Bioinformatics, 35, 2019, i233–i241
doi: 10.1093/bioinformatics/btz347
ISMB/ECCB 2019

OXFORD

Efficient haplotype matching between a query and a panel for genealogical search

Ardalan Naseri[1,†], Erwin Holzhauser[1,†], Degui Zhi[2,*] and Shaojie Zhang [1,*]

[1]Department of Computer Science, University of Central Florida, Orlando, FL 32816, USA and [2]School of Biomedical Informatics and School of Public Health, University of Texas Health Science Center at Houston, Houston, TX 77030, USA

*To whom correspondence should be addressed.
†The authors wish it to be known that, in their opinion, the first two authors should be regarded as Joint First Authors.

Abstract

Motivation: With the wide availability of whole-genome genotype data, there is an increasing need for conducting genetic genealogical searches efficiently. Computationally, this task amounts to identifying shared DNA segments between a query individual and a very large panel containing millions of haplotypes. The celebrated Positional Burrows-Wheeler Transform (PBWT) data structure is a pre-computed index of the panel that enables constant time matching at each position between one haplotype and an arbitrarily large panel. However, the existing algorithm (Durbin's Algorithm 5) can only identify set-maximal matches, the longest matches ending at any location in a panel, while in real genealogical search scenarios, multiple 'good enough' matches are desired.
Results: In this work, we developed two algorithmic extensions of Durbin's Algorithm 5, that can find all *L*-long matches, matches longer than or equal to a given length *L*, between a query and a panel. In the first algorithm, PBWT-Query, we introduce 'virtual insertion' of the query into the PBWT matrix of the panel, and then scanning up and down for the PBWT match blocks with length greater than *L*. In our second algorithm, L-PBWT-Query, we further speed up PBWT-Query by introducing additional data structures that allow us to avoid iterating through blocks of incomplete matches. The efficiency of PBWT-Query and L-PBWT-Query is demonstrated using the simulated data and the UK Biobank data. Our results show that our proposed algorithms can detect related individuals for a given query efficiently in very large cohorts which enables a fast on-line query search.
Availability and implementation: genome.ucf.edu/pbwt-query
Contact: degui.zhi@uth.tmc.edu or shzhang@cs.ucf.edu
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

The increasing volumes of whole-genome genotype data, partly due to the steady drop in the cost of genotyping (Campbell *et al.*, 2015; Jiang *et al.*, 2016), offer a new opportunity to study genetic relationships and relatedness between individuals. In the public domain, biobank-scale SNP array projects have collected large amounts of genotype data. For example, UK Biobank has released genotype and health-related data of ∼500 k individuals in the UK (Bycroft *et al.*, 2018; Sudlow *et al.*, 2015). However, this sample size is dwarfed by the collections in consumer genetics companies. As of July 2018, 23andMe claims to have over 5 million customers and have collected the genotype data of all customers. AncestryDNA has collected DNAs of more than 10 million individuals. It has been projected that genotype data of more than 100 million individuals will be available through direct-to-consumer companies by 2021 (Khan and Mittelman, 2018).

© The Author(s) 2019. Published by Oxford University Press.

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/4.0/), which permits non-commercial re-use, distribution, and reproduction in any medium, provided the original work is properly cited. For commercial re-use, please contact journals.permissions@oup.com

Genetic relationships among individuals are reflected in Identity by Descent (IBD) segments, shared DNA segments between a pair of individuals that have been inherited from a common ancestor. The length of IBD segments correlates with how recently two individuals share a common ancestor. A pair of individuals that share a common ancestor in a more recent generation may share more IBD segments, of larger length, compared with a pair of individuals that share a common ancestor less recently (Thompson, 2013). Therefore, in genotype or haplotype sequences, IBD segments that define a recent common ancestor correspond to long matches of DNA sequences. It has been projected that about 60% of individuals of European descent have a third-degree cousin or closer relative in a current database of over 1 million (Erlich *et al.*, 2018) that can be found using IBD segments.

Previous research efforts were mostly focused on 'M-vs-M' matches of IBD segments. For a collection (panel) with M individuals or haplotypes, the goal was to identify all IBD segments between any pairs of individuals in the panel. Several methods have been proposed to detect all pairwise IBD segments in a panel (Browning and Browning, 2011, 2013a, b; Gusev *et al.*, 2009; Purcell *et al.*, 2007; Rodriguez *et al.*, 2013, 2015). Most of these works rely on pairwise comparisons and thus with computational complexity $O(NM^2)$, where N is the length of the genome. GERMLINE (Gusev *et al.*, 2009) is a fast 'M-vs-M' method claimed to be linear to the size of the panel [$O(NM)$] for random sequences. However, real genetic sequences typically contain repetitive local haplotypes and thus GERMLINE does not demonstrate linear behavior in real sequences. Durbin (2014) proposed an efficient genotype indexing method for storing and searching haplotype sequences. The proposed method, called Positional Burrows-Wheeler Transform (PBWT), is based on sorting the haplotype sequences based on their reversed prefix order. PBWT enables an efficient haplotype search among the haplotypes within a panel and has been applied for genotype phasing and imputation (Loh *et al.*, 2016a, b).

For genealogical search, finding all matches greater than or equal to a certain length between a haplotype query and panel is desired. This can be thought of as finding all individuals in a panel related to the query that have a common ancestor in recent generations, which reveals more information about the query than set-maximal matches. A trivial example illustrates our point: in a large population containing a single pair of identical twins, the only set-maximal match that exists between any pair of individuals is the match between the twins spanning the length of the twins. All other relationships between any pair of individuals are missed. In the general case, to a lesser degree, examining only the longest set-maximal matches misses the majority of measurable genetic relationships between pairs of individuals in a population. The length of matches correlates with the number of generations that two individuals share a common ancestor. As a result, it can be applied to find multiple relatives of a given query individual up to a given degree of relatedness.

The problem under consideration of this project is to develop an algorithm independent from the number of haplotypes for identifying all long IBD segments between a query and a panel, i.e. '1-vs-M' search, given M haplotypes in the panel. Why is the time complexity independent from the number of haplotypes or individuals important? This is because when the panel contains millions of individuals, naive methods of all pairwise comparison [with computational complexity $O(NM)$] will be too slow for real-time applications. However, no efficient on-line method yet exists to identify all long IBD segments between a query and a panel. GERMLINE (Gusev *et al.*, 2009) does yet not offer a direct algorithm for '1-vs-M' search. Durbin's Algorithm 5 (Durbin, 2014) can find the single longest set-maximal matches between a query haplotype and all haplotypes in the panel with runtime $O(N)$, i.e. independent from the number of haplotypes in the panel. However, in practice, multiple 'good enough' matches are desired for a genealogical search. Of note, one might be tempted to repeatedly apply Durbin's set-maximal matches algorithm and exclude the detected match at each run. This solution would not be practical, since the indices for the panel need to be re-computed or updated after each detected match. To our knowledge, there are no algorithms to find all L-long matches, matches longer than or equal to a given length L between a query and a panel of haplotypes, independent from M.

