haplotype panel. However, as stated, because of *linkage disequilibrium*, a case where the constant is non-negligible is expected to be rare. Insertion of z into dynamic

Algorithm 1: Insertion: Insert new sequence z into d-PBWT

```
/* insert into linked list without divergence values                    */
```
$t_0 =$ last node of column 0;
z_0.ID $= M$;
`// update above/below pointers accordingly`
insert z_0 above t_0;
for $k = 0 \rightarrow N-1$ **do**
    make new node z_{k+1};
    $temp = z_k.above$;
    while $temp[k] \neq z[k]$ **do**
        $temp.w(z[k]) = z_{k+1}$;
        $temp = temp.above$;
    $z_k.w(z[k]) = z_{k+1}$;
    $z_k.w($opposite of $z[k]) = z_k.below.w($opposite of $z[k])$;
    $t_{k+1} = t_k.w(z[k])$;
    z_{k+1}.ID $= M$;
    `// update above/below pointers accordingly`
    insert z_{k+1} above t_{k+1};
    `// update dynamic haplotype panel`
    $x_M[k] = z[k]$;
`// calculate divergence values`
$z_{dtemp} = N$;
$b_{dtemp} = N$;
for $k = N \rightarrow 0$ **do**
    $z_{dtemp} = \min(z_{dtemp}, k)$;
    $b_{dtemp} = \min(b_{dtemp}, k)$;
    while $z_k[z_{dtemp}-1] = z_k.above[z_{dtemp}-1]$ **do**
        $z_{dtemp}--$;
    while $z_k[b_{dtemp}-1] = z_k.below[b_{dtemp}-1]$ **do**
        $b_{dtemp}--$;
    $z_k.d = z_{dtemp}$;
    $z_k.below.d = b_{dtemp}$;
$M++$;

haplotype panel is amortized $O(N)$, therefore it is average case $O(N)$. The insertion of the nodes of z into the correct position of the panel is worst case $O(N)$ because insertion into one column is constant time and there are N columns inserted into. The divergence calculation is also worst case $O(N)$ because the outer loop runs for N iterations and the sum of all iterations of the inner loop will

be at most N. The sum of all iterations of the inner loop will be at most N because it decrements an index from $N \rightarrow 0$ over the whole algorithm. The fact that a "virtual insertion" algorithm (i.e. find all divergence and t_k values without updating u and v pointers or inserting z) is worst case $O(N)$ will be used later to show the time complexity of the query algorithms.

Deletion

Algorithm 2: Deletion: Delete sequence x_i from d-PBWT

s_0 = first node of sequence x_i;
l_0 = first node of sequence x_{M-1};
for $k = 0 \rightarrow N-1$ **do**
    $s_{k+1} = s_k.w(s[k])$;
    $l_{k+1} = l_k.w(l[k])$;
    $temp = s_k.above$;
    while $temp[k] \neq s[k]$ **do**
        $temp.w(s[k]) = s_k.below.w(s[k])$;
        $temp = temp.above$;
    l_k.ID = i;
    $s_k.below.d = max(s_k.d, s_k.below.d)$;
    $x_i[k] = x_{M-1}[k]$; `// update dynamic haplotype panel`
    delete s_k from linked list;
l_N.ID= i;
$s_N.below.d = max(s_N.d, s_N.below.d)$;
delete s_N from linked list;
$M--$;

Deletion of a sequence from the d-PBWT is easy. If sequence i is to be deleted, sequence x_{M-1} needs to have the ID of all its nodes changed from $M-1$ to i so that the ID definition is maintained after deletion of x_i. Furthermore, an array of pointers to the node in column 0 of each node needs to be kept so that the node of x_i in column 0 can be accessed in constant time. Maintenance of this array is just an amortized constant time operation in the insertion and deletion algorithms. The contiguous block of sequences above the node of x_i in column k needs to have their $\{u$ if $x_i[k] = 1, v$ otherwise$\}$ pointers updated. They are set to value of the corresponding pointer of the node

below x_i's node. Lastly, the node of x_i in each column is deleted and the divergence of the node below it is updated. The whole algorithm can be done in one sweep. The time complexity of this algorithm is average case $O(N)$. This is not worst case because of the update of the u and v pointers and the haplotype panel. However, as stated before, update of the dynamic haplotype panel is amortized $O(N)$ and the number of u and v pointers updated per column will be a small constant on average. See Algorithm 2.

Equivalencies and Conversion

Algorithm 3: Conversion1: Converts d-PBWT into an array PBWT

Make arrays $a_{N,M}, d_{N,M}, u_{N,M}$, and $v_{N,M}$;
for $j = 0 \rightarrow N$ **do**
    $temp =$ top node of column j;
    for $i = 0 \rightarrow M-1$ **do**
        $a_j[i] = temp.\text{ID}$;
        $d_j[i] = temp.d$;
        $u_j[i] = temp.u.\text{ID}$;
        $v_j[i] = temp.v.\text{ID}$;
        $temp = temp.below$;
// Convert dynamic array haplotype panel into regular array haplotype panel

Equivalencies between data structures and d-PBWT (Table 3.2) suggest that all construction algorithms and search algorithms can be translated between PBWT and d-PBWT with minimal changes. Moreover, a d-PBWT data structure can be initialized by direct bulk conversion from an existing PBWT. Conversion of the d-PBWT to a PBWT in $O(MN)$ time is trivial given its description. So is conversion of a PBWT to a d-PBWT. See Algorithms 3 and 4.

Durbin's Algorithms 1-5 can be implemented on the d-PBWT with a little modification. Furthermore, the pseudocode of our query algorithms are presented in the notation of d-PBWT, however, they can easily be applied to PBWT. The time complexity of these algorithms are the same on both

Algorithm 4: Conversion2: Converts PBWT into d-PBWT

```
// make column N, store array indexed by i with pointer to node of x_i in
   column N
for j = N − 1 → 0 do
    new node n;
    top node of column j = n;
    for i = 0 → M − 1 do
        n.ID = a_j[i];
        n.d = d_j[i];
        n.u = nodePointer[u_j[i]];
        n.v = nodePointer[v_j[i]];
        newnodePointer[n.ID] = n;
        above = n;
        new node n;
        above.below = n;
        n.above = above;
    nodePointer = newnodePointer;
// Convert array haplotype panel into dynamic array haplotype panel
```

data structures.

Query Algorithms

Time Complexity of Durbin's Algorithm 5

A set maximal match query takes a new sequence z and outputs all set maximal matches from the panel to z. Durbin has presented an algorithm to solve this problem, he claims his algorithm runs in $O(N)$ time complexity [8]. However, Durbin's Algorithm 5 is not worst case $O(N)$.

Durbin claims "*The while loop in f' or g' is inevitable because it only takes as many iterations as there are matches to report the next time $f' = g'$*" [8]. However, this is false. Fig. 3.4 shows a possible haplotype panel that causes Durbin's algorithm to run in $\omega(N + |output|)$ time. All sequences between x_2 and x_{M-3} match with x_0 at $[0, 15)$. f' is the first sequence in the block

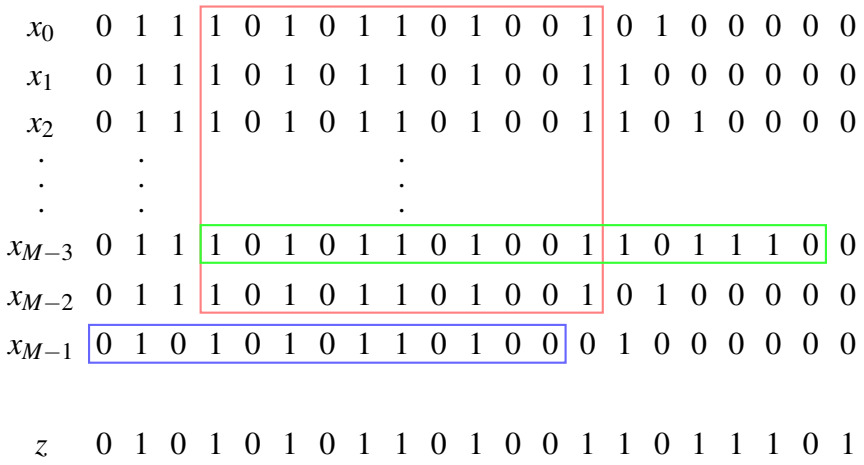

Figure 3.4: An example haplotype panel and z that causes Durbin's algorithm 5 to run in $\omega(N+c)$ time.

of local maximally matching blocks. g' is the first sequence below the block of local maximally matching blocks. When $f' = g'$, f' and g' are expanded one by one to include every sequence local maximally matching at $k+1$.

At $k = 13$, Durbin's Algorithm 5 will output x_{M-1} as a set maximal match at $[0,13)$ and the f' and g' loops will be entered to find the new block. The new block will have $M-1$ sequences in it $\{x_0 \rightarrow x_{M-2}\}$. However, only one of these sequences will be outputted as a set maximal match (x_{M-3} at $[3,20)$). Therefore, the number of times the f' and g' while loops are entered is not bound by c (number of matches) and Algorithm 5 is not $O(N+|output|)$. Of course, we have empirical evidence of the roughly $O(N+|output|)$ performance in practice of Durbin's Algorithm 5 [17].

Set Maximal Match Query

Here we show a new algorithm for outputting set maximal matches from z to X with a worst-case $O(N)$ time complexity. The set maximal match query algorithm begins by virtually inserting z into

the d-PBWT. The sweep back of the virtual insertion algorithm is modified so that set maximal matches are simultaneously outputted. The set maximal match query is fairly straightforward after one vital element is understood. If z's locally maximal match ending at k matches farther back than its locally maximal match ending at $k+1$, then z's locally maximal match ending at k is a set maximal match. Therefore, we can just compare divergence values at k and $k+1$ when calculating them to find and output set maximal matches.

Lemma 2. $\min(z_k.d, belowz_k.d) < \min(z_{k+1}.d, belowz_{k+1}.d)$ *iff z's locally maximal matches ending at k are set maximal.*

Proof. Assume $\min(z_k.d, belowz_k.d) < \min(z_{k+1}.d, belowz_{k+1}.d)$ and

$$\exists \text{ sequence } s \text{ and } d_1 < \min(z_k.d, belowz_k.d) \text{ s.t. } s[d_1,k) = z[d_1,k).$$

Then the local maximally matching sequence to z is not adjacent to it at k or the divergence values are incorrect (contradiction). Therefore there does not exist a sequence that has a match with z that extends this match to the left.

Assume $\min(z_k.d, belowz_k.d) < \min(z_{k+1}.d, belowz_{k+1}.d)$ and

$$\exists \text{ sequence } s \text{ and } k_1 > k \text{ s.t. } s[\min(z_k.d, belowz_k.d), k_1) = z[\min(z_k.d, belowz_k.d), k_1).$$

Then $\min(z_k.d, belowz_k.d) = \min(z_{k+1}.d, belowz_{k+1}.d)$ (contradiction). Therefore there does not exist a sequence that has a match with z that extends this match to the right.

Therefore there is no sequence that can extend this match and this match is locally maximal. So $\min(z_k.d, belowz_k.d) < \min(z_{k+1}.d, belowz_{k+1}.d) \implies z$'s locally maximal matches ending at k are set maximal.

Assume z has a set maximal match at $[d_2, k)$ and $d_2 = \min(z_k.d, belowz_k.d) = \min(belowz_{k+1}.d, z_{k+1}.d)$. Then there is a match at $[d_2, k+1)$ and the $[d_2, k)$ match can be extended. Therefore it is not set maximal (contradiction). We have already shown that the divergence at $k \leq$ divergence at $k+1$ in Lemma 1. Therefore z's locally maximal matches ending at k are set maximal $\implies \min(z_k.d, belowz_k.d) < \min(z_{k+1}.d, belowz_{k+1}.d)$.

Therefore $\min(z_k.d, belowz_k.d) < \min(z_{k+1}.d, belowz_{k+1}.d) \iff z$'s locally maximal matches ending at k are set maximal. □

Intuitively, a match is a set maximal match if the match is locally maximal and there is no match with z that encompasses this match. We know that the match is locally maximal because we defined it as "locally maximal match ending at k" and it ends at k, therefore it is locally maximal (we know it ends at k because if it didn't the locally maximal matches of k and $k+1$ would match to the same point). Lastly, if there was a match with z that encompassed this match, then the locally maximal matches of k and $k+1$ would match to the same point. Therefore, the z's locally maximal match ending at k is a set maximal match and can be outputted. Furthermore, there might be multiple sequences with this match, this is easily checked with divergence values.

Lastly, since the sequence above and below z can't match z with the same divergence, locally maximal matches will either be all above or all below, therefore only the direction with the smaller divergence value (longer match) will be checked. Assume the sequence above and below z in the sort order match z with the same divergence, the sequence above has value 0 one position behind and the sequence below has value 1, z must have either 0 or 1 at this position, Therefore the sequences above and below z do not match z with the same divergence.

The time complexity of the Set Maximal Match Query algorithm (Algorithm 5) is worst case $O(N + |output|)$. The virtual insertion is $O(N)$ because the haplotype panel and the u and v pointers

Algorithm 5: Set Maximal Match Query: Find set maximal matches from z to sequences in the d-PBWT

```
// Find all t_k s
t_0 = last node of column 0;
for k = 0 → N − 1 do
    t_{k+1} = t_k.w(z[k]);
// calculate divergence values and output set-maximal matches
z_dtemp = N;
b_dtemp = N;
for k = N → 0 do
    z_dtemp = min(z_dtemp, k);
    b_dtemp = min(b_dtemp, k);
    while z_k[z_dtemp − 1] = t_k.above[z_dtemp − 1] do
        z_dtemp − −;
    while z_k[b_dtemp − 1] = t_k[b_dtemp − 1] do
        b_dtemp − −;
    z_k.d = z_dtemp;
    belowz_k.d = b_dtemp;
    if min(z_k.d, belowz_k.d) < min(z_{k+1}.d, belowz_{k+1}.d) then
        if z_k.d < belowz_k.d then
            output set maximal match from z.ID to t_k.above.ID at [z_k.d, k);
            temp = t_k.above;
            while temp.d ≤ z_k.d do
                temp = temp.above;
                output set maximal match from z.ID to temp.ID at [z_k.d, k);
        else
            output set maximal match from z.ID to t_k.ID at [belowz_k.d, k);
            temp = t_k.below;
            while temp.d ≤ belowz_k.d do
                output set maximal match from z.ID to temp.ID at [belowz_k.d, k);
                temp = temp.below;
```

are not updated. The while loops are only entered when there is a set maximal match to output and each match is outputted exactly once. Therefore the sum of iterations of the output while loop is bounded by the number of matches found and the whole algorithm is $O(N + |output|)$.

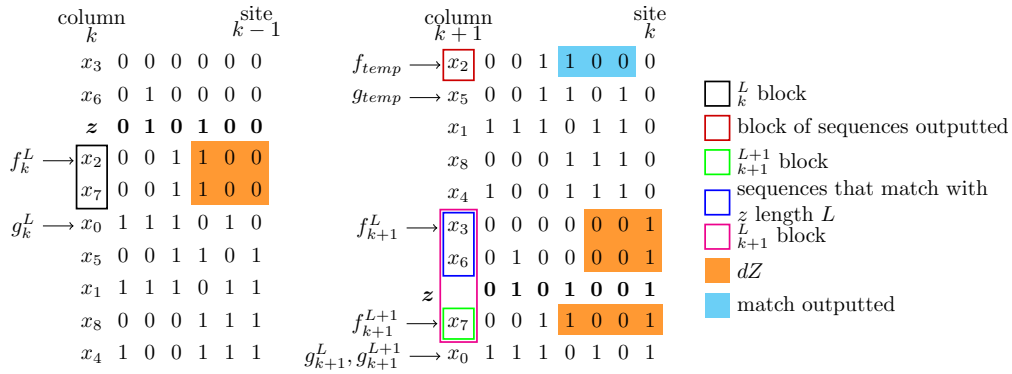

Figure 3.5: Computation of $\{f^L_{k+1}, g^L_{k+1}\}$ using $\{f^L_k, g^L_k\}$ and extension function. $L = 3$. $f^L_k.w(z[k])$ and $g^L_k.w(z[k])$ gives f^{L+1}_{k+1} and g^{L+1}_{k+1}. From there, the boundaries are expanded to include sequences that match with z length L until $k+1$. The new boundaries are f^L_{k+1} and g^L_{k+1}. At the same time, $f^L_k.w(\text{opposite of } z[k])$ and $g^L_k.w(\text{opposite of } z[k])$ is used to get f_{temp} and g_{temp}. These mark the block of sequences that match with z length L or longer and the match ends at site k.

Long Match Query

A long match query takes a query haplotype z and outputs all locally maximal matches between the panel and z that are length L or longer for some constant $L > 0$. Naseri et al. first proposed an efficient algorithm (L-PBWT-Query) to find all long matches between a query haplotype z and a database of haplotypes X in $O(N + |out\,put|)$ time in practice by using PBWT and LEAP arrays to skip unnecessary checks [17]. The time complexity of L-PBWT-Query is not truly $O(N + |out\,put|)$ in the worst case because it relies on Durbin's Algorithm 5 as a subroutine. Here we propose a new algorithm for finding long matches without using LEAP arrays in worst case $O(N + |out\,put|)$ time.

We will need the divergence values for our query algorithm, therefore the first thing we do is virtually insert z into the data structure. This means we get all the t_k values and all the new divergence values if z was inserted. Then we do a third sweep of the data while keeping track of a matching block. Note that we don't update the haplotype panel or u and v pointers.

The high level idea of this algorithm is to keep track of the block of sequences that match with z length L or longer until k. We will denote the boundaries of this block f_k^L and g_k^L, f_k^L points to the first sequence in the block and g_k^L points to the first sequence below f_k^L not in the block. Note that the definition of f^L and g^L is different from Durbin's f and g.

