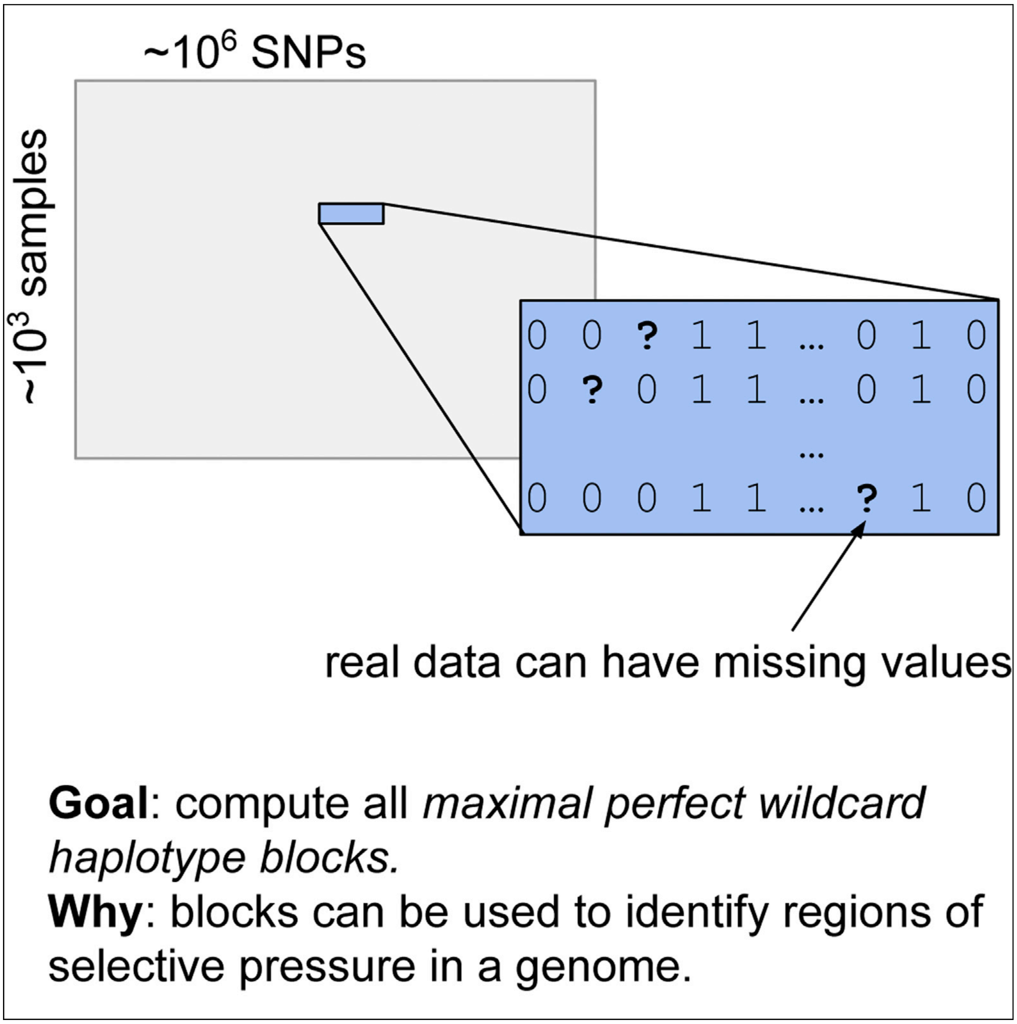
iScience

CellPress
OPEN ACCESS

Article

Maximal Perfect Haplotype Blocks with Wildcards



Goal: compute all *maximal perfect wildcard haplotype blocks.*
Why: blocks can be used to identify regions of selective pressure in a genome.

Lucia Williams,
Brendan Mumey

lucia.williams@montana.edu

HIGHLIGHTS

Defined haplotype blocks to study SNP population data with missing values

Developed a fast software tool to find these blocks

Tested on a human chromosome 22 dataset of 5,008 samples and over one million SNPs

Williams & Mumey, iScience *23*, 101149
June 26, 2020 © 2020 The Author(s).
https://doi.org/10.1016/j.isci.2020.101149

Article

Maximal Perfect Haplotype Blocks with Wildcards

Lucia Williams[1,2,*] and Brendan Mumey[1]

SUMMARY

Recent work provides the first method to measure the relative fitness of genomic variants within a population that scales to large numbers of genomes. A key component of the computation involves finding maximal perfect haplotype blocks from a set of genomic samples for which SNPs (single-nucleotide polymorphisms) have been called. Often, owing to low read coverage and imperfect assemblies, some of the SNP calls can be missing from some of the samples. In this work, we consider the problem of finding maximal perfect haplotype blocks where some missing values may be present. Missing values are treated as wildcards, and the definition of maximal perfect haplotype blocks is extended in a natural way. We provide an output-linear time algorithm to identify all such blocks and demonstrate the algorithm on a large population SNP dataset. Our software is publicly available.