In this work, we present a set of efficient algorithms that can identify all L-long matches of a given query in an arbitrarily large panel. We address the main limitation of Durbin's original PBWT algorithm that cannot find all L-long matches for a new query efficiently. First, we introduce an efficient approach for finding long matches to a query using PBWT, named PBWT-Query. The key idea of PBWT-Query is to 'virtually' insert the query haplotype into the PBWT matrix of the panel, and then scan up and down in the PBWT index for any long match. However, this algorithm may not be efficient when matches are numerous and the cost of scanning is non-negligible. To address this issue, we developed a second algorithm, L-PBWT-Query, that introduces additional pre-computed array data structures, named Linked Equal/Alternating Positions (LEAP) arrays, to help skip unnecessary repetitive up-and-down scanning of PBWT-Query at each site. Moreover, we developed a memory-mapped implementation of the L-PBWT-Query that can alleviate the main memory burden added by the LEAP arrays and the PBWT panel. The time complexity of both PBWT-Query and L-PBWT-Query is independent from the number of haplotypes in the panel.

In the next section, we describe the algorithm in detail, followed by simulation results that show the efficiency of PBWT-Query and L-PBWT-Query. Finally, the application on real data is demonstrated by using the one million haplotypes from the UK Biobank data to search for related individuals extracted from the panel.

2 Materials and methods

2.1 Overview and notation

PBWT (Durbin, 2014) facilitates an efficient approach to find all L-long matches, matches longer than or equal to a given length L, among the haplotypes in a panel. It also provides a fast approach to find the longest matches for a haplotype query in a panel. However, finding L-long matches to a query is of greater interest. Suppose we have a haplotype $s = s[0]s[1]\ldots s[N-1]$. Then, $s[i, j)$ denotes the subsequence $s[i]s[i+1]\ldots s[j-2]s[j-1]$, and $|s|$ denotes the length of s, i.e. $|s| = N$. When we compare any haplotype sequences, we always assume that they share the same sites, number of sites, and ordering of sites. To follow the notations of Durbin (2014), we define the haplotype matrix of X as the matrix whose ith row is x_i.

We can formulate our problem as follows: Given a query haplotype z and a database of haplotypes $X = x_1, x_2, \ldots, x_M$, where $|z| = |x_i| = N$ for any $x_i \in X$, we would like to find all L-long matches that are greater than or equal to length L between z and all $x_i \in X$. We say that there is an L-long match between z and x_i from sites e to k if $k - e \geq L$, $e = 0$ or $z[e-1] \neq x_i[e-1]$ and $k = N$ or $z[k] \neq x_i[k]$, i.e. that match from e to k between z and x_i cannot be extended in either direction. To find L-long matches to a query, we first find the position of the query in the PBWT panel at each site. All of the possible L-long matches ending at a given site k will occur in contiguous blocks of haplotypes adjacent to the 'would be'

position of the query in the panel sorted by the reverse of the prefix ending at site k. Hence, we can scan the neighboring sequences in the PBWT panel to find L-long matches. The time complexity of PBWT-Query is $O(N + c(R - L + 1))$, where R is the average length of the matches, and c is the total number of matches.

To speed up the search, we use LEAP arrays to reduce the search space of possible matches that terminate at each site k by skipping over haplotypes whose matches can still be extended to further sites. The LEAP arrays consist of arrays I^k_{ps}, I^k_{pd}, I^k_{ns}, I^k_{nd}, D^k_{ps}, D^k_{pd}, D^k_{ns} and D^k_{nd} for each site k, which we formally define in the Notation section. These arrays are generated from the PBWT matrix and independent from the query haplotype. We called this new approach L-PBWT-Query, whose worst-case time complexity is $O(N + c)$, where c is the number of L-long matches. The LEAP arrays can be pre-computed in $O(NM)$ time, and require $O(NM)$ space in main memory or hard disk. Therefore, the greatest utility of our approach is the use-case where the user has a large amount of hard disk or main memory space and a constant (or infrequently changing) database haplotype panel and wants to perform many queries on the same panel. In this case, the database population needs only be pre-computed once to achieve subsequently fast. Following, we introduce PBWT data structures and LEAP arrays.

2.1.1 PBWT matrix and positional prefix array

We define the haplotypes in X sorted by the kth reversed prefix (i.e. by the reverse of sites 0 to $k - 1$) as y^k, and the positional prefix array a_k containing a permutation of the indices 0 to $M - 1$ such that $y^k_i = X_{a_k[i]}$. For every y^k, we refer to the contiguous block of haplotypes $y^k_i, y^k_{i+1} \ldots, y^k_{j-1}, y^k_j$ as y^k_i through y^k_j; similarly, $y^k_i[k]$ through $y^k_j[k]$ refers to $y^k_i[k], y^k_{i+1}[k] \ldots, y^k_{j-1}[k], y^k_j[k]$, etc. For example, if $X = \{1001, 1111, 0100\}$, then, $y^3 = \{1001, 0100, 1111\}$ and $a_3 = \{0, 2, 1\}$. Then, we refer to the PBWT matrix as the matrix whose ith column corresponds to the ith column of y^i.

2.1.2 Divergence array

We define the divergence array d_k such that $d_k[i]$ is the starting position of the longest match between y^k_i and y^k_{i-1} ending at k (Durbin, 2014), i.e. d_k gives the starts of all the longest matches ending at a given k for any two adjacent haplotypes in y^k. The longest match between some y^k_i and y^k_j ($i < j$) ending at k begins at $\max_{i<m\leq j} d_k[m]$.