Given f_k^L and g_k^L, we want to get f_{k+1}^L and g_{k+1}^L. First we use the extension function. $f_k^L.w(z[k])$ will give us the position in column $k+1$ of the first sequence after f_k^L that has the same value as z at k. Likewise with $g_k^L.w(z[k])$. Therefore, $f_k^L.w(z[k])$ and $g_k^L.w(z[k])$ will give us f_{k+1}^{L+1} and g_{k+1}^{L+1} (see Lemma 3 for a proof). f_{k+1}^{L+1} and g_{k+1}^{L+1} mark the boundaries of the block of sequences that match with z length $L+1$ or longer until $k+1$. The difference between $\{f_{k+1}^{L+1}, g_{k+1}^{L+1}\}$ and $\{f_{k+1}^L, g_{k+1}^L\}$ is only the sequences that match with z length L until $k+1$. Therefore, we can intuitively use the divergence values to check if a sequence on the boundary matches with z length L, if it does, we move the boundary to include it in the block. After both boundaries reach a sequence that doesn't match with z length L until $k+1$, we have found f_{k+1}^L and g_{k+1}^L.

Lemma 3. *$f_k^L.w(z[k]) = f_{k+1}^{L+1}$ and $g_k^L.w(z[k]) = g_{k+1}^{L+1}$*

Proof. All the sequences that match with z length $L+1$ or longer until $k+1$ all match with z length L or longer until k,i.e., the set of sequences that match with z length $L+1$ until $k+1$ is a subset of the set of sequences that match with z length L or longer until k. Specifically, it is the subset that has the same value at k as z.

$f_k^L.w(z[k])$ gives us the (node in column $k+1$ of the) first sequence after f_k^L (inclusive) that has $z[k]$ at position k. This is the first sequence in the $\frac{L}{k}$ block that has $z[k]$ at k. Since relative order of sequences with the same value is preserved, this will be the first sequence of the $\frac{L+1}{k+1}$ block.

$g_k^L.w(z[k])$ gives us the (node in column $k+1$ of the) first sequence after g_k^L (inclusive) that has $z[k]$ at position k. This is the first sequence outside of the $\frac{L}{k}$ block that has $z[k]$ at k. Since relative order

of sequences with the same value is preserved, this will be the first sequence after the $\frac{L+1}{k+1}$ block.

Therefore $f_k^L.w(z[k]) = f_{k+1}^{L+1}$ and $g_k^L.w(z[k]) = g_{k+1}^{L+1}$. □

There are two cases when we try to expand our $\frac{L+1}{k+1}$ block. If it is not empty ($f_{k+1}^{L+1} \neq g_{k+1}^{L+1}$), we can use the divergence values of the sequences in the d-PBWT to expand the boundaries. However, if it is empty ($f_{k+1}^{L+1} = g_{k+1}^{L+1}$), we must use the divergence values we calculated during virtual insertion to expand the boundaries initially. Lastly, when we expand the boundaries to include a new sequence in the block we also remember the starting position of the match ($k+1-L$) in an array dZ so that we can output it later. Meanwhile, $f_k^L.w(\text{opposite of } z[k])$ and $g_k^L.w(\text{opposite of } z[k])$ will give us the block of sequences (in column $k+1$) that have matches length L or longer until k and their match ends at k, we output these. We can repeat this procedure for all k to output all matches longer than L between query sequence and database. See Fig. 3.5.

The time complexity of this algorithm is easy to analyze. The Long Match Query algorithm (Algorithm 6) runs in worst case $O(N + |output|)$ time. The virtual insertion portion of the algorithm runs in worst case $O(N)$ because the haplotype panel and the u and v pointers are not updated. The query sweep loop has N iterations. All operations in one iteration of the query sweep loop take constant time except for the output while loop and the boundary expansion while loop. The output while loop will only output each match once, therefore the sum of all times it is entered in the algorithm is the number of matches found. The boundary expansion loop is entered once for every match that has length L (exactly) at some k. Every match will have length L exactly one time throughout the whole algorithm, therefore the sum of all times the boundary expansion loop is entered is the number of matches found. Therefore the algorithm runs in worst case $O(N + |output|)$ time.

Algorithm 6: Long Match Query: Find long matches between query sequence z and sequences in the d-PBWT in $O(N)$ time and three sweeps

```
/* Insert but without actually inserting, just calculate t_k s and
   divergence values                                                  */
t_0 = last node of column 0;
for k = 0 → N − 1 do
    t_{k+1} = t_k.w(z[k]);
z_dtemp = N;
b_dtemp = N;
for k = N → 0 do
    z_dtemp = min(z_dtemp, k);
    b_dtemp = min(b_dtemp, k);
    while z_k[z_dtemp − 1] = t_k.above[z_dtemp − 1] do
        z_dtemp − −;
    while z_k[b_dtemp − 1] = t_k[b_dtemp − 1] do
        b_dtemp − −;
    z_k.d = z_dtemp;
    belowz_k.d = b_dtemp;
// Query sweep
g_0^L = f_0^L = t_0;
for k = 0 → N − 1 do
    f_temp = f_k^L.w(opposite of z[k]);
    g_temp = g_k^L.w(opposite of z[k]);
    f' = f_k^L.w(z[k]) ;                                  // here f' = f_{k+1}^{L+1}
    g' = g_k^L.w(z[k]) ;                                  // here g' = g_{k+1}^{L+1}
    while f_temp ≠ g_temp do      // output matches longer than L that ended at k
        output match at [dZ[f_temp.ID],k) between f_temp.ID and z;
        f_temp = f_temp.below
    if f' = g' then                          // case where {f_{k+1}^{L+1}, g_{k+1}^{L+1}} block is empty
        if k + 1 − z_{k+1}.d = L then
            f' = f'.above;
            dZ[f'.ID] = k + 1 − L;
        if k + 1 − belowz_{k+1}.d = L then
            dZ[g'.ID] = k + 1 − L;
            g' = g'.below;
    if f' ≠ g' then                        // expand boundaries of block if not empty
        while f'.d ≤ k + 1 − L do
            f' = f'.above;
            dZ[f'.ID] = k + 1 − L;
        while g'.d ≤ k + 1 − L do
            dZ[g'.ID] = k + 1 − L;
            g' = g'.below;
    f_{k+1}^L = f';
    g_{k+1}^L = g';
```

Algorithm 7: Single Sweep Long Match Query: Find long matches between z and sequences in d-PBWT in average case $O(N)$ time and one sweep

$ef_0 = eg_0 = g_0^L = f_0^L = t_0 =$ last node of column 0;
for $k = 0 \to N-1$ **do**
  // calculate e_{k+1}
  $ef' = ef_k.w(z[k])$;
  $eg' = eg_k.w(z[k])$;
  if $ef' \neq eg'$ **then**
    $e' = e_k$;
  else
    $e' = ef'.d - 1$;
    if $z[e'] = 0$ **then**
      $ef' = ef'.above$;
      while $z[e'-1] = ef'[e'-1]$ **do** $e'--$;
      while $ef'.d \leq e'$ **do** $ef' = ef'.above$;
    else
      while $z[e'-1] = eg'[e'-1]$ **do** $e'--$;
      $eg' = eg'.below$;
      while $eg'.d \leq e'$ **do** $eg' = eg'.below$;
  $ef_{k+1} = ef', eg_{k+1} = eg', e_{k+1} = e'$;
  // Long Match Query
  $t_{k+1} = t_k.w(z[k])$;
  $f_{temp} = f_k^L.w($opposite of $z[k])$;
  $g_{temp} = g_k^L.w($opposite of $z[k])$;
  $f' = f_k^L.w(z[k])$; // here $f' = f_{k+1}^{L+1}$
  $g' = g_k^L.w(z[k])$; // here $g' = g_{k+1}^{L+1}$
  while $f_{temp} \neq g_{temp}$ **do** // output matches longer than L that ended at k
    output match at [$dZ[f_{temp}$.ID],k) between f_{temp}.ID and z;
    $f_{temp} = f_{temp}.below$
  if $f' = g'$ **then** // case where $\{f_{k+1}^{L+1}, g_{k+1}^{L+1}\}$ block is empty
    if $k+1-e_{k+1} = L$ **then**
      if $z_{k+1}[t_{k+1}.d-1] = 0$ **then**
        $f' = f'.above$;
        $dZ[f'.\text{ID}] = k+1-L$;
      else
        $dZ[g'.\text{ID}] = k+1-L$;
        $g' = g'.below$;
  if $f' \neq g'$ **then** // expand boundaries of block if not empty
    while $f'.d \leq k+1-L$ **do**
      $f' = f'.above$;
      $dZ[f'.\text{ID}] = k+1-L$;
    while $g'.d \leq k+1-L$ **do**
      $dZ[g'.\text{ID}] = k+1-L$;
      $g' = g'.below$;
  // now that boundaries were expanded, $f' = f_{k+1}^L$ and $g' = g_{k+1}^L$
  $f_{k+1}^L = f'$;
  $g_{k+1}^L = g'$;

Single Sweep Long Match Query

While a Long Match Query algorithm that runs in worst case $O(N)$ is an interesting theoretical development, an average case $O(N)$ algorithm that sweeps through the data only once may be more useful for real world applications, particularly implementations that use memory mapping. The pseudocode for an average case $O(N)$ Single Sweep Long Match Query algorithm is provided as Algorithm 7. This is done using Durbin's e_k. Of course the insertion algorithm can also be modified to run in a single sweep using e_k.

Results

Insertion & Deletion

We implemented the d-PBWT to test the run time of Insertion algorithm (Algorithm 1) and the Deletion algorithm (Algorithm 2). The experiment was performed with haplotypes from the UK Biobank dataset. The UK Biobank (UKB) dataset has 974,818 haplotypes (487,409 individuals) and approximately 700,000 sites. The experiment was performed on chromosome 21, which has 9,793 sites. The memory consumption of d-PBWT is approximately 48 bytes per site per haplotype, therefore the memory consumption of a d-PBWT on the full UKB dataset for chromosome 21 took approximately 450 gigabytes of memory. By extrapolation, a d-PBWT of the whole genome data of UKB would take approximately 29.8 terabytes of memory. The experiment was done by first randomizing the order of haplotypes to remove any bias. Then, the haplotypes were inserted one by one. Each insertion was timed. The insertion time relative to database size can be seen in Fig. 3.6A. There were 974,818 data points, therefore every thousand was averaged and the first 974 points (974,000 haplotypes) are plotted. The deletion experiment was performed by repeated removing a random haplotype until 974,817 haplotypes were removed. Each deletion was timed,

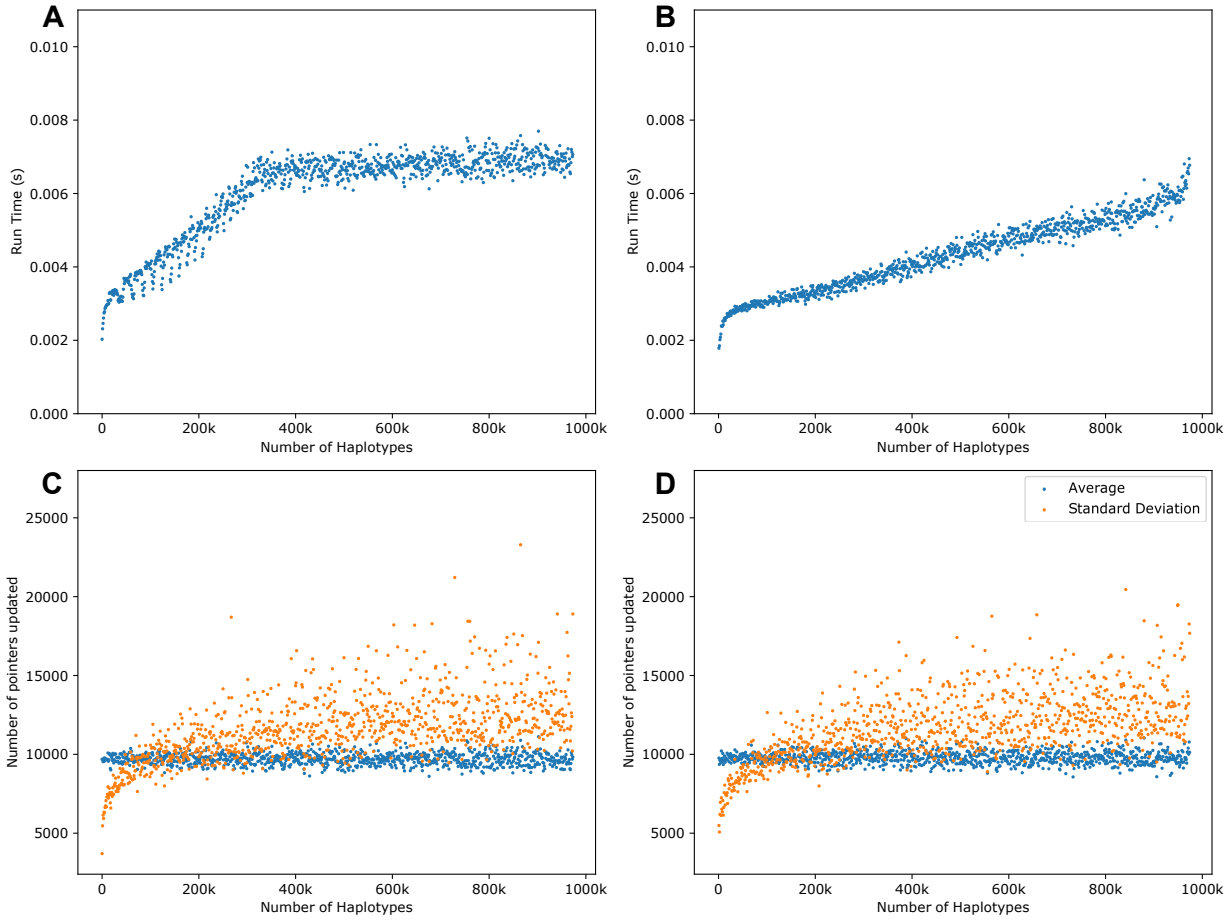

Figure 3.6: (**A**) Insertion time vs number of haplotypes in database. (**B**) Deletion time vs number of haplotypes in database. (**C**) Number of pointers updated per haplotype vs number of haplotypes in database for insertion. (**D**) Number of u/v pointers above z updated per haplotype vs number of haplotypes in database for deletion. The average of every 1000 measurements is plotted. For C and D, the standard deviation is also plotted.

and the average of every thousand is plotted in Fig. 3.6B. The last 817 deletions are not plotted. The program was run on a computer with an Intel Xeon Gold 5215 2.50GHz CPU and 3 terabytes of RAM.

Figs. 3.6A and 3.6B seem to imply that the average run time of the insertion and deletion algo-

rithms is not actually independent of the number of haplotypes in the database. This is not due to the number of u/v pointers updated. The number of u/v pointers updated per haplotype in the contiguous block of sequences above z is shown in Figs. 3.6C and 3.6D. The average and standard deviation of every thousand is plotted. The average number of pointers updated is roughly constant at 10,000 per haplotype. Each haplotype has 9,793 sites. This means the number of pointers updated per column is about 1 on average. However, the standard deviation increases with the number of haplotypes. The constant number of pointers updated is consistent with the theoretical property of the algorithms.

However, the non-constant run time suggests that when resource use scales up, other aspects of the computer system such as hardware architecture and operating system can affect run time.

d-PBWT Update

We tested the run time of insertion and deletion on a large d-PBWT. The d-PBWT was built with chromosome 21 of all of the haplotypes in the UK Biobank dataset except for a random 1000. Then, the d-PBWT was updated by 1000 insertions and 1000 deletions. The insertions and deletions were interspersed randomly. The inserted haplotypes were haplotypes that haven't been inserted into the d-PBWT yet while the deleted haplotypes were random haplotypes in the d-PBWT. This is meant to simulate a large database hosting millions of individuals' haplotypes that has frequent insertion and deletion requests. The results can be seen in Fig. 3.7. The 1000 insertions took on average 6.43 milliseconds and had a standard deviation of 8.73 milliseconds. The 1000 deletions took 6.28 milliseconds on average with a 7.98 millisecond standard deviation. Overall, the 2000 operations took 6.36 milliseconds on average and had a 8.36 millisecond standard deviation.

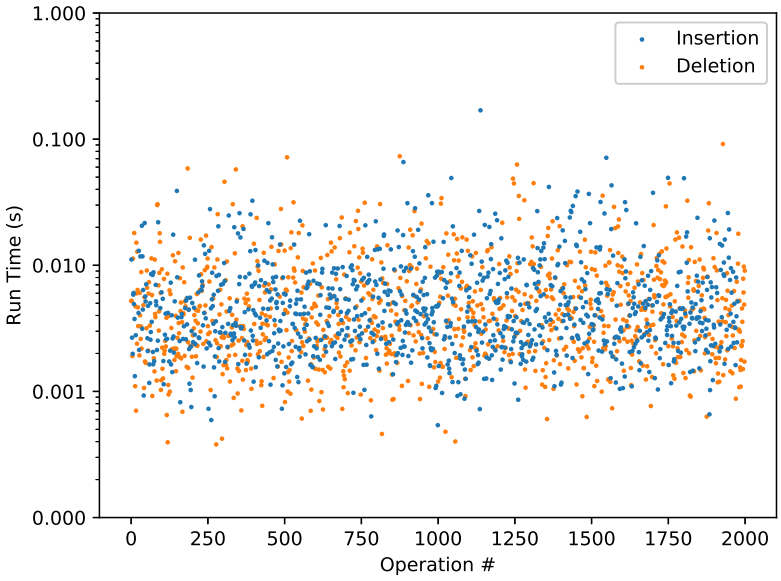

Figure 3.7: Run time of updating a d-PBWT with 973,818 haplotypes by 1,000 insertions and 1,000 deletions interspersed randomly.