INTRODUCTION

Several recent works have explored the problem of determining maximum perfect haplotype blocks (MPHBs) in large population datasets where each individual in the dataset is genotyped at a set of single nucleotide polymorphism (SNP) sites along their genome. These data can be viewed as a large $\{0, 1\}$ matrix, where the rows represent each individual or sample and the columns are the SNP locations along a chromosome of interest. Typically, a 0 indicates no change from the reference and a 1 represents a change. The study of algorithms to find MPHBs was motivated by their usefulness in determining the fitness of genetic variations for a set of loci along the genome by computing the *selection coefficient* (Gillespie, 2004, Chapter 5.3), which indicates how stable each loci is from an evolutionary standpoint. With an algorithm for computing MPHBs that scales to entire chromosomes and thousands of individual samples, it is possible to investigate the selective pressure at each locus over an entire chromosome. The first such algorithm runs in quadratic time (Cunha et al., 2018); more recent work (Alanko et al., 2020) gives two methods for finding MPHBs that both run in linear time. The first linear time method is based on suffix trees, and the second is based on the positional Burrows-Wheeler Transform (pBWT), both of which are data structures developed for efficient processing of string (alphabet character) data. In practice, the authors found the pBWT-based method to be the faster of the two.

In this work, we extend the definition of maximal perfect haplotype blocks to include the possibility of missing values in the data. We model such missing values as *wildcards*. Sequencing data are often noisy and missing SNP calls are common, so it is natural to consider this variation of the problem. Wildcards have also been considered in other pattern matching contexts; e.g., researchers have considered an indexing scheme for a text containing wildcards and a query pattern without wildcards (Tam et al., 2009) and indexing a text without wildcards to search for patterns with wildcards (Bille et al., 2014).

We update the definition of MPHBs to permit some SNP values to be wildcards (represented as *'s in Figure 1). We assume that in each column there are samples for each SNP variation, e.g., there is at least one row with a 0 value and at least one row with a 1 value. This assumption guarantees that each block resolves to a single sequence of SNPs, which simplifies the representation of blocks. Following the notation from the paper that first introduced MPHBs (Cunha et al., 2018), we index SNPs (columns) as 1 through n and sequences (rows) as 1 through k. For a set of sequences S and a subset of the row indices K, we define $S|_K$ as the K-induced subset of S. That is, $S|_K :\ = \{s_i \in S | i \in K\}$.

Given k sequences $S = (s_1, s_2, \ldots, s_k)$ of length n, a *maximal perfect wildcard haplotype block* is a triple (K, i, j) with $K \subseteq \{1, 2, \ldots, k\}$, $|K| \geq 2$, and $1 \leq i \leq j \leq n$ such that

[1] Gianforte School of Computing, Montana State University, Bozeman, MT 59717, USA

[2] Lead Contact

*Correspondence: lucia.williams@montana.edu

https://doi.org/10.1016/j.isci.2020.101149

$$\begin{array}{llll} s_1: & 0 & * & * \\ s_2: & 1 & * & * \\ s_3: & * & 0 & * \\ s_4: & * & 1 & * \\ s_5: & * & * & 0 \\ s_6: & * & * & 1 \end{array}$$

Figure 1. Six Sequences with Three SNPs Each

There can be many possible maximal perfect wildcard haplotype blocks if many wildcards are present; in this example, there are 22 possible blocks, e.g., $(\{1,3,5,6\},1,2)$ is one of the blocks.