2.1.3 LEAP arrays

Here, we formally define the LEAP arrays: I_{ps}, I_{pd}, I_{ns}, I_{nd}, D_{ps}, D_{pd}, D_{ns} and D_{nd}. For each $y^k[k]$, i.e. each column of the PBWT matrix, we maintain four indices: I^k_{ps}, I^k_{pd}, I^k_{ns} and I^k_{nd}. I_{ps} and I_{ns} allow us to jump around between equal values within each column of the PBWT matrix. Similarly, I_{pd} and I_{nd} allow us to jump around between differing values within each column of the PBWT matrix. The subscripts *ps*, *pd*, *ns* and *nd* are shorthand for previous-same, previous-different, next-same and next-different, respectively, which will make sense below. If j is the largest index such that $y^k_i[k] = y^k_j[k]$ and $j < i$, $I^k_{\mathrm{ps}}[i] = j$; that is, we can use I^k_{ps} to jump in y^k from a given haplotype to the closest preceding haplotype that has the same value at k. If j is the largest index such that $y^k_i[k] \neq y^k_j[k]$ and $j < i$, $I^k_{\mathrm{pd}}[i] = j$; that is, we can use I^k_{pd} to jump in y^k from a given haplotype to the closest preceding haplotype that has a different value at k. If j is the smallest index such that $y^k_i[k] = y^k_j[k]$ and $j > i$, $I^k_{\mathrm{ns}}[i] = j$; that is, we can use I^k_{ns} to jump in y^k from a given haplotype to the closest proceeding haplotype that has the same value at k. Finally, if j is the smallest index such that $y^k_i[k] \neq y^k_j[k]$ and $j > i$, $I^k_{\mathrm{nd}}[i] = j$; i.e. we can use I^k_{nd} to jump in y^k from a given haplotype to the closest proceeding haplotype that has the different value at k. Although these indices facilitate our ability to skip unnecessary haplotypes during our search, when we jump between haplotypes, we need to know if the block of potential L-long matches terminated in the skipped haplotypes. If any divergence value for the skipped haplotypes is larger than $L - k$, we know that our query cannot have an L-long match ending at k to the haplotype that we jumped to. Therefore, we maintain four additional arrays: D_{ps}, D_{pd}, D_{ns} and D_{nd}, which are natural counterparts to I_{ps}, I_{pd}, I_{ns} and I_{nd}. Each of these stores the largest divergence value between the haplotypes that we skipped, depending on the index that we used to jump. For example, if we use $I^k_{\mathrm{ps}}[i] = j$ to jump from $y^k_i[k]$ to $y^k_j[k]$, $D^k_{\mathrm{ps}}[i]$ will give us $\max_{j<r<i} d_k[r]$; similarly, if we use $I^k_{\mathrm{ns}}[i] = j$ to jump from $y^k_i[k]$ to $y^k_j[k]$, $D^k_{\mathrm{ns}}[i]$ will give us $\max_{i<r<j} d_k[r]$. D_{ps}, D_{pd}, D_{ns} and D_{nd} are undefined when fewer than two haplotypes are skipped using I_{ps}, I_{pd}, I_{ns} and I_{nd}, respectively; in those cases, we can use the divergence arrays directly.

2.2 PBWT-Query: finding all L-long matches from a new sequence z to X in $O(N + c(R - L + 1))$ time

2.2.1 'Virtual insertion' of the query haplotype

Figure 1 shows a simple example of virtually inserting a query haplotype to the PBWT panel. The position of the new haplotype at each site will be adjacent to the longest match at the site k. Durbin (2014) proposed an efficient algorithm to find all set-maximal

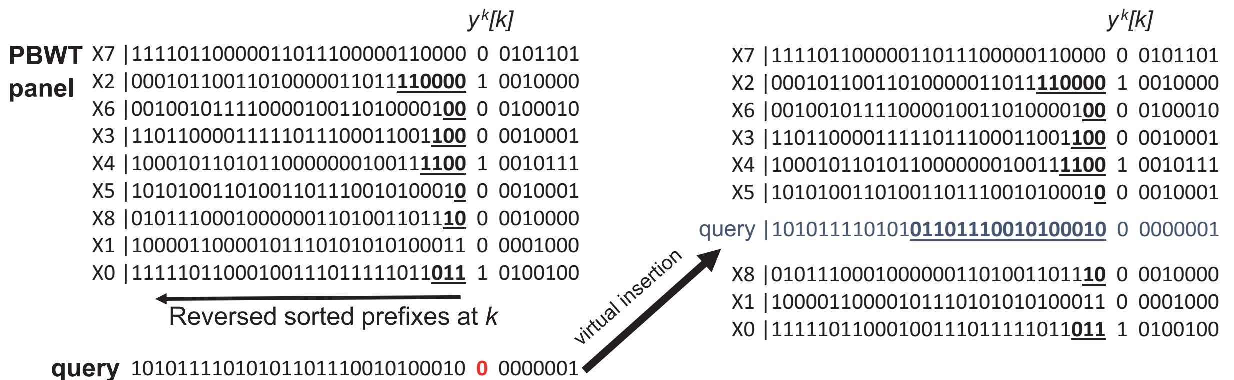

Fig. 1. An example of virtually inserting a new query haplotype to a PBWT panel at site k. The haplotype sequences of the panel are sorted based on the reversed prefix order in the PBWT panel at site k

matches between a query z and a panel X, UpdateZMatches [Durbin's Algorithm 5 (Durbin, 2014)]. A set-maximal match (Durbin, 2014), between z and some $x_i \in X$ is defined as an L-long match ($L \geq 1$) between z and x_i from e to k such that there does not exist a match between z and any other $x_j \in X\,(i \neq j)$ from e' to k' where $e' < e$ or $k' > k$. Simply, the match between z and x_i is set-maximal if it cannot be extended in either direction, and z does not have a larger match to any other haplotype in X whose indices enclose the indices of its match to x_i. To find all set-maximal matches at k, UpdateZMatches computes three values: f_k, g_k and e_k such that z has a set-maximal match to $y^k_{f_k}$ through $y^k_{g_k-1}$ from e_k to k if $y^k_{f_k}[k]$ through $y^k_{g_k-1}[k] \neq z[k]$. It computes f_k, g_k and e_k based on f_{k-1}, g_{k-1} and e_{k-1}. A detailed routine for computing f_k, g_k and e_k can be found in the Supplementary Algorithm S1. To follow the notations of Durbin (2014), we define w such that $w(k,i,0) = u_k[i]$ and $w(k,i,1) = c_k + v_k[i]$, where $u_k[i]$ is the number of zero values at site k in y^k_0 through y^k_{i-1}, $v_k[i]$ is the number of one values at site k in y^k_0 through y^k_{i-1}, and c_k is the total number of zeros at site k.

It is the case that if z were in y^k, it would occur in sorted order immediately preceding either $y^k_{f_k}$ or $y^k_{g_k}$. This is because $z[e_k, k)$ matches $y^k_{f_k}[e_k, k)$ through $y^k_{g_k-1}[e_k, k)$, but $y^k_{f_k}[e_k - 1] \neq y^k_{g_k-1}\,[e_k - 1]$ (or $e_k = 0$). In both cases, if $z[e_k - 1] = 0$, z can go in sorted order immediately preceding $y^k_{f_k}$. Otherwise, it can immediately proceed to $y^k_{g_k-1}$. We refer to this index of y^k which would immediately proceed to z as h_k, and we can use this to find all L-long matches between z and X ending at k.

y^k_{i+1} through y^k_j all have a match to y^k_i of at least length $k - d_k[m]$, and y^k_i through y^k_{j-1} all have a match to y^k_j of at least length $k - d_k[m]$. This implies that if we pick some haplotype y^k_i in X, we know that all matches to y^k_i ending at k of length greater than or equal to L occur in some block y^k_t through y^k_b in y^k such that $t < b$, $0 \leq t \leq i+1$, $i-1 \leq b < M$ and $k - \max_{t<m\leq b} d_k[m] \geq L$. It follows that all matches to X ending at k, whether from an external query haplotype or internal haplotype, occur in a contiguous block at y^k.