Long Match Query

We tested the single sweep and triple sweep long match queries on the PBWT and d-PBWT. All the tests were performed on chromosome 21 of the UK Biobank dataset with $L = 1000$ sites. For both data structures, we tested seven sizes: 1,000, 25,000, 50,000, 100,000, 250,000, 500,000, and 974,618 (the full dataset $-$ 200, approx. one million). For each size a, we built a PBWT or d-PBWT on the first a haplotypes, then the next 200 haplotypes in the shuffled order were searched using a query algorithm. The same shuffled order was used for all tests. The results of these tests can be seen in Table 3.3.

Overall, the run time remained fairly constant across all experiments. Some minor differences are noticed: The fastest search was the triple sweep on the d-PBWT, which took around one millisec-

Table 3.3: Avg. long match query run times (ms) on PBWT and d-PBWT

	PBWT				d-PBWT			
	Single Sweep		Triple Sweep		Single Sweep		Triple Sweep	
# Haplotypes	Time	Std. Dev	Time	Std. Dev	Time	Std. Dev	Time	Std. Dev
1,000	1.06	0.04	2.00	0.05	1.19	0.70	0.79	0.07
25,000	1.37	0.07	2.77	0.03	1.77	0.87	0.79	0.10
50,000	1.46	0.08	3.02	0.08	1.85	1.13	0.78	0.10
100,000	1.50	0.08	3.28	0.07	1.75	1.06	0.74	0.11
250,000	1.61	0.16	3.44	0.15	2.01	1.92	0.72	0.12
500,000	1.79	0.20	4.02	0.16	3.38	3.05	0.70	0.12
974,618	2.13	0.14	4.65	0.22	3.55	4.31	0.94	0.22

ond at approximately one million individuals in the panel. The slowest was the triple sweep on the PBWT, which took 4.65 milliseconds for the full panel. The single sweep long match query took half the time the triple sweep took on the PBWT but more than three times as much as the triple sweep on the d-PBWT. This is because our implementation of the d-PBWT keeps consecutive nodes of a sequence adjacent in memory, this allows for fast tracking of a sequence because of locality of reference. In the PBWT, elements of the same sequence are not necessarily close to each other in memory, locality of reference is not guaranteed. However, the PBWT provides locality of reference in a column while the d-PBWT does not. The single sweep long match query needs to check multiple sequences in the same column to update e_k, however the triple sweep long match query just tracks the query haplotype and outputs matches. Therefore, the single sweep long match query is slower than triple sweep long match query on the d-PBWT and faster on the PBWT. The slight increase in run time as the databases get larger can be attributed to increasing number of matches or the changing proportion of the database in the cache. Given enough main memory to store the index (30 TB), a triple sweep long match query on a d-PBWT of the whole genome UKB data could take as little as 0.07 seconds.

Boundary Cases in Implementation

There are a few notable boundary cases to be aware of when implementing d-PBWT, PBWT, or any of their algorithms. Specifically, the following require attention: when a pointer to a node points to the bottom of a column (has value M in the case of PBWT), the u/v values of the bottom most contiguous block of 1s and 0s, and the set maximal matches when divergence is 0.

When the pointer to a node points to the bottom of a column k, its corresponding u pointer is equivalent to $(k,0).v$. Its v pointer is equivalent to the bottom of $k+1$. A review of what the u and v pointers represent will make this apparent. Furthermore, when the sequence at the bottom of a column k has 0 at site k, the v pointers of the contiguous block of 0s at the bottom of the column point to the bottom of column $k+1$. When the sequence at the bottom of column k has 1 at site k, the u pointers of the contiguous block of 1s at the bottom of column k point to $(k,0).v$. Lastly, the set maximal match query algorithm assumes that the set maximal matches are either all above or all below z, this is true in almost all cases. The only case where this isn't true is when the divergence is 0, in this case both directions need to be checked. This is also the case when the PBWT supports more an alphabet size more than 2. This doesn't affect the time complexity of the algorithm.

Discussion

In this chapter, we developed the first dynamic PBWT data structure that allows efficient updating. When inserting or deleting a haplotype in a static PBWT panel, one has to reconstruct the entire PBWT panel in $O(MN)$ time, while using dynamic PBWT, these can be achieved in Avg.$O(N)$ time. In addition, we simplified and improved the PBWT query search algorithms (Durbin's Algorithm 5 and Naseri *et al.*'s L-PBWT-Query). The new query algorithms run in worst case $O(N)$

time and use no additional data structures. We believe these algorithms have brought the PBWT data structure closer to its full potential.

This work would enable efficient genealogical search in large databases. For example, large consumer facing population databases hosting millions of individuals' haplotypes typically have a constant burden of maintaining the population haplotype data structure in order to serve to report real-time genealogical search results. We believe that d-PBWT provides a practical solution for maintaining the population haplotype data structure. Our insertion and deletion algorithms can be implemented to handle high-volume updates in a real-time fashion. Meanwhile, the performance of genealogical search queries can be guaranteed by efficient long match query algorithms.

While dynamic PBWT enables efficient updates, it indeed consumes a larger memory overhead (48 bytes per haplotype per site) than its static counterpart (16 bytes per haplotype per site, including u and v pointers). Therefore, for practical applications, the method of choice may depend on the costs of memory and CPU time. We estimate for the full UK Biobank, including all autosomal chromosomes, the memory requirements would be 30 TB for d-PBWT and 10 TB for the PBWT. We estimate the time to insert a haplotype into the d-PBWT is 0.42 seconds while the estimated time to rebuild the PBWT is 31.25 hours.

Notably, all three long match algorithms, L-PBWT-Query in [17] and Algorithms 6 and 7, achieve average case time complexity independent of database size. The only differences are their worst case time complexity, the number of sweeps required, and the memory needed for holding the additional data structures. The optimal algorithm will be a trade-off of these and other factors and will depend on the resources available. In this chapter, we assume the genotype data is bi-allelic. However, real genotype data is often multi-allelic. Future work may be done in extending this framework to multi-allelic data. This is simple enough using the data structure presented here and the mPBWT presented by [16]. Other future work includes reduction of d-PBWT memory cost

and speeding up random access.

d-PBWT and our algorithms open new research avenues for developing efficient genotype imputation and phasing algorithms. Current practices of imputation and phasing are mainly based on a fixed reference panel. With d-PBWT, individual's haplotypes in the reference panel can be iteratively refined, offering improved results.

CHAPTER 4: MINIMAL POSITIONAL SUBSTRING COVER

Introduction

Human chromosomes are a mosaic of ancestral chromosomal segments, resulting from accumulated recombination events. Therefore, in modeling a panel of haplotype sequences arising from population genetics processes, a useful model is to view a haplotype as a mosaic copy of subsequences of other haplotypes in a panel. In other words, a haplotype (the query) is threaded through different haplotypes (as templates) in the panel. For many years, the state of the art haplotype threading method was the Li & Stephens (LS) Model [12]. The classic Li and Stephens model parameterized this process as a hidden Markov model which can take into account the uncertainties regarding mismatches (emission probabilities) and template switching (transition probabilities). This model is quite sufficient for moderate sample sizes (hundreds to thousands) as it scales linearly with sample size. As a result, this model has served as a foundation for haplotype phasing and genotype imputation for the past two decades [2, 3, 6, 7, 13, 21]. However, in the biobank era when the panel size is large, the standard Li and Stephens model may not be efficient enough.

When sample size of the panel is large, Li & Stephens model becomes inefficient, as its time complexity is linear to the size of the panel, $O(MN)$, where M is the number of haplotypes in the panel and N is the number of sites per haplotype. The latest imputation methods based on LS have used PBWT to quickly identify a subset of templates as "surrogate parents" of the query, but still use LS HMM to sample the threading over the subset [2, 21].

Recent work has attempted to obtain the optimal haplotype threading in the Li & Stephens model in time sublinear to the size of the panel. Gerton Lunter described "fastLS", an algorithm that implements the Li & Stephens model and obtains the optimal haplotype threading through the use of

the Burrows-Wheeler transform [14]. His algorithm obtains the optimal haplotype threading orders of magnitude faster than the Viterbi algorithm. Yohei Rosen and Benedict Paten also described an algorithm that achieved runtime orders of magnitude faster than the Viterbi algorithm [20]. Their method achieves this using the efficient sparse representation of haplotypes and the lazy evaluation of dynamic programming. These algorithms have also been claimed to have runtime sublinear to the size of the reference panel. However, this claim has only been shown empirically.

In this work, we discuss an alternative formulation of the haplotype threading problem. This is based on the observation that when the size of the panel is large enough, approaching the size of the population, the IBD segments shared between the query and the template haplotypes can be much longer. Also, the error rates on modern large panels are very low ($\sim$0.1%). Therefore, the problem of haplotype threading for large panels is Li & Stephens model at the regime where the mismatch rates and switch rate are very low. In other words, it is of interest to make combinatorial formulations of the problem. There are a number of possible scoring functions for threading, however there are common themes between them. Usually, one wants to represent z using a small amount of haplotypes in X or a small amount of distinct segments. We minimize the number of segments we use to represent z. We formulate the Minimum Positional Substring Cover problem (MPSC), given a query z and a set of strings X, find a smallest set of positional substrings contained in z and a string in X that cover all characters of z. While this formulation simplifies the original haplotype threading by ignoring potential mismatches in the Li and Stephens model flavor, it enables efficient enumeration of all possible solutions, leveraging the structure of PBWT. Augmenting our algorithms with mismatch-tolerating methods such as random projection [18] or PBWT-smoothing [28], our formulation can capture the bulk of the high-probability threading paths, and thus provide flexibility for designing variations of downstream tasks such as genotype imputation and haplotype phasing.

In a sense, MPSC is a more general formulation of haplotype threading. The traditional haplotype

Table 4.1: Summary of algorithms on haplotype threading

	Functionalities	Alg.	Complexity
Lunter [14]	fastLS	Alg. 4, 5	Emp. $o(MN)$
Rosen and Paten [20]	LS Lazy Evaluation	-	Emp. $o(MN)$
This Chapter	MPSC	Alg. 8	$O(N)$
	h-MPSC	Alg. 17	$O(h\|\mathscr{C}\|+N)$
	h-MPSC	Alg. 18	$O(N)$
	Leftmost MPSC	Alg. 9	$O(N)$
	Rightmost MPSC	Alg. 10	$O(N)$
	Set Maximal MPSC	-	$O(N)$
	MPSC Graph	-	$O(N)$
	Length Maximal MPSC	Alg. 13	$O(N)$
	Set Max. MPSC Solution Space Count	-	$O(N)$
	Set Max. MPSC Solution Space Enumerate	-	$O(N+S_c)$
	L-MPSC	-	$O(N)$

M is the number of sequences. N is the number of sites. $\mathscr{C}$ is the outputted cover. h is the coverage guaranteed. S_c is the number of MPSCs outputted.

threading is a special case of MPSC with non-overlapping segments. MPSC formulation captures the fact that while the switching of templates indicates some recombination events, the exact breakpoint of the recombination events may be anywhere within the overlapping region between the templates. Furthermore, this combinatorial formulation is theoretically attractive because it is can be solved in worst case $O(N)$ time [22]. I.E. given a PBWT of the reference panel, a MPSC haplotype threading of a query haplotype can be done in time independent to the number of haplotypes in the reference panel. Further, a number of variations of the MPSC formulation can be solved efficiently, including Leftmost and Rightmost MPSCs, MPSC composed of only set maximal matches, and h-MPSC.

We formulate the Minimal Positional Substring Cover problem, discuss properties of MPSCs, and provide a solution given a PBWT of X. We discuss three variations of the MPSC problem: the leftmost, rightmost, and set maximal match only minimal positional substring covers. Then we

show that our original solution is leftmost and provide linear time solutions for the rightmost and set maximal match only MPSC problems. We discuss the h-MPSC problem and provide solutions in $O(h|\mathscr{C}|+N)$ and $O(N)$ time. We also explore the solution space of minimal positional substring covers, describe and solve the Length Maximal MPSC and L-MPSC problems, and demonstrate the usefulness of the MPSC formulation haplotype threading through an imputation benchmark. Table 4.1 summarizes the major algorithmic contributions of this chapter.

Background

A string z with N characters is indexed from 0 to $N-1$. The first character of z is $z[0]$ and the last is $z[N-1]$. The string $z[i,j]$ is the substring of z that starts at character i and ends at character j.

A **positional substring** of a string z is a 3-tuple, (i,j,z), where i and j are nonegative integers, $i \leq j+1 \leq |z|$, and z is the "source" of the substring. The substring corresponding to the positional substring (i,j,z) is $z[i,j]$. If $i = j+1$, then (i,j,z) corresponds to the empty string, ε. A non-empty positional substring (i,j,z) is contained in a string s if $0 \leq i \leq j < |s|$ and $s[i,j] = z[i,j]$. An empty positional substring $(i,i-1,z)$ is contained in a string s iff $0 \leq i \leq |s|$. Two positional substrings, (i,j,s) and (k,l,t) are equal iff $i = k$, $j = l$, and $s[i,j] = t[k,l]$. The length of a positional substring (i,j,s) is $j-i+1$.

A **positional substring cover**, $\mathscr{C}$, of a string z by a set of strings X is a set of positional substrings such that every character of z is contained in a positional substring and every positional substring in the set is present in z and a string in X. The "source," s, of a positional substring (i,j,s) in a positional substring cover can be any string s s.t. $s[i,j] = x[i,j] = z[i,j]$ for some $x \in X$. The size of a positional substring cover is the number of positional substrings it contains. The length of a positional substring cover is the sum of the lengths of its positional substrings.

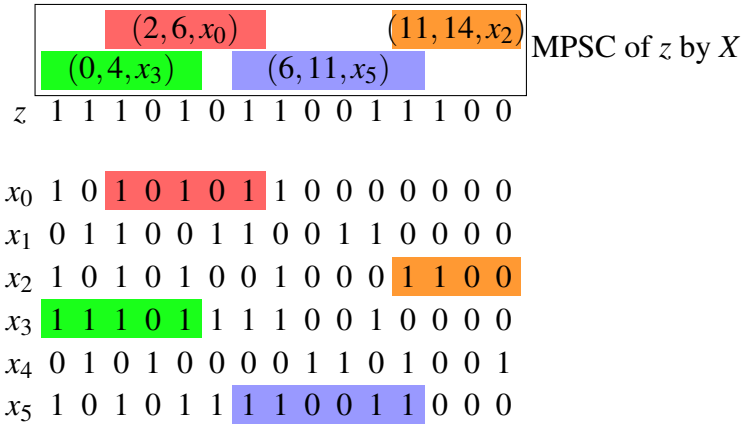

Figure 4.1: A minimal positional substring cover of z by X.

π is a projection on tuples. $\pi_1(i,j,z) = i, \pi_2(i,j,z) = j$, and $\pi_3(i,j,z) = z$.

In this chapter, we will use a variation of the PBWT that allows an alphabet of arbitrary size. The key difference is that instead of using the u and v arrays to keep track of next position, we use a three dimensional array, w, of size $M \times N \times |\Sigma|$, where Σ is the alphabet. $w[i][j][c]$ holds the position $a[i][j]$ would have in $a[i+1]$ if it had c at index i. This variation of the PBWT uses $O(MN|\Sigma|)$ space instead of the previous $O(MN)$. However, the set maximal match query time complexity remains the same as before ($O(N+c)$ where c is the number of matches found). This variation of the PBWT has been explored by Naseri et al. [16]. Lastly, when comparing a string s to a set of strings X, a match from s to $x \in X, s[i,j] = x[i,j]$, is a **longest match from s to X ending at index j** if it is a longest match between s and all strings in X that ends at index j. $\forall t \in X \forall k \in \{0,\ldots,i\}, k = i$ or $t[k,j] \neq s[k,j]$. The PBWT notation used in this chapter is different than the previous chapter, previously the column index of an array was indicated using a subscript, as in $a_k[i]$. In this chapter we use $a[k][i]$ instead. Furthermore, the superscript of y is moved to the second parameter of its subscript, the previous chapter's y_k^i is equivalent to this chapter's $y_{k,i}$.

Minimal Positional Substring Cover

The Minimal Positional Substring Cover (MPSC) problem is, given a set X of M strings and a string z, find a positional substring cover of z by X of the smallest size out of all positional substring covers of z by X. Call this cover a minimal positional substring cover of z by X. Refer to Fig. 4.1 for a depiction of an MPSC.

Properties

Claim 1. *A minimal positional substring cover of z by X exists iff for every $i \in \{0,\ldots,|z|-1\}$, there exists a string in X that has the same character as z at index i.*

Proof. If there exists an $i \in \{0,\ldots,|z|-1\}$ s.t. $\forall x \in X, z[i] \neq x[i]$, there exists no positional substring cover of z by X since a positional substring that covers index i and is contained in z and a string in X doesn't exist. There doesn't exist a positional substring cover of z by X, so a minimal positional substring cover of z by X doesn't exist.