1. $\forall\ c \in [i,j],\ \left|\left(\bigcup_{s\in S|_K} s[c]\right) \cap \{0,1\}\right| = 1$ (consensus),
2. $i = 1 \vee \left|\left(\bigcup_{s\in S|_K} s[i-1]\right) \cap \{0,1\}\right| \neq 1$ (left-maximality),
3. $j = n \vee \left|\left(\bigcup_{s\in S|_K} s[j+1]\right) \cap \{0,1\}\right| \neq 1$ (right-maximality),
4. $\forall\ s' \in [1,k] \setminus K,\ \exists\ c \in [i,j],\ \left|\left(\bigcup_{s\in S|_{K\cup\{s'\}}} s[c]\right) \cap \{0,1\}\right| = 2$ (row-maximality).

Figure 1 shows a set of six sequences with three SNPs each. There, the ordered triple $(\{1,3,5,6\},1,2)$ is a maximal perfect wildcard haplotype block, because it satisfies all four of the above conditions:

5. At every column $c \in [1,2]$, the sequences included in the block of rows s_1, s_3, s_5, s_6 contain either a 0 or a 1 but do not contain both.
6. i = 1, so the block cannot be extended left.
7. Among sequences $s \in S|_{\{1,3,5,6\}}$, there exists at least one with $s[j+1] = 0$ and at least one with $s[j\ +\ 1] =$ 1, so the block cannot be extended right.
8. For every sequence s' not included in the block, s' disagrees with a sequence s included in the block at at least one position, so the block cannot contain any additional rows.

By contrast, the ordered triple $(\{1,2\},1,2)$ is not an MPWHB, since $s_1[1] = 1$ and $s_2[1] = 0$, violating the first (consensus) property.

The *maximal perfect wildcard haplotype block* (MPWHB) problem is to find all maximal perfect wildcard haplotype blocks for a given set of sequences. In previous work (Cunha et al., 2018), it was shown that in the original formulation of the problem without wildcards, there are at most k possible maximal perfect haplotype blocks that end at each column in the dataset; thus there are most kn maximal perfect haplotype blocks in total. The number of maximal haplotype blocks grows considerably if wildcards are present. The example shown in Figure 1 illustrates a simple construction in which for each column (SNP position), there is exactly one row with a 0 value and one row with a 1 value.

In general, this construction has n SNPs and $k = 2n$ sequences, with $s_{2i-1}[i] = 0$ and $s_{2i}[i] = 1$ for $i = 1, \ldots, n$ and all other values set to be wildcards. It is easy to check that, for any $1 \le i \le j \le n$ and any binary string b of length $j - i + 1$, there is a subset K of the sequences such that (K,i,j) forms an MPWHB that agrees with b in all non-wildcard positions. Summing over the possible lengths, the total number of MPWHBs is

$$\sum_{l=1}^{n} (n+1-l) \cdot 2^l, \qquad \text{(Equation 1)}$$

as there are $n+1-l$ starting positions for a block length l and 2^l possible agreed upon strings. In practice, we expect that there will be many fewer solutions in normal biological inputs. Motivated by this, we propose the objective of finding an algorithm whose running time is linear in the number of blocks found.

In the remainder of the paper, we will refer to MPWHBs simply as *blocks* for convenience.

In the Supplemental Information Transparent Methods Section S, we discuss how to represent all blocks using trie data structure and describe an algorithm for finding all solutions using depth-first search to implicitly search the trie. A trie (pronounced "try") is a tree where all nodes are labeled with a single letter

Wildcards	Min. Area	Run Time	# Blocks	#DFS Calls/Block	$\|K\|$	#SNP
0%		13 min 28 s	148,613,645	35.5	1,497.3	490.0
0%	500,000	19 min 37 s	16,076,453	294.5	1,498.5	690.4
0%	1,000,000	18 min 11 s	2,228,762	1,888.4	1,659.1	889.2
0%	2,000,000	13 min 47 s	4,779	660,363.0	1,634.9	1,287.9
5%		2 h 22 min	506,675,436	30.8	545.9	426.1
5%	500,000	2 h 12 min	18,155,762	815.9	1,477.3	710.9
5%	1,000,000	1 h 47 min	2,652,944	5,277.8	1,645.0	909.8
5%	2,000,000	2 h 9 min	13,387	926,786.2	1,546.8	1,380.9
10%		5 h 32 min	1,128,831,659	27.3	334.4	439.9
10%	500,000	4 h 18 min	20,144,453	1,471.3	1,455.3	736.5
10%	1,000,000	5 h 21 min	3,101,221	9,157.3	1,627.0	1,627.0
10%	2,000,000	5 h 3 min	36,145	721,431.8	1,506.1	1,452.0