2.2.2 Reporting all L-long matches

We know that all matches to z ending at k of length greater than or equal to L occur in a contiguous block $y^k_{t_k}$ through $y^k_{b_k}$ in y^k such that $t_k < b_k$, $0 \leq t_k \leq h_k$, $h_k - 1 \leq b_k < M$ and $k - \max_{t_k<m\leq b_k} d_k[m] \geq L$. What is left, then, is to find the smallest index t_k and largest index b_k that enclose the block of haplotypes with potential L-long matches. We say *potential* because each $y^k_{t_k \leq i \leq b_k}$ is only a *match* if $y^k_i[k] = z[k]$, but not an *L-long match*, as the end of the match can be extended, i.e. $y^k_i[e_k, k+2)$ matches $z[e_k, k+2)$. We define d^k_z to be the smallest value such that $z[d^k_z, k)$ matches $y^k_{h_k-1}[d^k_z, k)$ and d^k_b to be the smallest value such that $y^k_{h_k}[d^k_b, k)$ matches $z[d^k_b, k)$. We can use d^k_z and d_k to scan up y^k beginning at $y^k_{h_k-1}$ until we find $y^k_{t_k}$ (except in the case where $t_k = h_k$), printing L-long matches along the way. Similarly, we can use d^k_b and d_k to scan down y^k beginning at $y^k_{h_k}$ until we find $y^k_{b_k}$ (except in the case where $b_k = h_k - 1$), printing L-long matches along the way. Specifically, we can keep scanning in either direction so long as $k - d_k[i] \geq L$ for the particular y^k_i we are iterating through [similar to Durbin's Algorithm 3 (Durbin, 2014)], and we can print an L-long match between z and y^k_i so long as $z[k] \neq y^k_i[k]$. Although scanning up, if $t_k \leq h_k - 2$, the match between z and a particular y^k_i would begin at $\max(d_z, \max_{i-1\leq m\leq h_k-1} d_k[m])$. Similarly, while scanning down, if $b_k > h_k$, the match between z and a particular y^k_i would begin at $\max(d_b, \max_{h+1\leq m\leq i} d_k[m])$. A detailed procedure of this approach is presented in the Supplementary Algorithm S2.

Figure 2 illustrates searching for L-long matches for $L = 3$. For clarity, we have included example y^k's for $k = 1, 2, 3$, where we have

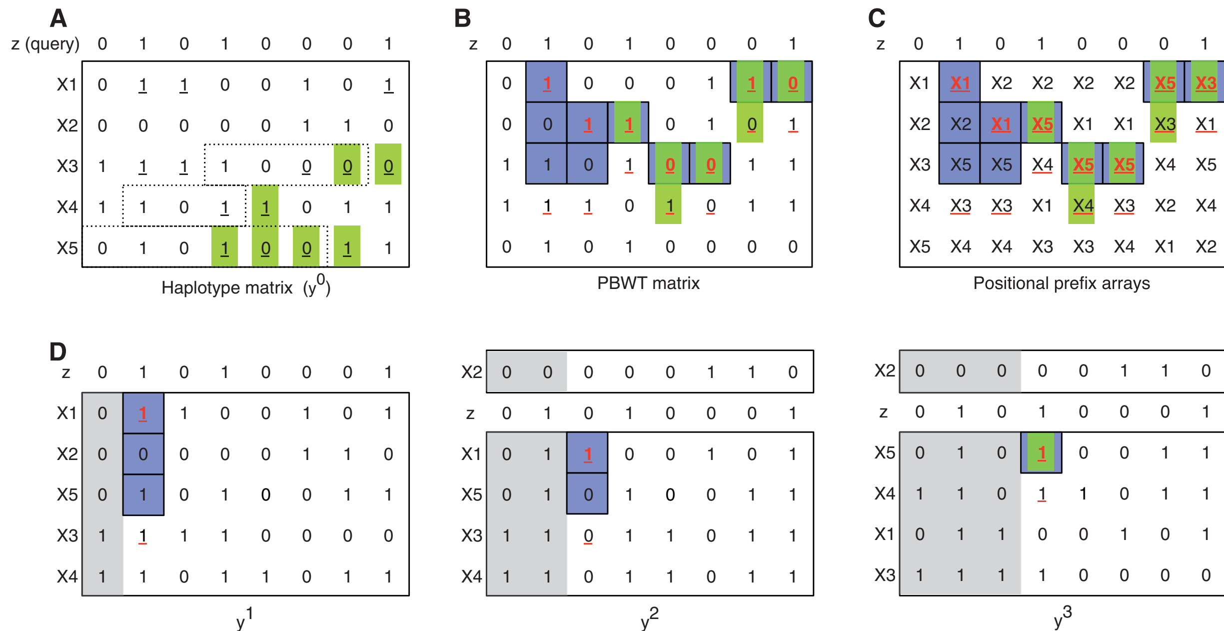

Fig. 2. An example of searching for L-long matches of length ≥ 3 in a panel with five haplotypes comprising eight sites. **(A)** The haplotype matrix (y^0) and the query z. **(B)** The PBWT matrix. ith column in the PBWT matrix corresponds to the ith column of y^i. **(C)** The positional prefix arrays. Each row depicts the positional prefix array a_k. Each positional prefix array a_k contains the permutation of indices 0 to $N-1$ such that $y^k_i = X_{a_k[i]}$. **(D)** y^k's for $k = 1, 2, 3$. The haplotypes in X sorted by the kth reversed prefix (i.e. by the reverse of sites 0 to $k-1$) are referred as y^k. The prefixes used to sort are shaded in gray in y^1, y^2 and y^3. The blocks of potential set-maximal matches ending at k are labeled in purple, the index of the haplotype that immediately proceeds to z is highlighted as red, and the intervals of potential L-long matches ending at each site are highlighted in lime green. f and g which enclose the block of potential set-maximal matches are underlined

Algorithm 1 Scan Up For L-Long Matches

```
function ScanUp(k, h, d_z)
  if k + 1 − d_z ≥ L and match ends at k + 1 then
    report match to y_{h−1}^{k+1} from d_z to k + 1
  d_max ← d_z, t ← h − 2
  if k + 1 − d_max ≥ L then
    if k ≤ N and k + 1 − d_{k+1}[t + 1] ≥ L then
      //Use I_ps^k or I_pd^k to find the first y_i^k (i ≤ h − 2) such
      //that y_i^k[k] ≠ z[k], if such exists
      d_max ← max(d_max, d_{k+1}[t + 1])
      if y_t^{k+1}[k + 1] = z[k + 1] and I_pd^{k+1}[t] ≥ 0 then
        d_max ← max(d_max, d_{k+1}[t], D_pd^{k+1}[t])
        t ← I_pd^{k+1}[t]
      else if y_t^{k+1}[k + 1] = z[k + 1] then
        t ← −1
      //If necessary, we can use I_ps^k to keep searching only
      //through haplotypes that differ from z at site k
      while t ≥ 0 and k + 1 − d_max ≥ L do
        report match to y_t^{k+1} from d_max to k + 1
        if I_ps^{k+1}[t] ≥ 0 then
          d_max ← max(d_max, d_{k+1}[t], D_ps^{k+1}[t])
        t ← I_ps^{k+1}[t]
    else
      while k + 1 − d_{k+1}[t + 1] ≥ L do
        if d_{k+1}[t + 1] > d_max then d_max ← d_{k+1}[t + 1]
        report match to y_t^{k+1} from d_max to k + 1
        t ← t − 1
```

highlighted in gray the reverse prefixes that are used to sort the rows of the haplotype matrix, and have included the sorted order of z in these panels. The key idea of our approach is, scanning through each site k from 0 to N, to find the would-be position of the query z in y^k. Then, we can scan up and down site k in y^k to search for L-long matches that end at that site. First, at each site k, we find the block of potential set-maximal matches ending at k, labeled in purple. If z were in y^k, it would occur at the beginning or end of this block. In this example, z always happens to be at the beginning of the purple blocks, but it can also occur at the end. As we scan through this block, we output an L-long match ending at each green position if, at that position, there is a mismatch between that individual and the query.