If $\forall i \in \{0,\ldots,|z|-1\}, \exists x \in X$ s.t. $z[i] = x[i]$, then there exists a positional substring cover $\mathscr{C}$ of z by X. $\mathscr{C} = \{\,(i,i,z) : i \in \{0,\ldots,|z|-1\}\,\}$. A positional substring cover of z by X exists, so a minimal positonal substring cover of z by X exists. □

Claim 2. *For a minimal positional substring cover $\mathscr{C}$ of z by X, every index $k \in \{0,\ldots,|z|-1\}$ is contained in at most two of its positional substrings. $\forall k \in \{0,\ldots,|z|-1\}, \exists (i_0,j_0,s_0),(i_1,j_1,s_1) \in \mathscr{C}$ s.t. $\forall (i_2,j_2,s_2) \in \mathscr{C}, i_2 \leq k \leq j_2 \iff ((i_2 = i_1 \wedge j_2 = j_1 \wedge s_2 = s_1) \vee (i_2 = i_0 \wedge j_2 = j_0 \wedge s_2 = s_0))$. Note that every index is contained in at least one positional substring by the definition of positional substring cover, therefore, every index is contained in at least one and at most two positional substrings in $\mathscr{C}$.*

Proof. Suppose there exists an index k that is contained in more than two positional substrings in a minimal positional substring cover $\mathscr{C}$ of z by X. Take $\mathscr{D}$, the set of positional substrings in $\mathscr{C}$ that contain i, $\mathscr{D} = \{p \in \mathscr{C} : \pi_1(p) \leq i \leq \pi_2(p)\}$. Take the positional substrings $p, q \in \mathscr{D}$ that start the earliest and end the latest respectively. $\forall r \in \mathscr{D}, \pi_1(p) \leq \pi_1(r)$ and $\pi_2(q) \geq \pi_2(r)$. p covers at least $[\pi_1(p), i]$ and q covers at least $[i, \pi_2(q)]$. Therefore, all the indices contained in positional substrings in $\mathscr{D}$ are covered by p and q. Therefore, removing all positional substrings in $\mathscr{D}$ except p and q from $\mathscr{C}$ would yield a smaller set of positional substrings that covers z by X (since $|\mathscr{D}| > 2$). This contradicts the fact that $\mathscr{C}$ is a minimal positional substring cover of z by X. □

Claim 3. *Given a minimal positional substring cover $\mathscr{C}$ of z by X, the starting points of all positional substrings in $\mathscr{C}$ are unique and their ending points are unique.* $\forall p, q \in \mathscr{C}, p \neq q \implies (\pi_1(p) \neq \pi_1(q) \wedge \pi_2(p) \neq \pi_2(q))$.

Proof. Suppose that there are two positional substrings in $\mathscr{C}$ that start at the same position, i. Then, the one with the smaller ending position can be removed from $\mathscr{C}$. This new set is a smaller positional substring cover of z by X because all of the sites are still covered by positional substrings contained in z and a string in X. This contradicts the assumption that $\mathscr{C}$ is a minimal positional substring cover of z by X. Similar reasoning can be applied to positional substrings with the same ending position. □

Claim 4. *For a minimal positional substring cover $\mathscr{C}$ of z by X. For any positional substring $p \in \mathscr{C}$, if p has the i-th smallest starting point out of all positional substrings in $\mathscr{C}$, it also has the i-th smallest ending point out of all positional substrings in $\mathscr{C}$.* $\forall p, q \in \mathscr{C}, \pi_1(p) < \pi_1(q) \iff \pi_2(p) < \pi_2(q)$.

Proof. Suppose there are two positional substrings, $p, q \in \mathscr{C}$ s.t. the order of their starting points and ending points are different. Without loss of generality, say p starts earlier. Then, we have

$\pi_1(p) < \pi_1(q)$ and $\pi_2(p) > \pi_2(q)$. In this case, q is completely covered by p. Removing q from $\mathscr{C}$ results in a positional substring cover of z by X that is smaller than $\mathscr{C}$. This contradicts the assumption that $\mathscr{C}$ is a minimal positonal substring cover. □

We use Claim 4 to define i-th positional substring in a minimal positional substring cover of z by X. The i-th positional substring in a minimal positional substring cover $\mathscr{C}$ is the positional substring with the i-th smallest starting position in the cover, 0-indexed. This is equivalent to ordering by ending position (by Claim 4). We use $\mathscr{C}[i]$ to denote the i-th positional substring of $\mathscr{C}$.

Main Idea

We present a solution to the Minimal Positional Substring Cover problem here. This solution assumes that a PBWT of X is provided. We start with $\mathscr{P} = \emptyset$. Add to $\mathscr{P}$ the positional substring corresponding to the longest match from z to X ending at $N-1$. Every minimal positional substring cover of z by X must cover index $N-1$. Replacing the positional substring in any such cover with the longest match from z to X ending at $N-1$ will yield a minimal positional substring cover of z by X because the sets are the same size and the longest match ending at $N-1$ covers all sites any match ending at $N-1$ covers by definition. Now, $\mathscr{P}$ is a subset of a minimal positional substring cover $\mathscr{C}$ of a string z by X. It covers indices $\{k,\ldots,N-1\}$. For some integer $0 < k \leq N-1$. And none of the indices $\{0,\ldots,k-1\}$. We will show in Lemma 1 that given a subset of a minimal positional substring cover that covers a contiguous section of indices, the longest match ending at the last index it doesn't cover is also in a minimal positional substring cover. Using Lemma 1, we add the positional substring corresponding to the longest match ending at $k-1$ to $\mathscr{P}$. We repeat this with our new $\mathscr{P}$ and new k until $\mathscr{P}$ is a minimal positional substring cover of z by X. Refer to Fig. 4.2 for a depiction of Lemma 1.

Lemma 1 (Minimal Positional Substring Cover Modularity)**.** *Given a subset $\mathscr{P}$ of a minimal*

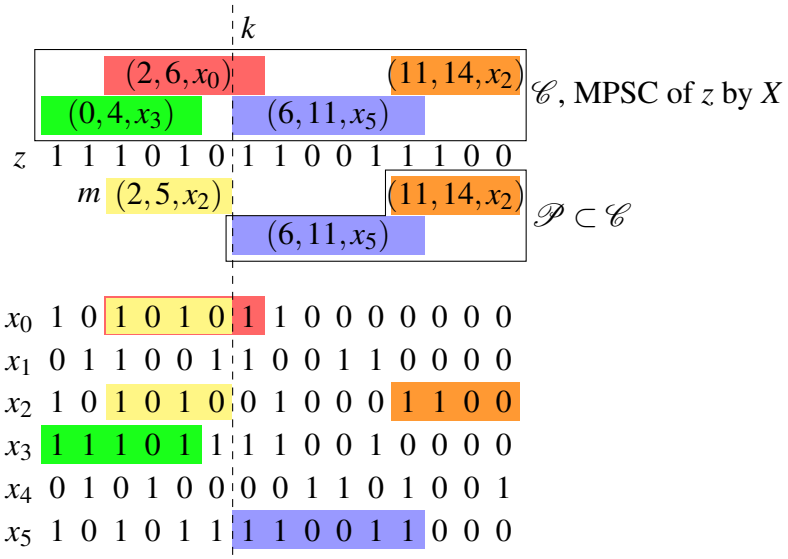

Figure 4.2: Depiction of Lemma 1. $\mathscr{C}$ is a MPSC of z by X. $\mathscr{P}$ is a subset of $\mathscr{C}$ that covers only indices $\{k,\ldots,N-1\}$. The longest match m ending at index $k-1$ and $\mathscr{P}$ form a subset of a MPSC $\mathscr{D}$ of z by X. $\mathscr{P}\cup\{m\}\subseteq\mathscr{D}$.

positional substring cover $\mathscr{C}$ of a string z by X that covers all the indices in $\{k,\ldots,N-1\}$ and none of the indices in $\{0,\ldots,k-1\}$ for $0<k\leq N-1$. Take $m=(i,k-1,s)$, the longest match ending at $k-1$ from z to X. $\mathscr{P}\cup\{m\}$ is a subset of a minimal positional substring cover of z by X.

Proof. If $\mathscr{P}$ is a subset of a minimal positional substring cover $\mathscr{C}$ of a string z by X, then, the $(|\mathscr{C}|-|\mathscr{P}|-1)$-th positional substring in $\mathscr{C}$ must cover index $k-1$. If it didn't, either the index $k-1$ is not covered by $\mathscr{C}$, or another substring in $\mathscr{C}$ covers index $k-1$. In the first case, our definition of $\mathscr{C}$ is contradicted.

In the second case, if an n-th positional substring covers $k-1$, if $n>|\mathscr{C}|-|\mathscr{P}|-1$, our defintion of $\mathscr{P}$ is contradicted. If $n<|\mathscr{C}|-|\mathscr{P}|-1$, our definition of n-th positional substring is contradicted (n-th positional substring ends after $(|\mathscr{C}|-|\mathscr{P}|-1)$-th positional substring), or $\mathscr{C}$ is not a minimal positional substring cover ($(|\mathscr{C}|-|\mathscr{P}|-1)$-th positional substring starts after $k-1$, in which case

it can be removed while maintaining the coverage of all sites).

Now, for any $\mathscr{C}$, replacing the $(|\mathscr{C}| - |\mathscr{P}| - 1)$-th positional substring with the longest match ending at $k-1$ of z and X will yield a minimal positional substring cover of z by X. This is because the new set is the same size as $\mathscr{C}$ and all the sites $\mathscr{C}$ covered are also covered by the new set. Any indices greater than $k-1$ are covered by $\mathscr{P}$ and there are no indices less than $k-1$ that the original $(|\mathscr{C}| - |\mathscr{P}| - 1)$-th positional substring covered that the longest match ending at $k-1$ doesn't cover by the definition of longest match ending at $k-1$. □

Obtaining the longest match ending at index i is easy using the PBWT. At any index i, for some sequence $a[i+1][j]$, the longest match ending at i between it and $X - \{y_{i+1,j}\}$ starts at the smaller of $d[i+1][j]$ and $d[i+1][j+1]$ (the starting position of the longest match ending at i between $\{a[i+1][j], a[i+1][j-1]\}$ and $\{a[i+1][j], a[i+1][j+1]\}$ respectively. Precisely, the longest match ending at i from z to X is $(\min(d[i+1][j], d[i+1][j+1]), i, z)$. Unfortunately, the query sequence is not in the PBWT, so using this method directly is not possible. We use the method of "virtually inserting" a query haplotype into a PBWT in time linear with respect to the length of the haplotype from Chapter 3. This calculates the positions in the prefix array and the divergence values a query string would have if it was present in a PBWT.

Algorithm

A virtual insertion entails the calculation of the positions the string would take in the prefix arrays a, and the calculation of the new divergence values of the string and the string below it in every prefix array index. The original algorithm was described for binary strings. Here we describe the algorithm on an arbitrary alphabet PBWT. We begin by calculating the locations of the query z in the prefix arrays.

We will keep track of the index $t[k], k \in \{0, \ldots, N\}$. $t[k]$ is the index in the k-th prefix array ($a[k]$) which the query sequence z would be placed above if it were in the PBWT. Choose $t[0]$ arbitrarily, we choose $t[0] = 0$. The position z would be above in $a[k+1]$ is the position $t[k]$ would be at in $a[k+1]$ if it had $z[k]$ at position k. In other words, $t[k+1] = w[k][t[k]][z[k]]$. We can use the w array to calculate each $t[k]$ in constant time. Overall, $O(N)$ where $N = |z|$. Next, we calculate the divergence values.

The key observation for the efficient calculation of divergence values is that the divergence value of the query sequence z at index k will be less than or equal to its divergence value at index $k+1$. The same holds for the divergence value of the sequence below z. See Section 2.3 of [24] for a proof of this claim. Therefore, we calculate two integer arrays, d_z and d_{belowz} of size $N+1$. We calculate $d_z[N]$ and $d_{belowz}[N]$ by starting at $d = N$ and decrementing until $z[d-1] \neq y_{N,t[N]-1}[d-1]$ for d_z (and until $z[d-1] \neq y_{N,t[N]}[d-1]$ for d_{belowz}). Then, we set $k = N-1 \rightarrow 0$ and calculate $d_z[k]$ and $d_{belowz}[k]$ by starting at $d_z[k+1]$ and $d_{belowz}[k+1]$ respectively and decrementing until the next characters are not equal, see condition in previous sentence. Overall, this takes $O(N)$ time since every index has a constant cost and over all the sites, the divergence calculation is a counter from N to 0.

As we noted before, obtaining the longest match ending at index k is easy given a PBWT of binary strings. This still holds in our arbitrary alphabet PBWT. The longest match ending at k of $a[k+1][j]$ to $X - \{y_{k+1,j}\}$ will be adjacent to it in $a[k+1]$, either above or below it. For our query z, the longest match ending at k starts at the smaller of $d_z[k+1]$ and $d_{belowz}[k+1]$. Therefore, using Lemma 1 after virtual insertion, we can output a minimal positional substring cover of z by X. Add the longest match ending at $N-1$ $(j, N-1, z)$ to the cover, save $k = j$. Repeat the following until $k = 0$: Add the longest match ending at $k-1$, $(j, k-1, z)$ to the cover, set $k = j$. At the end of this process, we have a minimal positional substring cover of z by X. Refer to Algorithm 8 for the pseudocode of this algorithm. Note that there are some boundary cases in the algorithm not

included in the pseudocode, these are discussed in "Boundary cases in implementation."

Algorithm 8: Minimal Positional Substring Cover

```
// Virtual Insertion
t[0] = 0;
for k = 0 → N − 1 do
    t[k+1] = w[k][t[k]][z[k]];
CURd_z = CURd_belowz = N;
for k = N → 0 do
    CURd_z = min(CURd_z, k);
    CURd_belowz = min(CURd_belowz, k);
    while CURd_z > 0 and z[CURd_z − 1] = y_{k,t[k]−1}[CURd_z − 1] do
        CURd_z − −;
    d_z[k] = CURd_z;
    while CURd_belowz > 0 and z[CURd_belowz − 1] = y_{k,t[k]}[CURd_belowz − 1] do
        CURd_belowz − −;
    d_belowz[k] = CURd_belowz;
// Minimal Positional Substring Cover Output
𝒞 = ∅;
k = N;
while k > 0 do
    oldk = k;
    k = min(d_z[k], d_belowz[k]);
    if k = oldk then
        output "No positional substring cover of z by X exists.";
        exit;
    𝒞 = 𝒞 ∪ {(k, oldk − 1, z)};
output 𝒞;
```

Time Complexity

The time complexity of this algorithm is $O(N)$ where N is the length of the query string. The calculation of locations in the prefix array is clearly $O(N)$ since the calculation of each index is constant time and there are N indices. The calculation of divergence values is not constant time per index, however each index incurs a constant cost and a variable cost. Over all N indices, the sum of the variable costs is $O(N)$ since the variable costs are counters from N to 0. Therefore, the virtual

insertion part of the algorithm takes $O(N)$ time. The minimal positional substring cover output section of the algorithm takes constant time per positional substring in the ouput, therefore it takes $O(|\mathscr{C}|)$ time. The size of a minimal substring cover of z by X is at most N because there may not be any empty positional substrings in a minimal positional substring cover. Therefore, given a PBWT of a set of M strings, and a query string z of length N, Algorithm 8 outputs a minimal positional substring cover of z by X in $O(N)$ time.

Leftmost Minimal Positional Substring Cover

A leftmost minimal positional substring cover $\mathscr{C}$ of z by X is a minimal positional substring cover of z by X with the following property: for any i-th substring in $\mathscr{C}$, it starts at least as early as the i-th substring of every other minimal positional substring cover of z by X. We use i-th positional substring of an MPSC as defined in Chapter 4. With this notation, a minimal positional substring cover $\mathscr{C}$ of z by X is leftmost if $\forall i \in \{0,\ldots,|\mathscr{C}|-1\}\forall \mathscr{D} \in \{\mathscr{P} : \mathscr{P}$ is a minimal positional substring cover of z by $X\}, \pi_1(C[i]) \leq \pi_1(D[i])$.

Claim 5. *If the i-th substring in a leftmost minimal positional cover $\mathscr{C}$ of z by X begins at index j, every $(i-1)$-th substring in a minimal positional substring cover of z by X contains index $j-1$.* $\forall i \in \{1,\ldots,|\mathscr{C}|-1\}\forall \mathscr{D} \in \{\mathscr{P} : \mathscr{P}$ *is a minimal positional substring cover of z by $X\}, \pi_1(\mathscr{D}[i-1]) \leq \pi_1(\mathscr{C}[i]) - 1 \leq \pi_2(\mathscr{D}[i-1])$.*

Proof. Suppose there existed an $i \in \{1,\ldots,|\mathscr{C}|-1\}$ and $\mathscr{D} \in \{\mathscr{P} : \mathscr{P}$ is a minimal positional substring cover of z by $X\}$ such that $\pi_1(\mathscr{C}[i]) - 1 > \pi_2(\mathscr{D}[i-1])$. Then, since $\mathscr{D}$ is a cover of z by X, it contains a positional substring that contains index $\pi_1(\mathscr{C}[i]) - 1$, call it $\mathscr{D}[j]$. If $j < i-1$, the definition of i-th positional substring is contradicted since $\pi_2(\mathscr{D}[j]) > \pi_2(\mathscr{D}[i-1]$ and $j < i-1$. If $j > i-1$, $j = i$ by definition of i-th positional substring, and $\mathscr{C}$ is not leftmost since $\pi_1(\mathscr{C}[i]) >$

$\pi_1(\mathscr{D}[i])$. Therefore, no such i and $\mathscr{D}$ exist.

Suppose there existed an $i \in \{1,\ldots,|\mathscr{C}|-1\}$ and $\mathscr{D} \in \{\mathscr{P} : \mathscr{P}$ is a minimal positional substring cover of z by $X\}$ such that $\pi_1(\mathscr{C}[i])-1 < \pi_1(\mathscr{D}[i-1])$. Then the set $\{\mathscr{D}[j] : 0 \leq j < i-1\}$ covers the indices $[0, \pi_1(\mathscr{D}[i-1])-1]$ with $i-1$ positional substrings and the set $\{\mathscr{C}[j] : i \leq j < |\mathscr{C}|\}$ covers the indices $[\pi_1(\mathscr{C}[i]), N-1]$ with $|\mathscr{C}|-i$ positional substring. Their union is a positional substring cover of z by X with $|\mathscr{C}|-1$ positional substrings. This contradicts the assumption that $\mathscr{C}$ is a minimal positional substring cover of z by X, therefore, no such i and $\mathscr{D}$ exist. □

Here, we will show that the minimal positional substring cover $\mathscr{C}$ of z by X outputted by Algorithm 8 is leftmost. The $(|\mathscr{C}|-1)$-th substring in any MPSC $\mathscr{D}$ of z by X must cover index $N-1$. This is because it is the substring with the greatest ending position in $\mathscr{D}$ (in order for $\mathscr{D}$ to be minimal) and $\mathscr{D}$ must cover index $N-1$. The leftmost starting point of any positional substring that ends at $N-1$ contained in z and a string in X is the starting point of the longest match from z to X ending at index $N-1$. This is exactly the $(|\mathscr{C}|-1)$-th positional substring in $\mathscr{C}$.