Table 1. Summary of Experiments Varying the Wildcard Frequency and Minimum Block Area for an SNP Dataset for Human Chromosome 22 Consisting of 5,008 Sequences and 1,055,454 SNPs

The rightmost three columns are averages.

from an alphabet; words that are present in a collection can be re*trie*ved by following any path from the root (top) of the tree to any terminal (leaf) node. For us, the "words" in the trie are representations of the blocks in the set of sequences. Depth-first search is a standard method for searching through a tree data structure that we can use to explore the entire trie systematically and efficiently. Figure S.1 shows an example of the trie data structure built over a small dataset; Algorithm 1 describes the algorithm. The running time of the algorithm is $O(kn \cdot T)$, where T is the total number of blocks found.

RESULTS

We tested our method for finding blocks using the human chromosome 22 dataset from 1000 Genomes Project (1000 Genomes Project Consortium et al., 2015) that was also used by others studying MPHBs (Alanko et al., 2020). This was the smallest dataset they looked at, but it still consists of $5,008$ sequences and $1,055,454$ SNPs and so is representative of current larger SNP population datasets. Our implementation, written in Java, employs a built-in parallel loop construct for multi-threading. The algorithm was run on a high-performance research cluster (Hyalite High Performance Computing System, operated and supported by University Information Technology Research Cyberinfrastructure at Montana State University), and each experimental run was done on a single Intel Xeon Ivy Bridge or similar node with either 32 or 40 cores available and 256 Gb of memory.

Table 1 summarizes the results as we varied both the percentage of random wildcards (defined as the probability that a specific SNP call is a *) and the minimum *block area* threshold. The area of a block (K, i, j) is defined as $|K| \cdot (j - i + 1)$, the number of samples in the block times the width of the block in terms of number of SNPs. The minimum block area is given as a parameter to the algorithm and is also used to prune the trie search; paths that cannot lead to a block that meets the minimum area requirement are not explored. We did not optimize I/O, so only start timing each run after the SNP input file has been read into memory (which takes several minutes). We note that, although our experiments follow the general form of those performed in previous work (Alanko et al., 2020), the running times presented here are not directly comparable with those reported there, since the implementations are written in different languages and the experiments performed on different systems. It is possible to optimize the DFS function to receive only the indices of the rows that are lost at each call, meaning that Algorithm 1 takes $O(k\lg k + n\lg k)$ space, accounting for the storage of up to k row indices along the currently explored trie path, and the length n cons.cnt vector. However, the algorithm reads the entire k by n matrix into RAM, so this dominates memory use. For this reason, we did not record memory usage in our experiments.