The time complexity of computing f_k, g and e_k is $O(N)$ across all k (Durbin, 2014). For a given site, we may scan through all M haplotypes in search of t_k and b_k. If there is a block of matches that does not terminate at each site, we will still need to scan the entire block. As we do this for N sites, the time complexity of PBWT-Query is $O(N + c(R - L + 1))$, where R is the average length of the matches and c the total number of matches. In the next section, we introduce L-PBWT-Query which employs additional data structures to achieve a time complexity independent from the number of haplotypes and the lengths of the matches.

2.3 L-PBWT-Query: finding all L-long matches from a new sequence z to X in $O(N + c)$ time

Although searching for t_k and b_k, the smallest and largest indices that contain potential L-long matches, every y_i^k scanned through

Algorithm 2 L-PBWT-Query

```
for k ← 0 to N − 1 do
  //z matches y_{f_{k+1}}^{k+1} through y_{g_{k+1}−1}^{k+1} from e_{k+1} until
  //at least k + 1
  f_{k+1}, g_{k+1}, e_{k+1} ← UpdateFandG(k)
  //z can be inserted into y^{k+1} in sorted order either
  //before y_{f_{k+1}}^{k+1} or before y_{g_{k+1}}^{k+1}
  if z[e_{k+1} − 1] = 1 then
    h_{k+1} ← g_{k+1}
    d_b^{k+1} ← max(e_{k+1}, d_{k+1}[h_{k+1}])
    d_z^{k+1} ← e_{k+1}
  else
    h_{k+1} ← f_{k+1}
    d_z^{k+1} ← max(e_{k+1}, d_{k+1}[h_{k+1}])
    d_b^{k+1} ← e_{k+1}
  if h_{k+1} > 0 then
    ScanUp(k, h_{k+1}, d_z^{k+1})
  if h < M then
    ScanDown(k, h_{k+1}, d_b^{k+1})
```

where $z[k] = y_i^k[k]$ (i.e. the match can be extended further) is unnecessary work. Therefore, if we can restrict our search at each site only to those y_i^k where $z[k] \neq y_i^k[k]$, we can improve the time complexity of searching for t_k and b_k across all sites to $O(N + c)$, where c is the number of L-long matches. We can achieve this by maintaining a data structure that allows us to efficiently jump between $y_i^k[k]$ of the same k, for $i = 0$ to $M - 1$ and $k = 1$ to $N - 1$. Again, the key idea is that we want to jump around between only those $y_i^k[k]$ where $z[k] \neq y_i^k[k]$. For bi-allelic data, if $z[k] = 1$, we primarily want to move around $y_i^k[k]$ that equal to 0, and similarly, if $z[k] = 0$, we primarily want to move around between $y_i^k[k]$ that equal to 1, to find matches that end at $k + 1$.

2.3.1 Speeding up PBWT-Query using LEAP arrays

Using our data structure, we can modify the procedures to scan up and down in search of t_k and b_k once we have found h_k. Namely, when we scan up, we can use I_{ps}^k or I_{pd}^k to find the first y_i^k ($i \leq h - 2$) such that $y_i^k[k] \neq z[k]$, if such exists. Then, if necessary, we can use I_{ps}^k to keep searching only through haplotypes that differ from z at site k. Algorithm 1 demonstrates the updated routines which efficiently scan up a given y^k to find t_k and b_k using our data structure. Similarly, when we scan down, we can use I_{ns}^k or I_{nd}^k to find the first y_i^k ($i \geq h + 1$) such that $y_i^k[k] \neq z[k]$, if such exists. Then, if necessary, we can use I_{ns}^k to keep searching only through haplotypes that differ from z at k. A detailed routine of the algorithm for scanning down is included in the Supplementary Algorithm S3. Then, Algorithm 2 demonstrates our updated L-PBWT-Query that makes use of our improved scanning routines in Algorithm 1 and Supplementary Algorithm S3.

Figure 3 demonstrates y^{30} for an example genotype panel, along with the indices used to facilitate efficient query search. In the panel, t and b refer to the indices that define the block of haplotypes y_t^{30} through y_b^{30} where all L-long matches for $L = 4$ ending at site 30 may occur. Within that block of haplotypes, we are interested in those y_i^{30} where $y_i^{30}[30] \neq z[30]$. Since we would have to scan up from y_{h-1}^{30} to find t and scan down from y_h^{30} to find b, without the indices, we could end up scanning through the entire column, a worst-

i	$a_{30}[i]$	$y^{30}[30]$				
0	X4	100010001110000000100000000000	0		100010001	
1	X5	000000000010000000000001**0000000**	1		101010000	
2	X7	10001000001000000 1**0000100000000**	0		010101000	
3	X3	1000100001100000000100000000 1**00**	0		100010001	
4	X9	**100010000110000000100000000100**	0		100010001	
5	X8	00000000000010111001000000101**0**	0		010101000 ◂	t
6	X0	000100000000111100001001**001010**	0	$I^{30}_{pd}[7]$	010100000	
		0100010100001011000000001 1**01010**	1	$I^{30}_{ps}[7]$	101011000 ◂	z
7	X1	0100010100001011000000000 1**01010**	0		010101010 ◂	h
8	X6	00100000000110110000100101 1**1010**	1	$I^{30}_{nd}[7]$	010100000 ◂	b
9	X2	000001010000101100000000101011	1		010101100	

i	I^{30}_{ps}	I^{30}_{pd}	I^{30}_{ns}	I^{30}_{nd}	D^{30}_{ps}	D^{30}_{pd}	D^{30}_{ns}	D^{30}_{nd}
0			2	1			23	
1			8	2			29	
2	0	1	3	8	23			29
3	2	1	4	8		18		29
4	3	1	5	8		28		29
5	4	1	6	8		28		25
6	5	1	7	8		29		25
7	6	1		8		29		
8	1	7	9					
9	8	7				26		