Therefore, the set containing only the $\mathscr{C}[|\mathscr{C}|-1]$ is a subset of a leftmost minimal positional substring cover of z by X. By Claim 5, the $(|\mathscr{C}|-2)$-th substring of a leftmost minimal positional substring cover must contain index $\pi_1(\mathscr{C}[|\mathscr{C}|-1])-1$. The leftmost starting point of any positional substring that ends at $\pi_1(\mathscr{C}[|\mathscr{C}|-1])-1$ contained in z and a string in X is the longest match between z and X ending at that index. This is exactly the $(|\mathscr{C}|-2)$-th positional substring in $\mathscr{C}$. Therefore, the set $\{\mathscr{C}[|\mathscr{C}|-2], \mathscr{C}[|\mathscr{C}|-1]\}$ is a subset of a leftmost minimal positional substring cover of z by X. This logic can be repeated for every positional substring in $\mathscr{C}$ to show that it is a leftmost minimal positional substring cover of z by X.

Algorithm 9: LeftmostMPSC: Output Leftmost MPSC of z by X

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

Rightmost Minimal Positional Substring Cover

A rightmost minimal positional substring cover $\mathscr{C}$ of z by X is a minimal positional substring cover of z by X with the following property: for any i-th substrings, it ends at least as late as the i-th substring of every other minimal positional substring cover of z by X. $\forall i \in \{0, \ldots, |\mathscr{C}| - 1\} \forall \mathscr{D} \in \{\mathscr{P} : \mathscr{P}$ is a minimal positional substring cover of z by $X\}, \pi_2(\mathscr{C}[i]) \geq \pi_2(\mathscr{D}[i])$.

The logic for obtaining a rightmost minimal positional substring cover is similar to that for a leftmost. One method is to find the leftmost cover using Algorithm 8 of the query $\mathrm{R}(z)$ and $X_{\mathrm{R}} = \{\mathrm{R}(x) : x \in X\}$, then reverse all positional substrings in the outputted cover. This would require building a PBWT on X_{R}, which would take $O(|\Sigma|MN)$ time. Another method is to build it with the longest matches starting at i. The first positional substring the cover $\mathscr{C}$ would be the longest match starting at index 0. The second would be the longest match starting at $\pi_2(\mathscr{C}[0]) + 1$, the third at $\pi_2(\mathscr{C}[1]) + 1$, and so on. The proof for this cover being rightmost is very similar to the leftmost proof.

To obtain the longest match from z to X starting at index k efficiently, create an analogous array to the divergence array for longest match starting at k. Take array δ of length N where $\delta[i] = \min(d_z[i+1], d_{belowz}[i+1])$ for $i \in \{0, \ldots, N-1\}$ (use the definitions of d_z and d_{belowz} from Al-

gorithm 8). Create the analogous array b of length N where $b[j] = \max_{i \in \{k \in \{0,\ldots,N-1\}:\delta[k] \leq j\}} i$ for $j \in \{0,\ldots,N-1\}$. This can be done in $O(N)$ time. Now $b[j]$ corresponds to the end of the longest match starting at j from z to X. Therefore, we can build a rightmost minimal positional substring cover of z by X in $O(N)$ time given a PBWT of X.

Algorithm 10: 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;
```

Minimal Positional Substring Cover Using Set Maximal Matches

In haplotype threading, it is usually better to have longer matches when possible. This is because matches that are longer are usually between individuals that are more closely related. In this case, it may be useful to have a MPSC of z by X that is composed of only set maximal matches. Here we provide a brief description of how to output a MPSC of z by X containing only set maximal matches in $O(N)$ time given a PBWT of X.

We begin by running Algorithm 8 on z and the PBWT of X. Call its outputted cover $\mathscr{D}$. All the positional substrings (i, j, s) in $\mathscr{D}$ are longest matches ending at j from z to X, therefore the longest match (m) starting at i from z to X is a set maximal match from z to X. This is because for another

match to encompass this match, it needs to contain the indices $\{i, \ldots, j\}$ and start earlier or end later. No match starts earlier because (i, j, s) is the longest match ending at j. No match ends later because m is the longest match starting at i. Therefore, in order to obtain an MPSC containing only set maximal matches, we just have to replace every positional substring $(i, j, s) \in \mathscr{D}$ with the longest match starting at i from z to X. We can do this efficiently using the b array from Chapter 4. $b[i]$ contains the ending point of the largest match starting at i. We can compute the b array in $O(N)$ time. We can replace every match in $\mathscr{D}$ in $O(1)$ time and $|\mathscr{D}| \leq N$. Therefore we can obtain a MPSC composed of only set maximal matches in $O(N)$ time.

MPSC Solution Space

Although a minimal positional substring cover of a query haplotype z by a panel X is an inference for haplotype threading, there may be many possible minimal positional substring covers of z by X. The MPSC outputted may not be the most accurate haplotype threading. Therefore, we consider the set of all possible MPSCs of z by X.

We begin the exploration of the solution space of minimal positional substring covers of z by X by attempting to bound it. We already have two useful bounds that can be efficiently obtained, namely the leftmost and rightmost MPSCs. The starting point of every i-th positional substring in an MPSC of z by X must be at least $\pi_1(\mathscr{C}_l[i])$, the starting point of the i-th positional substring of $\mathscr{C}_l$, the leftmost MPSC. Likewise, the ending point of every i-th positional substring in an MPSC of z by X is at most $\pi_2(\mathscr{C}_r[i])$, the ending point of the i-th positional substring of $\mathscr{C}_r$, the rightmost MPSC. We have proved the existence of leftmost and rightmost MPSCs for any z and X where there exists an MPSC of z by X.

We have even tighter bounds of the solution space of MPSCs. We showed in Claim 5 that if the

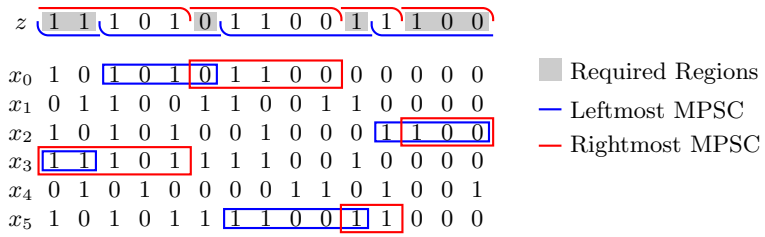

Figure 4.3: Required regions for MPSC of z by X. The i-th required region is bounded by the $i+1$-th positional substring in the Leftmost MPSC and the $i-1$-th positional substring in the Rightmost MPSC.

i-th positional substring in a leftmost MPSC of z by X begins at index j, every $i-1$-th positional substring in an MPSC of z by X contains the index $j-1$. Here, we show a similar property for rightmost MPSCs.

Claim 6. *Every $i+1$-th positional substring in any MPSC of z by X contains index $\pi_2(\mathscr{C}[i])+1$. Where $\mathscr{C}$ is a rightmost MPSC of z by X.*

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

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

For every i-th positional substring in a minimal positional substring cover of z by X, we now have two positions that it is guaranteed to contain: $\pi_2(\mathscr{C}_r[i-1])+1$ and $\pi_1(\mathscr{C}_l[i+1]-1)$. Where $\mathscr{C}_l$ is a leftmost MPSC of z by X and $\mathscr{C}_r$ is a rightmost MPSC of z by X. Furthermore, there are MPSCs of z by X where the positions just after and before these positions are not contained in the i-th positional substrings (rightmost and leftmost MPSCs of z by X with no overlap, respectively). Therefore, we have the exact range of sites common to all i-th positional substrings in MPSCs of z by X. Call it the i-th **required region**. The i-th required region is contained in every i-th positional substring in an MPSC of z by X. The i-th required region is exactly the first site after the $i-1$-th positional substring in a rightmost MPSC to the last site before the $i+1$-th positional substring in a leftmost MPSC. See Fig. 4.3 for a depiction of required regions and how they are obtained. The i-th required region is defined in Lemma 5.

Lemma 5 (Required Regions)**.** *There exists a contiguous nonempty range of sites for every $i \in \{0,\ldots,|\mathscr{C}|-1\}$, such that $\mathscr{C}[i]$ contains this range of sites for all MPSCs $\mathscr{C}$ of z by X. Call the largest such range the i-th required region. For $i \in \{1,\ldots,|\mathscr{C}|-2\}$, this range is $[\pi_2(\mathscr{C}_r[i-1])+1, \pi_1(\mathscr{C}_l[i+1])-1]$, where $\mathscr{C}_r$ and $\mathscr{C}_l$ are rightmost and leftmost MPSCs of z by X respectively. The required region for $\mathscr{C}[0]$ is $[0,\pi_1(\mathscr{C}_l[1])-1]$, and for $\mathscr{C}[|\mathscr{C}|-1]$, $[\pi_2(\mathscr{C}_r[|\mathscr{C}|-2]),|z|-1]$.*

Proof. By Claim 6, every i-th substring must contain index $\pi_2(\mathscr{C}_r[i-1])+1$ for $i \in \{1,\ldots,|\mathscr{C}|-1\}$. The 0-th substring must contain index 0 by definition of MPSC and i-th substring (every site is covered and $\mathscr{C}[0]$ has the smallest starting point of all positional substrings in $\mathscr{C}$). Therefore, the required region exists and is nonempty for all i.

By Claim 5, every i-th substring must contain index $\pi_1(\mathscr{C}_l[i+1])-1$ for $i \in \{0,\ldots,|\mathscr{C}|-2\}$ [22]. Furthermore, the $|\mathscr{C}|-1$-th positional substring must contain index $|z|-1$ by definition of MPSC and i-th substring (every site is covered and $\mathscr{C}[|\mathscr{C}|-1]$ has the largest ending point of all positional substrings in $\mathscr{C}$).

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

MPSC Graph

We attempt to represent the solution space of MPSCs as a graph. Positional substrings are vertices in the graph and edges occur between positional substrings that are adjacent or overlapping. Consider the naive graph $G=(V,E)$ where the set of nodes V is the set of positional substrings that are contained in both z and X with z as the source string. I.E. $v=(i,j,z) \in V \iff (i,j,z)$ is present in z and $x \in X$. There is an edge between two positional substrings u and v if u starts before v and u and v are adjacent or overlapping ($\pi_1(u) \leq \pi_1(v)$ and $\pi_2(u) \geq \pi_1(v)-1$). In this graph, all shortest paths from $(-1,0,z)$ to $(N-1,N,z)$ correspond to minimal positional substring covers of z by X. Furthermore, all minimal positional substring covers are represented by a shortest path from $(-1,0,z)$ to $(N-1,N,z)$. However, there are possibly $O(N^2)$ nodes and $O(N^4)$ edges in this graph, so a shortest path finding algorithm would run in $O(N^4)$ time. Therefore, we attempt to simplify the graph. We begin with the following observations.

Claim 7. *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 8. *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. □

Therefore, we consider a graph where the set of nodes is the set of set maximal match positions from z to X. Every possible non-empty match is contained in at least one of these nodes. There is an edge between nodes u and v with the same conditions as before: $\pi_1(u) \leq \pi_2(v)$ and $\pi_2(u) \geq \pi_1(v) - 1$. Call the node that contains index 0 the source node, s. Call the node that contains index $N-1$ the sink node, t. s and t are unique by Claim 8. There is a one-to-one correspondence between shortest paths from s to t in this graph and minimal positional substring covers of z by X composed of set maximal matches. By Claim 8, the number of nodes in this graph is $O(S) \subseteq O(N)$. However, the number of edges in this graph may be $O(S^2)$, therefore a shortest path finding algorithm on this

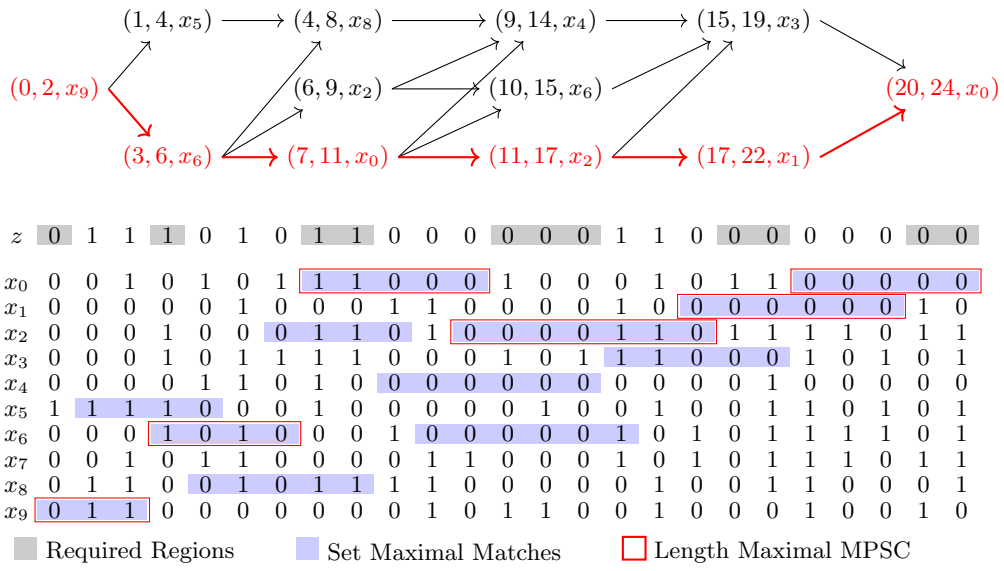

Figure 4.4: Constructed MPSC graph of z by X. In gray are the required regions for z and X. Highlighted in blue are the set maximal matches from z to X. Highlighted in red is the Length Maximal MPSC of z by X.

graph would still take $O(S^2) \subseteq O(N^2)$ time. We simplify the graph of the solution space again, this time considering the i-th required regions.

We will construct a graph where the set of nodes is the set of set maximal match positions from z to X. However, we will also consider the information gained from the i-th required regions. For two nodes u, v, there is an edge from u to v in the graph if the previous requirements are fulfilled, u contains the complete i-th required region, and v contains the complete $i+1$-th required region. The following property is useful in this construction.

Claim 9. *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 $\mathscr{C}_l[i]$ with m and removing any overlap of $\mathscr{C}_l[i-1]$ and $\mathscr{C}_l[i]$ from $\mathscr{C}_l[i-1]$ (for leftmost MPSC of z by X, $\mathscr{C}_l$).

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

If u contains the i-th required region and v contains the $i+1$-th required region, then u starts before v. Therefore, there is an edge from u to v in this graph iff u contains the i-th required region, v contains the $i+1$-th required region, and u and v are adjacent or overlapping. See Fig. 4.4 for a depiction of this graph for z and X. All paths from s to t in this graph correspond to a MPSC of z by X composed of only set maximal matches. Furthermore, all MPSCs of z by X correspond to a path from s to t in the graph where the i-th positional substring is contained in the i-th set maximal match position in the path.

This graph can be constructed very efficiently. As before, there are $O(S) \subseteq O(N)$ nodes in the graph. These nodes can be found in $O(N)$ time. All i-th required regions can be obtained in $O(N)$ time. This is done by first calculating a leftmost MPSC and a rightmost MPSC of z by X in $O(N)$ time[22]. Then, each i-th required region can be obtained in constant time by Lemma 5. The set maximal match positions can be obtained in $O(N)$ time [24]. The last step is the creation of the edges between nodes. Although there may be $O(S^2) \subseteq O(N^2)$ edges in this graph, we avoid the explicit construction of all of them by exploiting the following property. Call R_i the set of nodes that contained the i-th required region. Call $R_i[j]$ the node in R_i with the j-th smallest starting position, $j \in \{0, \ldots, |R_i|-1\}$. Then,

Lemma 6. *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 9. $R_i[j]$ 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. □

Therefore, during the construction of our graph, we don't directly construct all $O(N^2)$ edges. For every node $R_i[j+1]$, we only construct edges to nodes that $R_i[j]$ does not have an edge to. Although, there may be $O(N^2)$ edges in the complete graph, we fully encode them with $O(S) \subseteq O(N)$ edges. With these edges in place and the property in Lemma 6 in mind, we have constructed a graph that fully describes the solution space of MPSCs of z by X in $O(N)$ time. The combination of this graph and the PBWT supports many efficient queries through the exploitation of Lemma 6. This includes the counting of the number of set maximal match only MPSCs of z by X, the counting of the number of MPSCs a positional substring is an element of, and the counting of Length Maximal MPSCs, all in $O(N)$ time. Furthermore, each of these sets of MPSCs can be enumerated in $O(N+S_c)$ time, where S_c is the number of MPSCs outputted. These problems are very useful for the efficient and accurate imputation and phasing of haplotypes.

Length Maximal MPSC

The Length Maximal Minimal Positional Substring Cover problem is, given a set X of M strings of and a string z, find a MPSC of z by X that has the largest length out of all MPSCs of z by X. Note that by Claim 2, the length of any MPSC of z by X is less than or equal to $2N$.

Algorithm 11: 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 12: SetMaximalMatchPositions: Output positions of set maximal matches from z to X