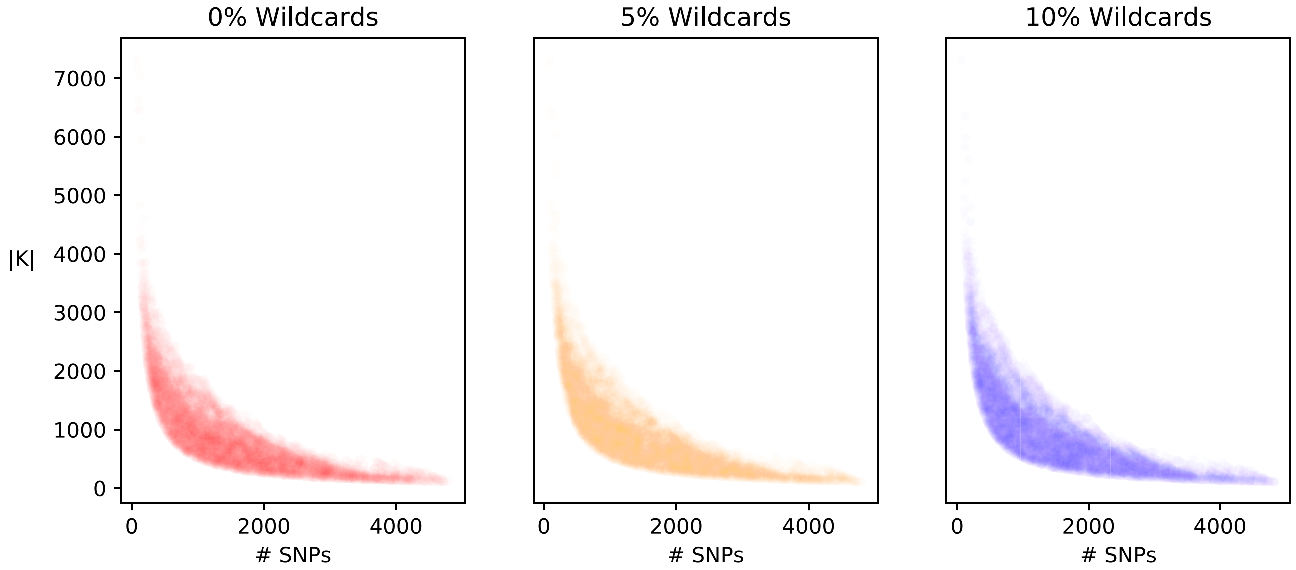


Figure 2. Scatterplots Showing the Distributions of Maximal Perfect Wildcard Haplotype Block Shapes Found for the Different Wildcard Rates and the Minimum Block Area Threshold Set to 500,000

Although block statistics vary significantly when there is no minimum area threshold, we see that the distributions of the larger blocks are not drastically different when wildcards are present. We also plot the distributions of block shapes found for the varying wildcard rates and a minimum block area of $500,000$ in Figure 2.

DISCUSSION

In this work, we give the first method for computing maximal perfect haplotype blocks in the presence of missing data. To do so, we define the maximal perfect wildcard haplotype block problem and give an output-linear time algorithm to solve it, with a running time of $O(kn \cdot T)$, where T is the total number of blocks found. Although it is possible to find maximal perfect haplotype blocks (without wildcards) in $O(kn)$ time (Alanko et al., 2020), that method cannot be directly adapted to the wildcard setting, because rows with wildcards cannot be sorted. It remains to be shown whether a faster algorithm can be given in the wildcard case.

Our experimental results suggest that randomly distributed missing values do not substantially alter the distribution of larger blocks. Thus, for applications such as estimating the selection coefficients of loci along a genome, missing SNP values should be tolerable. We plan to investigate this further on synthetic and actual datasets with missing values. In recent work (Williams and Mumey, 2020), we examined a version of the maximal perfect haplotype problem for pangenomic data. It would also be natural to consider missing values in those data; this would lead to a version of the problem where the sequences are paths in a graph and a wildcard could indicate that a path goes through an SNP but the value was unable to be called.

Limitations of the Study

We focused on updating the theory around maximal perfect haplotype blocks to account for unknown SNPs and giving an algorithm to find all blocks with wildcards. The main application of finding blocks has been to compute selection coefficients, so an important next step is to develop a method and a tool to compute selection coefficients in the presence of unknown SNPs.

In our experiments, we used only one human chromosome; future work could study a larger set of human chromosomes or datasets from other organisms. Additionally, we looked at only randomly distributed unknown values; real data may have unknown SNPs in a non-random fashion that could skew the distributions of large blocks.

We proposed an algorithm for finding all blocks with an asymptotic runtime of $O(kn)$ time per block found; an interesting algorithms question is whether blocks can be found asymptotically faster than this.

Resource Availability

Lead Contact

Requests for further information should be directed to and will be fulfilled by the Lead Contact, Lucia Williams (lucia.williams@montana.edu).

Materials Availability

No materials were generated in this study.

Data and Code Availability

VCF files from phase three of the 1000 Genomes Project are available from ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/. We used chromosome 22 and processed it to a binary haplotype matrix using the program vcf2bm, available at https://gitlab.com/bacazaux/haploblocks. Code to run our algorithm and experiments on a binary matrix is available at github.com/msu-alglab/WildHap/.

METHODS

All methods can be found in the accompanying Transparent Methods supplemental file.

SUPPLEMENTAL INFORMATION

Supplemental Information can be found online at https://doi.org/10.1016/j.isci.2020.101149.

ACKNOWLEDGMENTS

Support provided by US National Science Foundation grants DBI-1759522 and DBI-1661530.

AUTHOR CONTRIBUTIONS

Funding acquisition, B.M.; all other areas, equal contribution by both authors.

DECLARATION OF INTERESTS

The authors declare that they have no known competing financial interests or personal relationships that could have influenced the work reported in this paper.

Received: January 20, 2020
Revised: April 27, 2020
Accepted: May 5, 2020
Published: June 26, 2020

REFERENCES

1000 Genomes Project Consortium, Auton, A., Brooks, L.D., Durbin, R.M., Garrison, G.P., Kang, K.M., Korbel, J.O., Marchini, J.L., McCarthy, S., McVean, G.A., et al. (2015). A global reference for human genetic variation. Nature *526*, 68–74.