Fig. 3. An example of a haplotype panel with its corresponding indices to facilitate the search for L-long matches for $L = 4$ ending at site 30, exclusive. The indices I^{30}_{ps}, I^{30}_{pd}, I^{30}_{ns}, I^{30}_{nd} allow jumping from any $y^{30}_i[30]$ to the nearest $y^{30}_j[30]$ preceding or proceeding $y^{30}_i[30]$ such that $y^{30}_i[30] = y^{30}_j[30]$ or $y^{30}_i[30] \neq y^{30}_j[30]$. D^{30}_{ps}, D^{30}_{pd}, D^{30}_{ns} and D^{30}_{nd} give us the largest divergence values from haplotypes skipped using I^{30}_{ps}, I^{30}_{pd}, I^{30}_{ns} and I^{30}_{nd}, respectively. The indices for the seventh sequence in y^{30} are highlighted in the table. t and b refer to the top and bottom of the block of potential L-long matches in y^{30}, and h refers to the sequence that would immediately proceed to the query if it were in y^{30}

case $O(M)$ operation; across all sites, we could end up performing $O(NM)$ operations. However, with the indices, we only ever traverse through those sequences where the match ends at 30; so, at site 30, our scanning up and down is improved to worst-case O(number of matches ending at 30), and therefore, across all sites, our runtime is improved to worst-case $O(N + c)$, where c is the number of L-long matches.

As before, the time complexity of computing f_k, g_k and e_k is $O(N)$ across all k (Durbin, 2014). Now, our worst-case time complexity for scanning for t_k and b_k across all sites is $O(N + c)$, where c is the number of L-long matches. So, our total worst-case time complexity for L-PBWT-Query is $O(N + c)$, which is independent from the number of haplotypes M. Although our querying approach has a runtime linear in N and the number of L-long matches, our approach assumes that the PBWT, I matrices and D matrices for X are pre-computed, and stored in the hard disk. These can be computed in $O(NM)$ time, and they occupy $O(NM)$ space in the hard disk. Specifically, if X occupies NM memory, the additional I and D matrices occupy roughly $8NM$ memory.

2.3.2 Memory-efficient implementation of L-PBWT-Query

We implemented two versions of L-PBWT-Query: L-PBWT-Query (Memory-Mapped), where all of the pre-computed data structures are accessed using memory-mapped files using Boost libraries, and L-PBWT-Query (Memory-Extensive), where all of the pre-computed data structures are loaded into main memory. To clarify, L-PBWT-Query (Memory-Mapped) does not load all of the panel and data structures in main memory at once. Instead, parts of the panel and data structures are loaded from the appropriate files into memory in a 'lazy loading' fashion, i.e. as they are needed. Therefore, using memory-mapped files reduces I/O operations. This mechanism provides a relatively fast alternative to access the panel and data structures, especially if a panel has been recently queried by queries relatively similar to subsequent queries, which we refer to as 'warming up' a panel. When a panel is warmed up, relevant portions of the panel and data structures (a relatively small subset of the overall panel and data structures) have already been loaded into main memory and mapped to virtual memory, for faster subsequent access.

3 Results

3.1 Benchmarking using simulated data

We simulated a large haplotype panel using the Markovian Coalescent Simulator (MaCS) Chen *et al.* (2009) with the command macs 500001 2000000 -t .001 -r .001 -h 1e2 and extracted subsets of the panel for our benchmarking. For the purpose of benchmarking, we implemented PBWT-Query (Memory-Extensive) and Exhaustive Search. Exhaustive Search scans across the entire length of sites for each pair of the query and a haplotype in the panel, which has $O(NM)$ time complexity. For all of our benchmarks, we used the following protocol: When running L-PBWT-Query (Memory-Mapped), for a given panel, we always warmed up a panel with three runs on the particular query, then averaged the runtime of three additional runs. When running PBWT-Query (Memory-

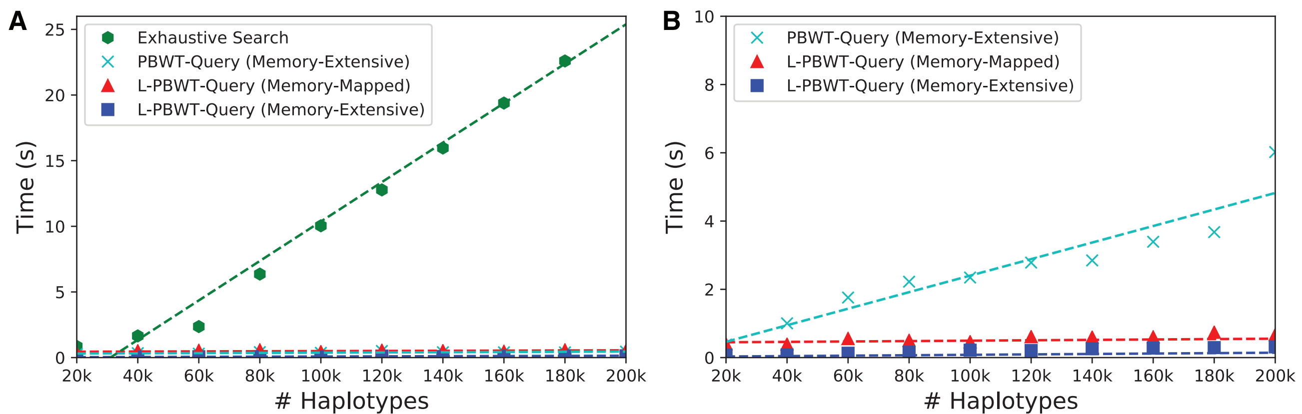

Fig. 4. Running time (in seconds) of searching for a query in panels containing 10 000 sites with increasing number of haplotypes **(A)** while keeping the number of matches constant and **(B)** with increasing number of matches

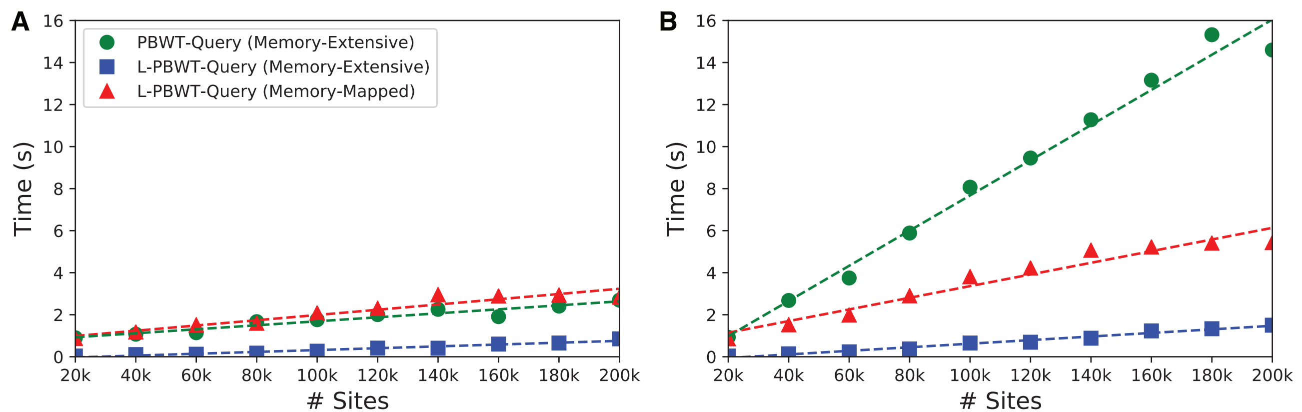

Fig. 5. Running time (in seconds) of searching for a query in panels containing 20 000 haplotypes with increasing number of sites: **(A)** while keeping the number of matches relatively constant (~70 000 matches) and **(B)** with increasing number of matches linearly with the number of sites

Extensive), L-PBWT-Query (Memory-Extensive) and Exhaustive Search, we averaged the runtimes of three runs.