```
// Assumes a positional substring cover of z by X exists
```
for $i = 0 \to N-1$ **do**
  if $d_{min}[i] < d_{min}[i+1]$ **then**
    S.push_back($(d_{min}[i], i-1, z)$);
return S;

Lemma 7. *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, $\mathscr{C}$, 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 $\mathscr{C}'$ with a larger length. Such a positional

substring always exists by Claim 7. $\mathscr{C}'$ is an MPSC because it covers all the sites $\mathscr{C}$ covered and has the same size. Its length is larger because the only difference between $\mathscr{C}$ and $\mathscr{C}'$ is a positional substring in $\mathscr{C}$ was replaced by a larger one $\mathscr{C}$. This contradicts the fact $\mathscr{C}$ 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. □

Algorithm 13: Output a Length Maximal MPSC of z by X given a PBWT of X

t, d_{min} = VirtualInsert(z, PBWT);
$\mathscr{C}_l$ = LeftmostMPSC(d_{min}, z);
if $|\mathscr{C}_l| = 0$ **then**
    output "No MPSC exists";
    return;
$\mathscr{C}_r$ = RightmostMPSC(d_{min}, z);
S = SetMaximalMatchPositions(d_{min}, z);
L = LongestPaths($S, \mathscr{C}_l, \mathscr{C}_r$);
$\mathscr{C}$ = Backtracking(L, S);
return $\mathscr{C}$;

Algorithm 14: Backtracking: Output Length Maximal MPSC by backtracking through graph

```
// Assumes a positional substring cover of z by X exists
```
$\mathscr{C} = \{S[0]\}$;
$l = L[0]$;
$s = |S[0]|$;
for $i = 1 \rightarrow |S| - 1$ **do**
    if $L[i] = l - s$ **then**
        $\mathscr{C} = \mathscr{C} \cup S[i]$;
        $l = L[i]$;
        $s = |S[i]|$;
return $\mathscr{C}$;

Considering Lemma 7, we begin with the MPSC graph in Fig. 4.4 and modify it in the following fashion. For every edge (u, v), we assign it a weight of the length of u. In this directed acyclic graph, every directed path from s to t still corresponds to a MPSC of z by X composed of only set maximal matches. As before, every MPSC of z by X composed of only set maximal matches corresponds to a directed path in the graph from s to t, there is a one-to-one correspondence.

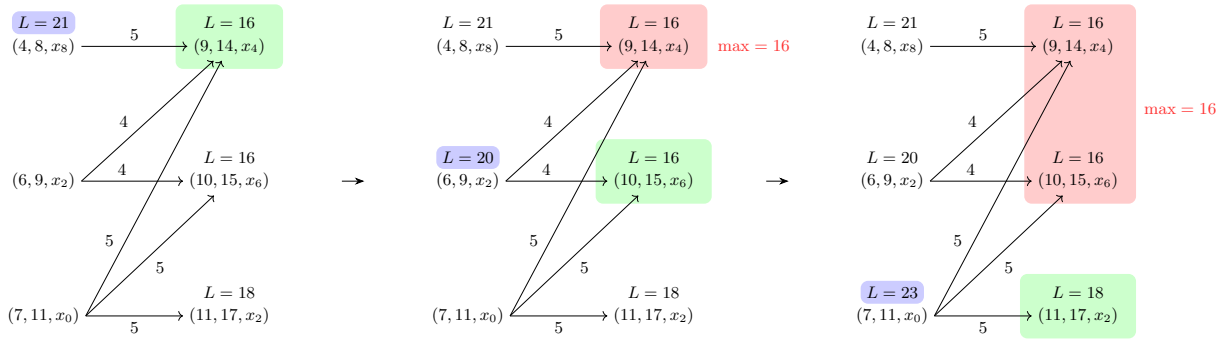

Figure 4.5: Obtaining the longest paths of the set maximal matches in R_i from the longest paths of the set maximal matches in R_{i+1} in $O(|R_i| + |R_{i+1}|)$ time. Subgraph of Fig. 4.4, $i = 2$.

Therefore, the length maximal MPSC of z by X corresponds to a path in the graph by Lemma 7. In fact, there is a one-to-one correspondence between longest directed paths in the graph from s to t and Length Maximal MPSCs of z by X (where the length of a path is the sum of the weights of its edges).

Therefore, we can obtain a Length Maximal MPSC by constructing this graph and finding a longest path in it from s to t. This is easy given a PBWT of X. We begin by finding leftmost and rightmost MPSCs of z by X. This can be done in $O(N)$ given a PBWT of X (where N is $|z|$). This was shown by [22]. We can then obtain all required regions in $O(|C|)$ time, where $|C|$ is the size of an MPSC of z by X. The calculation of each required region takes constant time. They can be calculated simply using the leftmost and rightmost MPSCs and Lemma 5. Next, we output all set maximal match positions and one string in X that each contains each set maximal match in $O(N)$ time. This can be done with a simple modification of the set maximal match query algorithm by [24]. We maintain the sorted order of the set maximal match positions provided by the query algorithm. Lastly, we keep track of which set maximal matches contain which complete required regions. This can be done in a simple sweeping fashion in $O(S)$ time due to Claim 9. So far, the algorithm has taken $O(N)$ time.

Algorithm 15: 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;
```

We now have all the required information to build the graph. We create the nodes of the graph in $O(S)$ time where S is the number of set maximal match positions, $S \leq N$. We create the edges leaving s and entering t in $O(S)$ time. Lastly, for every set maximal match position, if it fully contains a required region, we create an edge from it to nodes whose positional substrings fully contain the next required region and are adjacent or overlapping. This last step of construction of the graph takes $O(S^2) \subseteq O(N^2)$ time. We can find the longest path in this directed acylic graph in $O(V+E) = O(S+E) \subseteq O(S^2) \subseteq O(N^2)$ time. This can be done using a shortest path finding algorithm on the graph with the weights negated. The overall time complexity for this method of outputting a length maximal MPSC is $O(N+S^2)$. However, we have yet to exploit the property of the graph shown in Lemma 6.

Algorithm 16: 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 = (π2(𝒞_r[i − 1]) + 1, π1(𝒞_l[i + 1] − 1, z);
    k = j_old − 1;
    while π1(S[k]) > π1(reqReg) do
        L[k] = −1;
        k − −;
    j = k;
    while π2(S[j − 1]) ≥ π2(reqReg) do
        j − −;
    j_next = j;
    L[j] = |S[j]| + L[j_old];
    j_old + +;
    while j_old ≤ k_old and π2(S[k]) ≥ π1(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 π2(S[j]) ≥ π1(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 π2(S[0]) ≥ π1(S[j_old]) − 1 do
    L[0] = max(L[0], |S[0]| + L[j_old]);
    j_old + +;
return L;
```

We will now show that the longest path in the graph can be obtained in time linear to the number of nodes in the graph, $O(S)$ through the exploitation of Lemma 6. Therefore, a Length Maximal MPSC of z by X can be outputted in $O(N)$ time. Call the set of set maximal match positions (or the corresponding nodes) containing the i-th required region R_i. The key idea is the following.

Given the longest paths from all nodes in R_{i+1} to t, the longest paths from all nodes in R_i to t can be obtained in $O(|R_{i+1}| + |R_i|)$ time. This is despite the fact that there may be $|R_{i+1}| \times |R_i|$ edges between these nodes. We denote the set maximal match of R_i with the j-th smallest starting position as $R_i[j]$. I.E. $R_i[0]$ is the set maximal match in R_i with the smallest starting position, $R_i[1]$ has the next smallest starting position, etc. The calculation of the longest paths from R_i to t in linear time depends on Lemma 6. 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. Given this property, we can calculate the longest paths in a straightforward fashion that evaluates the longest path from each node in R_{i+1} to t at most once. This is done by calculating longest paths of nodes in order of starting position of the corresponding set maximal match, least to greatest. See Fig. 4.5 for a depiction of this process. Pseudocode of this process is provided in Algorithm 16.

After the lengths of the longest paths from each node to t is calculated, the longest path from s to t can be calculated in a simple linear backtracking step. Start with s. For each i-th required region for $i = 1 \rightarrow |\mathscr{C}| - 1$, take the first node with length of longest path to t equal to the length of the longest path to t of the $i-1$-th node in the path so far minus the length of the substring of the $i-1$-th node in the path so far. See Algorithm 14 for the pseudocode of this process. With this process for outputting the longest path in the graph from s to t in $O(S)$ time, we can find the longest path in the graph without explicitly constructing the graph. Therefore, given a PBWT of X, we output a length maximal minimal positional substring cover of z by X in $O(N)$ time. The pseudocode of this algorithm can be seen in Algorithm 13.

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 $O(S) \subseteq O(N)$.

h-Minimal Positional Substring Cover

The h-Minimal Positional Substring Cover problem, is, given a query string z, and a set of strings X, find the smallest cover out of all positional substrings covers $\mathscr{C}$ of z by X where every positional substring in $\mathscr{C}$ is contained in at least h strings in X. The solution to this problem is similar to the solution to the Minimal Positional Substring Cover problem, and it is biologically useful because the large group of similar individuals for every region suggests that they are closely related. Note that Claims 2 to 4 hold for h-MPSCs. Furthermore, with this definition, a 1-MPSC is a MPSC of z by X and a MPSC is a 1-MPSC of z by X.

The h-Minimal Positional Substring Cover problem is similar to the Minimal Positional Substring Cover problem. The main difference is that in the new problem, we only consider positional substrings that have h matches in X. Any h-MPSC will contain a positional substring that contains the last index and is contained in h strings in X by definition. We can replace this positional substring with the largest match ending at index $N-1$ contained in h strings in X. This will result in a set that is still a h-MPSC because it covers all the same sites as the previous set and is the same size. Therefore, we start building a h-MPSC with the longest match ending at index $N-1$ with h matches. Finding this match is easy using the PBWT.

Longest Match ending at k present in h strings in X

Once a query string z is virtually inserted into a PBWT, it is easy to find the largest match ending at index k between z and X that matches at least h strings in X. The idea is to keep track of a window in column $k+1$ of the prefix and divergence arrays. We will keep track of the boundaries of the window, $0 \leq f < g \leq N$. f is the index of the first haplotype in the window and g is the index of the first haplotype after f not in the window. We will also keep track of e, the

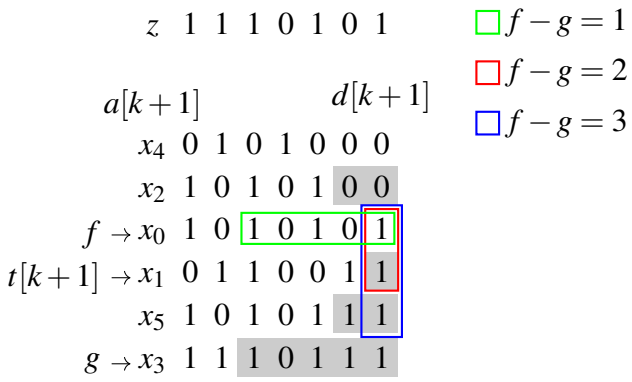

Figure 4.6: Finding the longest match ending at k present in h strings in X for $k = 6$ and $h = 3$. We only depict indices $\{0,\ldots,k\}$ for all strings. Highlighted in gray are divergence vales, $d[k+1][j]$. f and g values depicted are the final f and g values. The window is depicted for $f-g \in \{1,2,3\}$.

starting position of the match. We start with $e = \min(d_z[k+1], d_{belowz}[k+1])$, $f = t[k+1]-1$ if $d_z[k+1] < d_{belowz}[k+1]$, otherwise $f = t[k+1]$. Lastly $g = f+1$. The number of strings in the window at any point is $g-f$. Now, until $g-f = k$, we expand the boundaries of the window to include the next longest match to z. If $d[k+1][f] < d[k+1][g]$, then we decrement f and update e accordingly, $e = \max(d[k+1][f], e)$, $f = f-1$. Otherwise, we update g and e accordingly, $e = \max(d[k+1][g], e)$, $g = g+1$ Overall, the search of this match takes $O(h)$ time. Fig. 4.6 depicts this process.

Algorithm

Then, using similar logic to Lemma 1, we repeatedly add the longest match with h matches ending at the site just before the beginning of the last match to obtain the h-MPSC. See Lemma 2.

Lemma 2 (h-MPSC Modularity)**.** *Given a subset $\mathscr{P}$ of a h-minimal positional substring cover $\mathscr{C}$ of a string z by X that covers all the indices in $\{k,\ldots,N-1\}$ and none of the indices in $\{0,\ldots,k-1\}$ for $0 < k \leq N-1$. Take $m = (i,j,s)$, the longest match ending at $k-1$ from z to X that is contained*

Algorithm 17: h-Minimal Positional Substring Cover

Perform the same steps to virtually insert as in Algorithm 8;
// h-MPSC Output
$\mathscr{C} = \emptyset$;
$k = N$;
while $k > 0$ **do**
    // longest match ending at $k-1$ with h matches in X search
    $e = \min(d_z[k], d_{belowz}[k])$;
    if $d_z[k] < d_{belowz}[k]$ **then**
        $f = t[k] - 1$;
    else
        $f = t[k]$;
    $g = f + 1$;
    while $g - f < k$ **do**
        if $d[k][f] < d[k][g]$ **then**
            $e = \max(d[k][f], e)$;
            $f - -$;
        else
            $e = \max(d[k][g], e)$;
            $g + +$;
    if $e = k$ **then**
        output "No h-positional substring cover of z by X exists.";
        exit;
    $\mathscr{C} = \mathscr{C} \cup \{(e, k-1, z)\}$;
    $k = e$;
output $\mathscr{C}$;

in h strings in X. $\mathscr{P} \cup \{m\}$ *is a subset of a h-MPSC of z by X.*

Proof. If $\mathscr{P}$ is a subset of a h-minimal positional substring cover $\mathscr{C}$ of a string z by X, then the $(|\mathscr{C}| - |\mathscr{P} - 1)$-th positional substring in $\mathscr{C}$ must cover index $k-1$. If it didn't, either the index $k-1$ is not covered by $\mathscr{C}$, or another substring in $\mathscr{C}$ covers index $k-1$. In the first case, our definition of $\mathscr{C}$ is contradicted.

In the second case, if an n-th positional substring of C covers $k-1$, if $n > |\mathscr{C}| - |\mathscr{P}| - 1$, our definition of $\mathscr{P}$ is contradicted. If $n < |\mathscr{C}| - |\mathscr{P}| - 1$,our definition of n-th positional substring is

contradicted (n-th positional substring ends after $(|\mathscr{C}| - |\mathscr{P}| - 1)$-th positional substring), or $\mathscr{C}$ is not a minimal positional substring cover ($(|\mathscr{C}| - |\mathscr{P}| - 1)$-th positional substring starts after $k-1$, in which case it can be removed while maintaining the coverage of all sites).

Now, for any $\mathscr{C}$, replacing the $(|\mathscr{C}| - |\mathscr{P}| - 1)$-th positional substring with longest match ending at $k-1$ between z and h strings in X will yield a h-MPSC of z by X. This is because the new set is the same size as $\mathscr{C}$ and all the sites $\mathscr{C}$ covered are also covered by the new set. Any indices greater than $k-1$ are covered by $\mathscr{P}$ and there are no indices less than $k-1$ that the original $(|\mathscr{C}| - |\mathscr{P}| - 1)$-th positional substring covered that the longest match ending at $k-1$ between z and h strings in X doesn't cover by definition. □

Time Complexity

Given a string z, and a PBWT of a set of strings X, we output a h-MPSC of z by X. Finding each positional substring takes $O(h)$ time. Therefore, this algorithm takes $O(h|\mathscr{C}| + N)$ where $\mathscr{C}$ is the outputted cover. The N part of the time complexity comes from the virtual insertion and the $h|\mathscr{C}|$ component from the cover search and output. See Algorithm 17 for the pseudocode of this algorithm.

Improved Algorithm

We suspect that the previous algorithm outputs a "leftmost" h-MPSC. Here we describe an algorithm that improves on the previous h-MPSC algorithm running time. It outputs an h-MPSC of z by X in $O(N)$ time. We believe the h-MPSC it outputs is "rightmost".

Algorithm 18: 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;
```

The original h-MPSC algorithm traversed the PBWT from index N to 0. Here, we traverse in the opposite direction, from 0 to N. We begin at index 0 and keep track of the block of strings that match with z on the range $[i, j)$. The block is denoted by $[f, g)$ where f is the first string in the block in the prefix sorting and g is the first string not in the block after f. If $f = g$, the block is

empty. At site 0, f and g are initialized to $f = 0$, $g = M$. i and j are initialized to 0. The f and g blocks are updated in constant time per site using the w array between sites. Once the block at site $j+1$ has less than h strings, the positional substring of the previous block $(i, j-1, z)$ is added to the h-MPSC. This is repeated until all sites are covered by a positional substring contained in h strings in X. This algorithm runs in $O(N)$ time because the is traversed from index 0 to N and each site takes constant time to update the f and g block. Furthermore, the addition of a positional substring to the h-MPSC can happen at most once per site and takes constant time. A proof that the positional substring cover is an h-MPSC can be seen through a fairly simple modification of Lemma 2. The pseudocode of this algorithm can be seen in Algorithm 18.

L-MPSC

Here we present and solve a new variation of the MPSC problem, the L-MPSC. An L-MPSC is similar to an MPSC except it requires each of its elements to be at least length L. Specifically, an L-MPSC of a query string z by a set of strings X is a smallest positional substring cover of z by X where every positional substring in the cover has length at least L. This is a problem of biological interest since it guarantees a minimum relatedness at each location in the haplotype threading between the query haplotype and the source of the positional substring in the panel. This may make the L-MPSC more powerful than an MPSC for biological problems such as phasing and imputation.

It turns out that the L-MPSC has many of the same properties as the MPSC. In fact, we show leftmost, rightmost, and set maximal match only L-MPSCs with the same time complexity as the MPSC given a PBWT of the set X. We begin by proving that the L-MPSC has some of the same properties as the MPSC.