First, we want to verify that the runtime of PBWT-Query and L-PBWT-Query, in practice, is truly independent from the number of haplotypes. To that end, we developed a benchmark which we refer to as the increasing haplotypes benchmark. Figure 4 and Supplementary Table S1 show the results of our increasing haplotypes benchmark, respectively, which support our assertion that the runtime of our approach is independent from the number of haplotypes. For this benchmark, we tested panels where we successively increased the number of haplotypes in the panel by a constant amount (from 20 000 to 200 000 haplotypes in steps of 20 000). In the first experiment, we kept the number of sites (10 000) and the number of matches (25 359) constant for the added haplotypes (Fig. 4A and Supplementary Table S1). The minimum length of match L was set to 1000. We can see that, indeed, for the Exhaustive Search, run time increases roughly linearly with the number of haplotypes (M). The runtime of our memory-mapped implementation stays roughly constant, even as we increase the number of haplotypes ten times. When compared with L-PBWT-Query (Memory-Extensive), there appears to be some constant overhead associated with fetching data from memory-mapped files, instead of cache or main memory. PBWT-Query runtime is similar to L-PBWT-Query as the number of matches remains constant. In the second experiment, we increased the number of haplotypes and the number of matches increases linearly (Fig. 4B and Supplementary Table S1). Both L-PBWT-Query implementations show a better runtime compared with PBWT-Query with increasing number of haplotypes and increasing number of matches.

Additionally, we want to concretely investigate the additive effect of increasing sites and increasing matches on the runtime of PBWT-Query and L-PBWT-Query. To that end, we developed a benchmark which we refer to as the increasing sites benchmark. Figure 5 is a plot of our increasing sites benchmarks, and Supplementary Table S2 gives the runtimes of those benchmarks. We performed two benchmarks: one where we successively increase the number of sites (from 20 000 to 200 000) while keeping the number of matches constant (Fig. 5A), and another where we successively increase the number of sites (from 20 000 to 200 000) while the number of matches increases linearly with the number of sites (Fig. 5B). From both of these figures, we can see that the runtime of our approach increases as expected, linearly with the number of sites and matches. As before, there is some constant overhead associated with fetching from memory-mapped files instead of main memory. All benchmarks were run on a 2.1 GHz server with 500 GB of RAM. The maximum resident sizes for the benchmarks are included in the Supplementary Table S3.

3.2 Applying PBWT-Query and L-PBWT-Query on the UK Biobank data

PBWT-Query and L-PBWT-Query were tested on the UK Biobank data (487 409 participants and 974 818 haplotypes for autosomal

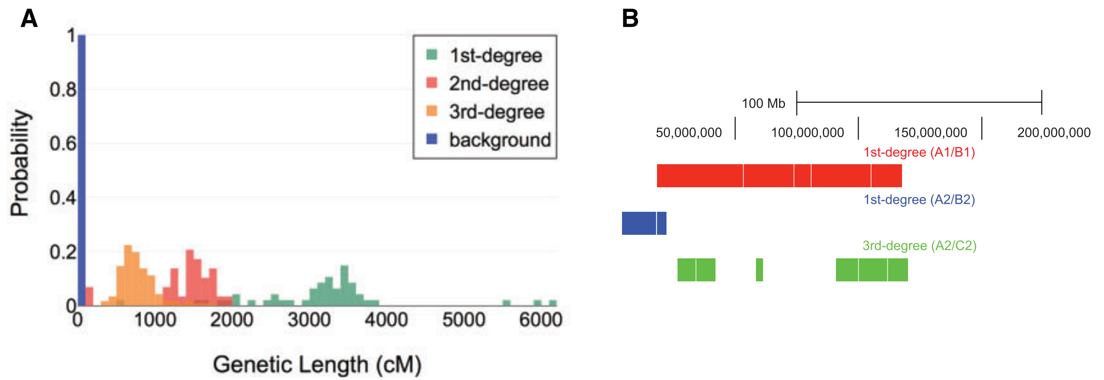

Fig. 6. (**A**) Probability distributions of the sum of *L*-long matches (in cM) between the query and related individuals (first-, second- or third-degree), and random individuals in the UK Biobank data. Relatedness is computed using KING. **(B)** An example of detected identical segments in Chromosome 1 (in bps) for an individual with at least two relatives (first- and third-degree relatives) in the UK Biobank data

chromosomes) (Bycroft *et al.*, 2018) to demonstrate their utility on real data. The UK Biobank dataset includes pairwise kinship coefficients between individuals computed using the KING toolset Manichaikul *et al.* (2010). According to the KING tutorial, the kinship coefficient ranges [0.177, 0.354], [0.0884, 0.177] and [0.0442, 0.0884] denote first-, second- and third-degree relationships, respectively. Here, a first-degree relationship refers to a parent–offspring or full-sibling relationship, a second-degree relationship includes half-siblings, avuncular pairs and grandparent–grandchild pairs, and a third-degree relationship includes the first-cousins Manichaikul *et al.* (2010).

The aim is to investigate whether potential genetic relationships from the UK Biobank data can be inferred by searching for exact matches using PBWT-Query or L-PBWT-Query. Because true IBDs can be disrupted by mismatches, the sum of the lengths of L-long matches between a pair of individuals was tested as a potential signal to differentiate first-, second- and third-degree relationships from each other, and from a background population. Two hundred individuals (400 haplotypes) which had genetic relationships within the UK Biobank data were chosen as queries. As a representative sample of the background population, one thousand individuals were randomly chosen from the UK Biobank excluding the 200 query individuals. The two hundred individuals were queried against the entire UK Biobank data for exact matches of length $\geq$ 700 SNPs in Chromosomes 1 through 22. This match length cutoff was chosen by following the precedent of 23andMe, whose simulations show that first, second and third cousin relationships can be reliably detected by haplotype matches of length $\geq$ 700 SNPs and 7 cM across all chromosomes. If we consider the detection power as the percentage of related pairs with any matches of length $\geq$ 700 SNPs, there is a 100% detection power in detecting potential first- and second-degree relationships by searching only in Chromosome 1. The detection power for first, second and third-degree relationships using all autosomal chromosomes is 100%. The queries were run using PBWT-Query and L-PBWT-Query (Memory-Extensive) on a 2.5 GHz server with 15 TB of SSD and 6 TB of RAM. The average time for running L-PBWT-Query a single query (one haplotype) on Chromosomes 1 through 22 is 6 s and for PBTW-Query 20 s, discounting the time to load the L-PBWT-Query data structures into memory. The maximum resident set size for L-PBWT-Query was 4.7 TB, and 2.4 TB for PBWT-Query.

There is a promising separation between the three degrees of relatedness and the background population. Figure 6A shows that the sum of the lengths of L-long matches between a pair of individuals is a capable signal to differentiate first-, second- and third-degree relationships from random pairs picked from the background population, and to filter potential first- second- and third-degree relationships for further processing. Figure 6B shows an example of detected identical segments in Chromosome 1 between an individual (A) and two of their relatives (B and C) in the UK Biobank data. The two haplotypes for each individual are distinguished by 1 and 2 (e.g. A has haplotypes A1 and A2). The total length of the segments shared with the first-degree relative is significantly larger than those shared with the third-degree relative. We computed the AUC values to differentiate three degrees of relationships using the sum of the lengths of the shared segments. The AUC value between first- and third-degree relationships is 0.98, and AUC value between first- and second-degree relationships is 0.96. There are a few outliers in the second-degree relationships; however, the AUC value between the second- and third-degree relationships is 90%. Thus, identical segments interrupted due to genotyping or phasing errors can be used to infer the related individuals, and to estimate the degree of relatedness between closely related individuals, without careful post-processing of the interrupted matches.

4 Discussion

In this work, we proposed two efficient algorithms, PBWT-Query and L-PBWT-Query, for finding all L-long matches between a query haplotype and a panel. The time complexity of L-PBWT-Query does not rely on the number of haplotypes in the panel, which enables on-line genealogical search in large cohorts. Furthermore, the memory-mapped version of the algorithm, L-PBWT-Query (Memory-Mapped), will facilitate fast search when extensive main memory is not available, in exchange for a slightly increased running time. PBWT-Query shows similar runtime to L-PBWT-Query as the number of matches remains constant with the increasing number of haplotypes. However, the difference in efficiency becomes more obvious with the increasing number of matches.

L-PBWT-Query (Memory-Mapped) does not require loading the entire panel into the main memory in order to facilitate a fast search. The tradeoff, however, is an increased running time due to increased I/O operations. We demonstrated that the running time of the L-PBWT-Query (Memory-Mapped) version is slightly worse than L-PBWT-Query (Memory-Extensive), but L-PBWT-Query (Memory-Mapped) will be more practical if extensive memory resources are not available.

We applied L-PBWT-Query on ~500 k individuals from the UK Biobank data and were able to detect close relatives of query individuals. The running time for all autosomal chromosomes in the UK Biobank data would be only a few seconds using a single CPU. Our results show that very close relatives can be easily found by L-PBWT-Query. We also ran PBWT-Query on the UK Biobank data. Although the memory usage for PBWT-Query was lower than L-PBWT-Query, the run time was moderately worse. To further improve detection, consideration of haplotype phasing quality will be needed. Another limitation of our work is that we have been focusing on developing efficient algorithms for exact matches of a query in the panel. In order to enhance the application on real datasets and detect more distant relatives, the algorithms need to be made tolerant to genotyping errors or mutations that could have occurred in real data, e.g. by random projection (Naseri *et al.*, 2017).

Acknowledgements

We thank Dr Xiaoqian Jiang for sharing computational resources. This research has been conducted using the UK Biobank Resource under Application Number 24247.

Funding

This work was supported by the US National Institutes of Health [R01HG010086].

Conflict of Interest: The authors, through their respective universities, have submitted a provisional patent application based on this technology. This does not restrict the non-commercial use of the methods described in this article.

References

Browning,B.L. and Browning,S.R. (2011) A fast, powerful method for detecting identity by descent. *Am. J. Hum. Genet.*, **88**, 173–182.

Browning,B.L. and Browning,S.R. (2013a) Detecting identity by descent and estimating genotype error rates in sequence data. *Am. J. Hum. Genet.*, **93**, 840–851.

Browning,B.L. and Browning,S.R. (2013b) Improving the accuracy and efficiency of identity-by-descent detection in population data. *Genetics*, **194**, 459–471.

Bycroft,C. *et al.* (2018) The UK Biobank resource with deep phenotyping and genomic data. *Nature*, **562**, 203–209.

Campbell,N.R. *et al.* (2015) Genotyping-in-Thousands by sequencing (GT-seq): a cost effective SNP genotyping method based on custom amplicon sequencing. *Mol. Ecol. Resour.*, **15**, 855–867.

Chen,G.K. *et al.* (2009) Fast and flexible simulation of DNA sequence data. *Genome Res.*, **19**, 136–142.

Durbin,R. (2014) Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). *Bioinformatics*, **30**, 1266–1272.

Erlich,Y. *et al.* (2018) Identity inference of genomic data using long-range familial searches. *Science*, **362**, 690–694.

Gusev,A. *et al.* (2009) Whole population, genome-wide mapping of hidden relatedness. *Genome Res.*, **19**, 318–326.

Jiang,Z. *et al.* (2016) Genome wide sampling sequencing for SNP genotyping: methods, challenges and future development. *Int. J. Biol. Sci.*, **12**, 100–108.

Khan,R. and Mittelman,D. (2018) Consumer genomics will change your life, whether you get tested or not. *Genome Biol.*, **19**, 120.

Loh,P.R. *et al.* (2016a) Fast and accurate long-range phasing in a UK Biobank cohort. *Nat. Genet.*, **48**, 811–816.

Loh,P.R. *et al.* (2016b) Reference-based phasing using the Haplotype Reference Consortium panel. *Nat. Genet.*, **48**, 1443–1448.

Manichaikul,A. *et al.* (2010) Robust relationship inference in genome-wide association studies. *Bioinformatics*, **26**, 2867–2873.

Naseri,A. *et al.* (2017). Ultra-fast identity by descent detection in biobank-scale cohorts using positional Burrows-Wheeler transform. *bioRxiv*, 103325.

Purcell,S. *et al.* (2007) PLINK: a tool set for whole-genome association and population-based linkage analyses. *Am. J. Hum. Genet.*, **81**, 559–575.

Rodriguez,J.M. *et al.* (2013). An accurate method for inferring relatedness in large datasets of unphased genotypes via an embedded likelihood-ratio test. In: *Proceedings of the 17th International Conference on Research in Computational Molecular Biology*, pp. 212–229. Springer, Berlin, Heidelberg.

Rodriguez,J.M. *et al.* (2015) Parente2: a fast and accurate method for detecting identity by descent. *Genome Res.*, **25**, 280–289.

Sudlow,C. *et al.* (2015) UK Biobank: an open access resource for identifying the causes of a wide range of complex diseases of middle and old age. *PLoS Med.*, **12**, e1001779.

Thompson,E.A. (2013) Identity by descent: variation in meiosis, across genomes, and in populations. *Genetics*, **194**, 301–326.