Claim 10. *An L-MPSC of z by X exists iff every index of z is contained in a positional substring*

that is length L or longer and contained in z and X

Proof. If there exists and L-MPSC of z by X, then every index of z is contained in a positional substring that is length L or longer and contained in z and X by definition of L-MPSC.

If for all indices of z, there exists a positional substring that contains it, is at least length L, and is contained in z and X. Then the union of all such positional substrings is a positional substring cover of z by X where every element is at least length L. Then a smallest such set (an L-MPSC of z by X) exists. □

Claim 11. *For any L-MPSC of z by X, $\mathscr{C}^{\mathscr{L}}$, every index of z is contained in at most two of its positional substrings.*

Proof. Suppose an L-MPSC exists with an index that is contained more than two of its positional substrings. Out of the positional substrings that contain this index, remove one that doesn't have the smallest starting point or the largest ending point (if they all have the smallest starting point or the largest ending point, remove any of them). The resulting set of positional substrings is a positional substring cover of z by X where each of its elements is at least length L. Furthermore it is smaller than the original L-MPSC of z by X, a contradiction. □

Claim 12. *For any L-MPSC of z by X, the starting points of its positional substrings are unique and their ending points are unique.*

Proof. Suppose an L-MPSC exists where two of its positional substrings share a starting point. The smaller one could be removed from the set (or any if they are equal length). The new set is smaller than the original L-MPSC and is a positional substring cover of z by X with every element at least length L (contradiction). Similar logic applies to two positional substrings sharing an ending point. □

Claim 13. *For any L-MPSC of z by X, the positional substring with the i-th smallest starting point has the i-th smallest ending point.*

Proof. If this property is not true, then there may exist an two positional substrings p, q in an L-MPSC where $\pi_1(p) < \pi_1(q)$ and $\pi_2(p) > \pi_2(q)$. In this case, q is fully contained in p and can be removed from the set. The new set is still a positional substring cover of z by X with all of its elements at least length L and is smaller than the original L-MPSC of z by X. This is a contradiction. □

Rightmost L-MPSC

Using these properties, we build a Rightmost L-MPSC. As before, we use Claim 13 to refer to the i-th positional substring in an L-MPSC as the positional substring with the i-th smallest starting position (or equivalently, ending position), 0-indexed. A rightmost L-MPSC is an L-MPSC where the i-th positional substring ends as late as the i-th positional substring of every L-MPSC of z by X. We solve the rightmost L-MPSC problem using the following lemma.

Lemma 6 (L-MPSC Modularity)**.** *Consider an L-MPSC of z by X, $\mathscr{C}$. For all subsets $\mathscr{P}$ of $\mathscr{C}$ that cover all the indices in $\{0,\ldots,k\}$, and none of the indices in $\{k+1,\ldots,N-1\}$, $P \cup (i,j,s)$ is also a subset of an L-MPSC of z by X if (i,j,s) is at least length L, is contained in z and X, contains index $k+1$, and has the largest ending point out of all matches between z and X that contain index $k+1$ and are at least length L.*

Proof. If $\mathscr{P}$ is a subset of $\mathscr{C}$ and covers all the indices in $\{0,\ldots,k\}$, and none of the indices in $\{k+1,\ldots,N-1\}$, then the $|P|$-th positional substring in $\mathscr{C}$ must contain position $k+1$.

For any $\mathscr{C}$, replacing its $|P|$-th positional substring with (i,j,s) yields an L-MPSC since all of

the sites covered previously remain covered, (i, j, s) is at least length L, and the new positional substring cover is the same size as $\mathscr{C}$. □

We construct an L-MPSC of z by X given a PBWT of X in the following way. Begin with a virtual insertion of z into the PBWT of X. Using the calculated divergence arrays, compute the array b_L. b_L is an array of length N where $b_L[i]$ is the ending index of the match with the largest ending point of all matches between z and X length L or more that contain index i. If no such match exists, $b_L[i] = -1$. In order to compute b_L, we use the b and d arrays. b is an array of length N where $b[i]$ contains the ending point of the longest match between z and X starting at i. d is an array of length N where $d[i]$ contains the starting point of the longest match between z and X ending at i. b and d can be computed in $O(N)$ time from the divergence values calculated during virtual insertion. See Rightmost Minimal Positional Substring Cover in Chapter 4 for the computation of b. $d[i] = min(d_z[i+1], d_{belowz}[i+1])$. Set $b_L[0] = b[0]$ if $b[0] + 1 \geq L$, otherwise $b_L[0] = -1$.

We compute b_L using the following recurrence relation.

$$b_L[i] = \begin{cases} b[i] & \text{if } b[i] > b[i-1] \text{ and } b[i] - i + 1 \geq L \\ b_L[i-1] & \text{otherwise, if } b_L[i-1] \neq -1 \text{ and } b_L[i-1] \geq i \\ -1 & \text{otherwise} \end{cases}$$

This allows us to compute b_L in $O(N)$ time. Using b_L and Lemma 6, obtaining an L-MPSC of z by X is straightforward. Consider k the first index not covered yet (initialized at 0). Until $k = N$, do the following:

1. If $b_L[k] = -1$, quit, no L-MPSC exists.
2. Add $(d[b_L[k]], b_L[k], z)$ to the cover.

3. Set $k = b_L[k] + 1$.

At the end of this process, the cover is an L-MPSC of z by X. This process takes $O(|\mathscr{C}|)$ time, therefore an L-MPSC of z by X can be found in $O(N)$ time given a PBWT of X. Now, we show that this L-MPSC is rightmost using the following claim.

Claim 14. *If the i-th substring in a rightmost L-MPSC ($\mathscr{C}^{\mathscr{L}}$) of z by X ends at index j, every $i+1$-th substring in an L-MPSC of z by X contains index $j+1$.*

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

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

Now, call the L-MPSC outputted by this method $\mathscr{C}^{\mathscr{L}}$. $\mathscr{C}^{\mathscr{L}}[0]$ is a subset of a rightmost L-MPSC since the 0-th positional substring of any L-MPSC of z by X must contain index 0 and $\mathscr{C}^{\mathscr{L}}[0]$ has the rightmost ending point of all such positional substrings. By Claim 14, index $\pi_2(\mathscr{C}^{\mathscr{L}}[0]) + 1$ must be contained in every 1-th positional substring in an L-MPSC of z by X. The rightmost ending position of any positional substring contained in z and X that contains index $\pi_2(\mathscr{C}^{\mathscr{L}}[0]) + 1$ and

is at least length L is precisely $\pi_2(\mathscr{C}^{\mathscr{L}}[1])$. Therefore $\{\mathscr{C}^{\mathscr{L}}[0], \mathscr{C}^{\mathscr{L}}[1]\}$ is a subset of a rightmost L-MPSC of z by X. This logic can be repeated for all positional substrings in $\mathscr{C}^{\mathscr{L}}$ to show that the $\mathscr{C}^{\mathscr{L}}$ is a Rightmost L-MPSC of z by X.

Leftmost L-MPSC

We build a Leftmost L-MPSC using analogous properties.

Claim 15. *If the i-th substring in a leftmost L-MPSC ($\mathscr{C}^{\mathscr{L}}$) of z by X starts at index j, every $i-1$-th substring in an L-MPSC of z by X contains index $j-1$.*

Proof. Suppose there existed an L-MPSC ($\mathscr{D}$) of z by X such that $\pi_1(\mathscr{C}^{\mathscr{L}}[i]) - 1 > \pi_2(\mathscr{D}[i-1])$. Then, since $\mathscr{D}$ is a cover of z by X, it contains a positional substring that contains index $\pi_1(\mathscr{C}^{\mathscr{L}}[i]) - 1$, call it $\mathscr{D}[j]$. If $j < i-1$, the definition of i-th positional substring is contradicted since $\pi_2(\mathscr{D}[j]) > \pi_2(\mathscr{D}[i-1]$ and $j < i-1$. If $j > i-1$, $j = i$ by definition of i-th positional substring, and $\mathscr{C}^{\mathscr{L}}$ is not leftmost since $\pi_1(\mathscr{C}[i]) > \pi_1(\mathscr{D}[i])$. Therefore, no such i and $\mathscr{D}$ exist.

Suppose there existed an L-MPSC ($\mathscr{D}$) such that $\pi_1(\mathscr{C}^{\mathscr{L}}[i]) - 1 < \pi_1(\mathscr{D}[i-1])$. Then the set $\{\mathscr{D}[j] : 0 \leq j < i-1\}$ covers the indices $[0, \pi_1(\mathscr{D}[i-1]) - 1]$ with $i-1$ positional substrings and the set $\{\mathscr{C}^{\mathscr{L}}[j] : i \leq j < |\mathscr{C}^{\mathscr{L}}|\}$ covers the indices $[\pi_1(\mathscr{C}^{\mathscr{L}}[i]), N-1]$ with $|\mathscr{C}^{\mathscr{L}}| - i$ positional substring. Their union is a positional substring cover of z by X with $|\mathscr{C}^{\mathscr{L}}| - 1$ positional substrings. This contradicts the assumption that $\mathscr{C}^{\mathscr{L}}$ is a minimal positional substring cover of z by X, therefore, no such i and $\mathscr{D}$ exist. □

Given Claim 15, we build a Leftmost L-MPSC of z by X using a method similar to the previous. The difference is, this time we build the cover from the right adding the match containing index $k-1$ with the smallest starting point out of those with length L or more. In order to do this, we

build the d_L array. d_L is an array of length N where $d_L[i]$ is the starting index of the match with the smallest starting point of all matches between z and X length L or more that contain index i. If no such match exists, $d_L[i] = -1$. In order to compute d_L, we use the d array. Set $d_L[N-1] = d[N-1]$ if $N - d[N-1] \geq L$, otherwise $d_L[N-1] = -1$.

$$d_L[i] = \begin{cases} d[i] & \text{if } d[i] < d[i-1] \text{ and } i - d[i] + 1 \geq L \\ d_L[i+1] & \text{otherwise, if } d_L[i+1] \neq -1 \text{ and } d_L[i+1] \geq i \\ -1 & \text{otherwise} \end{cases}$$

Therefore, we can compute d_L in $O(N)$ time. Using d_L and Claim 15, obtaining a leftmost $L-MPSC$ of z by X is straightforward. Set $k = N-1$. Until $k = -1$

1. If $d_L[k] = -1$, quit, no L-MPSC exists.
2. Add $(d_L[k], b[d_L[k]], z)$ to the cover.
3. Set $k = d_L[k] - 1$.

At the end of this process, the cover is a Leftmost L-MPSC of z by X by Claim 15. Therefore, a Leftmost (and a Rightmost) L-MPSC of z by X can be found in $O(N)$ time given a PBWT of X. The L-MPSCs outputted by both of these methods are composed of only set maximal matches from z to X.

Boundary cases in implementation

In the implementation of the algorithms discussed in this chapter, there are boundary cases that need to be accounted for which aren't dealt with in the pseudocode. These boundary cases are left

out of the pseudocode because their handling would make the pseudocode unnecessarily long and difficult to understand. We briefly discuss these boundary cases here and how to handle them. For virtual insertion $t[k]$ may equal M, while there is no $a[k][M]$, this represents the fact that z would sort below every string in X in $a[k]$. While there is no value $w[k][M][c]$, it is simple to calculate in constant time what this value should be. $w[k][M][c]$ is the same as $w[k][0][c+1]$, where $c+1$ is the lexicographically smallest character that is larger than c. This can also be calculated in constant time as $w[k][M][c] = w[k][M-1][c]+1$ if $y_{k,M-1}[k] = c$, or $w[k][M][c] = w[k][M-1][c]$ otherwise. It should also be checked if z is longer than the strings in X, if so, no cover exists. Lastly, for the h-MPSC problem, there are two boundary cases. Firstly, if $h > |X|$, no cover exists. Secondly, while incrementally widening the window to contain h strings, care should be taken to avoid trying to obtain the divergence value $d[k][-1]$ or $d[k][M]$. In other words, if $f = 0$ or $g = M$, stop trying to increment f or decrement g respectively.

Results

Haplotype Threading Properties

We explored the properties of haplotype threadings of the MPSC formulation. The dataset used was the UK Biobank (UKB) [5]. The UKB has 974,818 haplotypes and around 700,000 sites (microarray). We used chromosome 21, which has 9,793 sites. For each haplotype in the UKB, we run our method to identify an MPSC of it using all other haplotypes in the UKB as the reference panel. We also evaluate a rudimentary method of handling mismatches in the MPSC formulation of haplotype threading by using P-smoother [28]. P-smoother is a method for smoothing out sporadic mismatches in otherwise well-matched PBWT blocks, to attempt to remove very rare mutations and genotyping errors from the British only UKB panel. P-smoother was run on the default settings and flipped the alleles of roughly 1.4% of the data. The smoothed panel is expected to tolerate

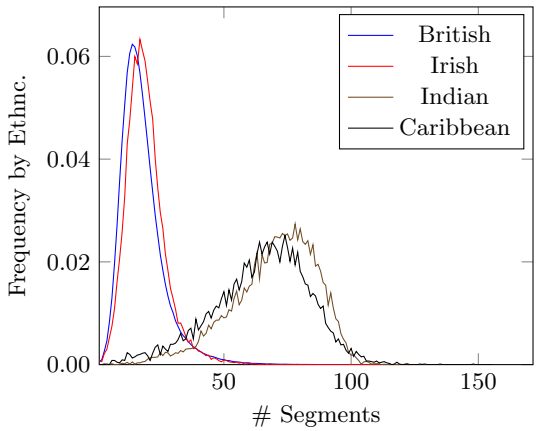

Figure 4.7: Frequency within self reported ethnic background of # of segments in haplotype threading of haplotypes in UK Biobank. Haplotype threading is generated with reference panel as the rest of the UK Biobank. Frequency is plotted by self reported ethnic background for the four most common in the UK Biobank: British, Irish, Indian, and Caribbean, with 860,584, 25,436, 11,320, and 8,598 haplotypes respectively.

mismatches, resulting in longer match segments, and smaller MPSC sizes.

We found that the MPSC segment count of an individual has an overall distribution due to the fact that not every one's relatives are sampled evenly. However, the mode of MPSC segment count is inversely correlated with the number of templates with closely related ethnic background (Fig. 4.7). This behavior is even more clear when we run the experiment with varied sample sizes (Fig. 4.8). We plot the number of segments in each haplotype threading by self reported ethnic background in Fig. 4.7. The x-axis is the number of segments in the haplotype threading and the y-axis is the frequency of that number within each self reported ethnic background. The plotted ethnic backgrounds are the 4 most commonly reported ethnic backgrounds in the UKB: British, Irish, Indian, and Caribbean (860,584, 25,436, 11,320, and 8,598 haplotypes respectively). A haplotype is classified as one of these ethnic backgrounds if it was the first ethnic background the individual who owns the haplotype reported themselves as. We also plot MPSC size distributions for random subsets of the UKB of varying sizes. No threading was found for $(7.3\%, 0.32\%, 0.003\%$, and $0\%)$

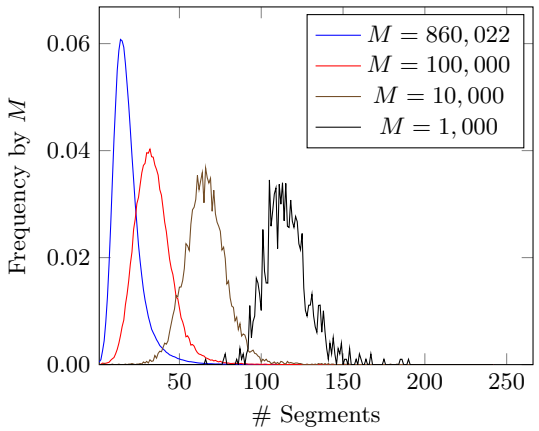

Figure 4.8: Frequency by panel size of # of segments in British only haplotype threading of haplotypes in UK Biobank. Haplotype threading is generated with reference panel random subset of the British only haplotypes in the UK Biobank. Plotted by panel size for $M \in \{1{,}000, 10{,}000, 100{,}000, 860{,}022\}$. Note that frequencies exclude haplotypes for which no threading was found.

of the haplotypes for M=(1,000, 10,000, 100,000, and 860,022) respectively. $M = 10$ and 100 were also tried, but no haplotype threadings were found in either panel. Haplotypes without MPSCs are left out of the frequency calculation.

A more interesting property we consider is the MPSC solution space. This is the number of possible coverings of the query with equally small number of switches, which would be all considered a Viterbi solution for the LS model. We plot the count of Set Maximal Match only MPSCs for each haplotype in the British only UKB dataset (860,022 haplotypes) in Fig. 4.9 in ascending order. The median solution space size was 120. The 90th and 99th percentile solution space sizes were 72,756 and 3.4×10^9 respectively. As expected, the median solution space size of the P-smoothed panel was 40. The fact that the number of solutions in MPSC solution space is surprisingly small validated that UKB belongs to the regime where MPSC could be a much more efficient alternative for the fully-parameterized Li & Stephens model.

The last property of the MPSC formulation of haplotype threading we explore is the distributions

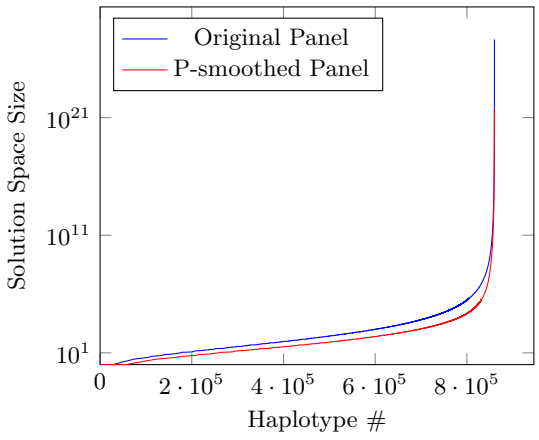

Figure 4.9: The number of Set Maximal Match only MPSCs per haplotype in British only UK Biobank panel in ascending order.

of lengths of the Length Maximal MPSCs. Note this length l is $N \leq l \leq 2N$, and thus the size of the overlap region, $l - N$ or $\frac{l-N}{N}$, provides one way of quantifying the potential computations (much more than a linear factor!) saved by MPSC compared to enumerating solutions of standard LS. We plot this length distribution of Length Maximal MPSCs for British only haplotypes in Fig. 4.10. P-smoother results in a smaller Length Maximal MPSC length. The length maximal MPSC provides about at average 1.21X (1.20X after smoothing) coverage of the chromosome, indicating about 20% of the genome belongs to the overlap regions of Length Maximal MPSC where location of recombination breakpoints cannot be unequivocally determined.

Run Time

We measure the run time of obtaining the Length Maximal MPSC for various M and N. Note that the run time of obtaining a Length Maximal MPSC is strictly larger than the run time of obtaining an MPSC, a Leftmost MPSC, a Rightmost MPSC, and a Set Maximal Match only MPSC. This is because the first three algorithms are subroutines of the Length Maximal MPSC algorithm. The

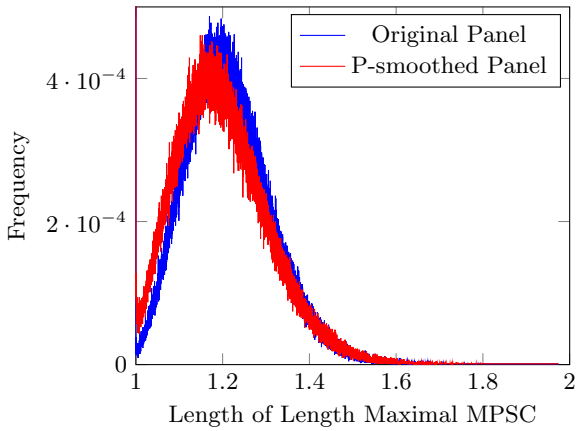

Figure 4.10: The distributions of lengths of Length Maximal MPSCs for all British only haplotypes in the UKB. Length is represented as a ratio of genome length. The min ratio of an MPSC is 1 and the max ratio is 2. The min and max ratios observed are 1 and 1.974.

run time of obtaining a Length Maximal MPSC is strictly larger than the run time of obtaining a Set Maximal Match only MPSC because a Length Maximal MPSC is a Set Maximal Match only MPSC. The results of these tests can be seen in Fig. 4.11.

Imputation Benchmark

We implemented an imputation benchmark to demonstrate the usefulness of the MPSC formulation of haplotype threading. The imputation algorithm is naive outside of the haplotype threading: Given a haplotype threading of the haplotype to be imputed at a site, obtain the positional substrings adjacent to it in the haplotype threading. For every haplotype that contains one of these positional substrings, vote once per positional substring it contains towards the allele this haplotype has at the site to be imputed. Impute an allele if the votes are unanimous. We also evaluate the use of a P-smoothed panel. In this method, the haplotype threading is generated using the P-smoothed panel and query haplotype. Votes are counted for haplotypes who have the positional substring in the P-smoothed panel. However, allele votes are counted using the original panel.

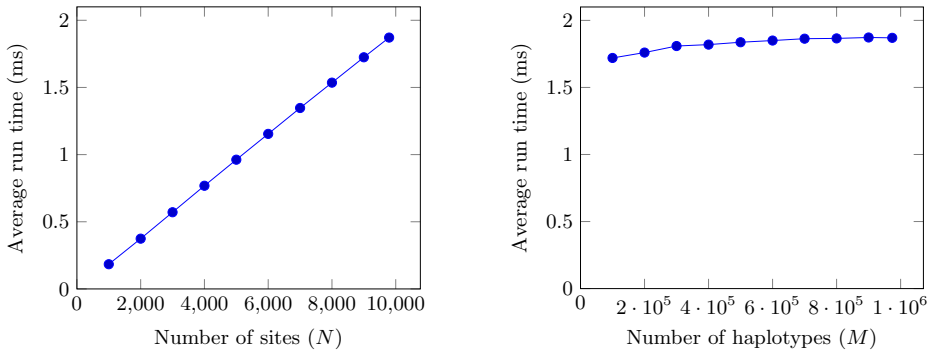

Figure 4.11: Run time by varying M and N. Average run time for the Length Maximal MPSC computation of 1,000 random haplotypes in the UKB British only dataset for varying number of sites, $N \in \{1{,}000, 2{,}000, 3{,}000, 4{,}000, 5{,}000, 6{,}000, 7{,}000, 8{,}000, 9{,}000, \text{ and } 9{,}793\}$, 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.

This imputation method generalizes to all of the variations of MPSC haplotype threading, therefore we test it on many of the MPSC variations, including Length Maximal MPSC, voting by Set Maximal MPSC Solution Space, and the h-MPSC with various choices of h. The imputation benchmark was performed in the following fashion: Select 1,000 random haplotypes from the UKB British only dataset (chromosome 21). Remove a random 8,814 sites (90%) from these haplotypes. Impute these haplotypes using the rest of the UKB British only haplotypes ($M =$ 859,022) as the reference panel using the above-mentioned MPSC-based imputation methods and Beagle (version 5.4), a state of the art imputation method [2].

As shown in Table 4.2, all tested MPSC-based imputation methods have an imputation accuracy comparable to Beagle. Interestingly, while MPSC-based methods do not cover all sites, the sites they covered are mostly “easier" sites as Beagle also have higher accuracies on those sites than the average of all sites. Over the covered sites, MSPC-based methods have roughly same accuracies compared to Beagle. Importantly, for a few cases, MPSC-based methods outperform Beagle,

Table 4.2: Imputation performance in percentages on a random 1,000 British only UKB haplotypes

Method	Original Panel			P-smoothed Panel		
	Accuracy			Accuracy		
	Beagle	MPSC	Sites Imputed	Beagle	MPSC	Sites Imputed
Beagle 5.4	97.94		100.00			
LM MPSC	98.45	98.43	97.15	98.43	**98.46**	97.23
SM MPSC SS	99.10	**99.19**	88.20	99.05	**99.21**	88.22
2-MPSC	98.70	98.32	93.14	98.67	98.36	93.32
3-MPSC	98.89	98.62	90.21	98.87	98.66	90.33
4-MPSC	99.00	98.79	87.81	98.99	98.84	87.92
5-MPSC	99.12	98.96	85.71	99.10	98.99	85.86
8-MPSC	99.30	99.20	80.45	99.27	99.21	80.69
16-MPSC	99.52	99.50	71.68	99.49	**99.50**	72.01

Reference panel is the rest of the UKB British only haplotypes. Imputed haplotypes have a random 90% of their sites missing. For each MPSC imputation method, its accuracy and Beagle's accuracy on the imputed sites are shown. LM MPSC stands for Length Maximal MPSC and SM MPSC SS stands for Set Maximal MPSC Solution Space. MPSC imputation methods that have higher accuracy than Beagle on imputed sites are in bold.

especially for Set Maximal MPSC Solution Space (SM MPSC SS), where MPSC methods beat Beagle in both based on the original panel and the P-smoothed panel. This is useful because one can immediately improve the current Beagle results by repainting the Beagle imputation results with MPSC results where MPSC methods cover. Our results also offer a way of studying behaviors of Li & Stephens-based methods. It seems when we increase the h in h-MPSC, the coverage of MPSC is gravitated towards "easier" regions where both Beagle and MPSC methods have high accuracy. Thus MPSC's h offers a measure of imputation confidence. Finally, the P-smoothed panel increases the power and accuracy of every MPSC threading imputation method.

Discussion

In this chapter, we have defined and proposed the Minimal Positional Substring Cover problem as a solution to the haplotype threading problem. We proved useful properties of Minimal Positional Substring Covers and provided a solution to the MPSC given a PBWT of X that takes time linear to the length of the query string. We also discussed variations of the Minimal Positional Substring Cover problem: leftmost, rightmost, and set maximal MPSCs. We provided solutions to these problems with the same time complexity as the original solution. Lastly, we proposed and solved biologically useful variants, the h-MPSC problem, Length Maximal MPSC, and L-MPSC. These problems were all solved in the same time complexity as the original MPSC problem.

In doing so, we introduced the MPSC graph, a method of representing the solution space of MPSC haplotype threadings of a haplotype that allows many efficient algorithms. These include the enumeration of all MPSCs, Set Maximal Match only MPSCs in optimal time. It also allows the counting of the number of MPSCs a particular positional substring is part of in $O(N)$ time.

Beyond algorithmic contributions, our algorithmic developments established the theoretical basis for linking PBWT and LS-style haplotype threading. When using these algorithms for analyzing properties of MPSC haplotype threadings for the UK Biobank dataset, we demonstrate the usefulness of the MPSC haplotype threadings through an imputation benchmark. We showed that, while the simple MPSC-based imputation does not impute all sites, when it does, the accuracy may be higher than the state-of-the-art imputation method Beagle. Especially, the variations of MPSC algorithms that offer Solution Space, Length Maximal, and high h (≥ 16) h-MPSC can outperform Beagle. In addition, our imputation results verified that P-smoother can be leveraged to soften the mismatch-intolerant MPSC solutions and makes MPSC more robust to real data. We believe further developments based on our sets of algorithms will empower practical applications based on LS such as phasing and imputation.

CHAPTER 5: CONCLUSION

In this dissertation, we have developed algorithms on the PBWT related to IBD segment detection, haplotype threading, haplotype phasing, and haplotype imputation. We also developed a variation of the PBWT, dynamic PBWT, that allows efficient insertion and deletion haplotypes.

In Chapter 3, we developed dynamic PBWT (d-PBWT). The d-PBWT maintains most of the capabilities of the PBWT with the same time complexity as the PBWT. Furthermore, we provide algorithms for insertion and deletion of haplotypes into the d-PBWT in average case $O(N)$ time. Furthermore we improved upon the query algorithms available for the PBWT. Set maximal and long match query are important problems for the detection of IBD segments. The previous set maximal match query algorithm was described by Durbin. He claimed $O(N)$ time complexity [8]. The previous long match query algorithm was described by Naseri et al. [17]. They claimed $O(N+c)$ time complexity where c is the number of long matches outputted. We have shown both of these claims to be incorrect. Furthermore, we have described set maximal match and long match query algorithms that do achieve these time complexities ($O(N)$ and $O(N+c)$ respectively).

In Chapter 4, we develop a new formulation of haplotype threading and solve it using the PBWT. We introduced the Minimal Positional Substring Cover (MPSC) problem, where a query haplotype is represented by a smallest possible set of overlapping copies of segments of haplotypes from a reference panel. We provide a solution to the MPSC problem in $O(N)$ time given a PBWT of the reference panel. We also solve variants of the problem: leftmost MPSC, rightmost MPSC, set maximal match only MPSC. Using these, we are able to represent the solution space of all MPSCs of a query haplotype through a reference panel. Then, we show how to output the biologically useful solutions of length maximal MPSCs, h-MPSCs, and L-MPSCs. We show how to solve each of these problems in $O(N)$ time given a PBWT of the reference panel. We also show that all possible

set maximal match only MPSCs can be counted and enumerated efficiently. Finally, we show that the MPSC formulation of haplotype threading is powerful with an imputation benchmark.

Future work in this area includes the development of an imputation tool using the MPSC formulation of haplotype threading, exploring a diploid MPSC formulation, and the development of a phasing tool using the MPSC. Furthermore, compression of genome datasets using the PBWT may be improved through the use of a graph variation of the PBWT.

REFERENCES

[1] Jarno Alanko et al. “Finding all maximal perfect haplotype blocks in linear time”. *Algorithms for Molecular Biology* 15.1 (2020), p. 2.

[2] Brian L Browning, Ying Zhou, and Sharon R Browning. “A one-penny imputed genome from next-generation reference panels”. *The American Journal of Human Genetics* 103.3 (2018), pp. 338–348.

[3] Brian L Browning et al. “Fast two-stage phasing of large-scale sequence data”. *The American Journal of Human Genetics* 108.10 (2021), pp. 1880–1890.

[4] Michael Burrows and David J Wheeler. “A block-sorting lossless data compression algorithm”. *Digital Equipment Corporation, Tech. Rep. 124.* (1994).

[5] Clare Bycroft et al. “The UK Biobank resource with deep phenotyping and genomic data”. *Nature* 562.7726 (2018), pp. 203–209.

[6] Sayantan Das et al. “Next-generation genotype imputation service and methods”. *Nature genetics* 48.10 (2016), pp. 1284–1287.

[7] Olivier Delaneau et al. “Accurate, scalable and integrative haplotype estimation”. *Nature communications* 10.1 (2019), pp. 1–10.

[8] Richard Durbin. “Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT)”. *Bioinformatics* 30.9 (2014), pp. 1266–1272.

[9] Jordan M. Eizenga et al. “Pangenome Graphs”. *Annual Review of Genomics and Human Genetics* 21.1 (2020). PMID: 32453966, pp. 139–162. DOI: 10.1146/annurev-genom-120219-080406. eprint: https://doi.org/10.1146/annurev-genom-120219-080406. URL: https://doi.org/10.1146/annurev-genom-120219-080406.

[10] William A Freyman et al. “Fast and Robust Identity-by-Descent Inference with the Templated Positional Burrows–Wheeler Transform”. *Molecular Biology and Evolution* 38.5 (Dec. 2020), pp. 2131–2151. ISSN: 1537-1719. DOI: 10.1093/molbev/msaa328. eprint: https://academic.oup.com/mbe/article-pdf/38/5/2131/37799064/msaa328.pdf. URL: https://doi.org/10.1093/molbev/msaa328.

[11] E Garrison et al. “Variation graph toolkit improves read mapping by representing genetic variation in the reference”. *Nature Biotechnology* 36.9 (2018), p. 875.

[12] Na Li and Matthew Stephens. “Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data”. *Genetics* 165.4 (2003), pp. 2213–2233.

[13] Po-Ru Loh et al. “Reference-based phasing using the Haplotype Reference Consortium panel”. *Nature Genetics* 48.11 (2016), p. 1443.

[14] Gerton Lunter. “Haplotype matching in large cohorts using the Li and Stephens model”. *Bioinformatics* 35.5 (2019), pp. 798–806.

[15] Ardalan Naseri, Degui Zhi, and Shaojie Zhang. “Discovery of runs-of-homozygosity diplotype clusters and their associations with diseases in UK Biobank”. *medRxiv* (2020).

[16] Ardalan Naseri, Degui Zhi, and Shaojie Zhang. “Multi-allelic positional Burrows-Wheeler transform”. *BMC bioinformatics* 20.11 (2019), p. 279.

[17] Ardalan Naseri et al. “Efficient haplotype matching between a query and a panel for genealogical search”. *Bioinformatics* 35.14 (2019), pp. i233–i241.

[18] Ardalan Naseri et al. “RaPID: ultra-fast, powerful, and accurate detection of segments identical by descent (IBD) in biobank-scale cohorts”. *Genome Biology* 20.1 (2019), p. 143.

[19] Adam M Novak, Erik Garrison, and Benedict Paten. “A graph extension of the positional Burrows–Wheeler transform and its applications”. *Algorithms for Molecular Biology* 12.1 (2017), p. 18.

[20] Yohei M Rosen and Benedict J Paten. “An average-case sublinear forward algorithm for the haploid Li and Stephens model”. *Algorithms for Molecular Biology* 14.1 (2019), pp. 1–12.

[21] Simone Rubinacci, Olivier Delaneau, and Jonathan Marchini. “Genotype imputation using the positional burrows wheeler transform”. *PLoS genetics* 16.11 (2020), e1009049.

[22] Ahsan Sanaullah, Degui Zhi, and Shaoije Zhang. “Haplotype Threading Using the Positional Burrows-Wheeler Transform”. In: *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[23] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. “d-PBWT: Dynamic Positional Burrows-Wheeler Transform”. In: *Research in Computational Molecular Biology*. Ed. by Russell Schwartz. Cham: Springer International Publishing, 2020, pp. 269–270. ISBN: 978-3-030-45257-5.

[24] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. “d-PBWT: dynamic positional Burrows-Wheeler transform”. *Bioinformatics* 37.16 (2021), pp. 2390–2397.

[25] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. “Minimal Positional Substring Cover: A Haplotype Threading Alternative to Li & Stephens Model”. In: *Research in Computational Molecular Biology*. Ed. by Haixu Tang. Cham: Springer Nature Switzerland, 2023, pp. 249–250. ISBN: 978-3-031-29119-7.

[26] Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. “Minimal Positional Substring Cover: A Haplotype Threading Alternative to Li & Stephens Model”. *bioRxiv* (2023). DOI: 10.1101/2023.01.04.522803. eprint: https://www.biorxiv.org/content/early/2023/01/

06/2023.01.04.522803.full.pdf. URL: https://www.biorxiv.org/content/early/2023/01/06/2023.01.04.522803.

[27] Jouni Sirén et al. "Haplotype-aware graph indexes". *Bioinformatics* 36.2 (July 2019), pp. 400–407. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btz575. URL: https://doi.org/10.1093/bioinformatics/btz575.

[28] William Yue et al. "P-smoother: efficient PBWT smoothing of large haplotype panels". *Bioinformatics Advances* 2.1 (2022), vbac045.

[29] Ying Zhou, Sharon R. Browning, and Brian L. Browning. "A Fast and Simple Method for Detecting Identity-by-Descent Segments in Large-Scale Data". *The American Journal of Human Genetics* 106.4 (2020), pp. 426–437. ISSN: 0002-9297. DOI: https://doi.org/10.1016/j.ajhg.2020.02.010. URL: https://www.sciencedirect.com/science/article/pii/S0002929720300525.