Alanko, J., Bannai, H., Cazaux, B., Peterlongo, P., and Stoye, J. (2020). Finding all maximal perfect haplotype blocks in linear time. Algorithms Mol. Biol. *15*, 2.

Bille, P., Gørtz, I.L., Vildhøj, H.W., and Vind, S. (2014). String indexing for patterns with wildcards. Theor. Comput. Syst. *55*, 41–60.

Cunha, L., Diekmann, Y., Kowada, L., and Stoye, J. (2018). Identifying maximal perfect haplotype blocks. Adv. Bioinform. Comput. Biol. *11228*.

Gillespie, J.H. (2004). Population Genetics: A Concise Guide (JHU Press).

Tam, A., Wu, E., Lam, T.W., and Yiu, S.M. (2009). Succinct text indexing with wildcards. In String Processing and Information Retrieval. SPIRE, J. Karlgren, J. Tarhio, and H. Hyyrö, eds. (Springer Berlin Heidelberg), pp. 39–50.

Williams, L., and Mumey, B. (2020). Extending maximal perfect haplotype blocks to the realm of pangenomics. In Algorithms for Computational Biology. AlCoB 2020. Lecture Notes in Computer Science, *12099*, C. Martín-Vide, M.A. Vega-Rodríguez, and T. Wheeler, eds. (Springer), pp. 41–48.

iScience, Volume *23*

Supplemental Information

Maximal Perfect Haplotype Blocks with Wildcards

Lucia Williams and Brendan Mumey

S. Transparent Methods

We describe the details of our trie-based representation of blocks and an algorithm that implicitly searches the trie to find all blocks.

S.1. A Trie-Based Representation of Blocks

Our approach uses a trie to identify blocks ending at each column. This idea was first proposed for the non-wildcard version of the problem by Cunha et al. (2018) and later improved upon by Alanko et al. (2019). The key observation is that, at each column, left-maximal blocks correspond to the internal branching nodes of the trie constructed over the reverses of input sequences up to that column. (Cunha et al. (2018) use right-maximal blocks and sequences from the current column to n.) In the wildcard case, the trie must contain valid haplotype blocks formed using wildcards; however, simply including every enumeration of a sequence's wildcards may yield internal blocks that do not satisfy the consensus constraint. For example, the sequences {`0*`, `1*`} do not form any blocks, but a trie constructed over {`00`, `01`, `10`, `11`} would have two internal branching nodes. Thus, we must be careful in defining how we construct the trie at a given column. In this section, we define a trie so that all internal branching nodes and a subset of the leaves correspond to blocks. For a column ℓ, we call this structure the *extended block trie at* ℓ and denote it by T_ℓ. We first define the strings over which T_ℓ is built, and then discuss useful properties of T_ℓ.

For every block (K, i, j) we define two types of strings: its *consensus pattern* and its set of *single-character extensions*.

Definition 1. *Let* (K, i, j) *be a block. The* consensus pattern *for* (K, i, j) *is the sequence without wildcards that matches to* $s[i..j]$ *for every sequence* $s \in S|_K$*.*

Because the first property (consensus) of the definition of block requires that at least one row take the value `0` or `1` for every column of the block, there is exactly one such consensus pattern for every block.

Definition 2. *Let* (K, i, j) *be a block with* $i > 1$ *and consensus pattern* P*. If at least one of the rows in* K *has a* `0` *(*`1`*) in the* $i - 1$ *position, then the string* `0`P *(*`1`P*) is the* `0` *(*`1`*)* single-character extension *for the block.*

A block (K, i, j) may have two, one, or zero single-character extensions, depending on the values in the $i - 1$ positions of sequences in $S|_K$.

For a given column ℓ, we denote the set of all blocks ending at ℓ as B_ℓ; that is, $B_\ell = \{(K, i, j) | j = \ell\}$. We note that because each block ending at column ℓ must have a unique consensus pattern, the total number of unique patterns associated to the blocks in B_ℓ is exactly the same as the number of blocks in B_ℓ.

The trie for column ℓ is built over two sets of strings:

1. The reverse of the consensus pattern for every block in B_ℓ.
2. The reverse(s) of the single-character extension(s) for every block (K, i, ℓ) in B_ℓ such that $i > 1$.

We label each edge of T_ℓ with either a `0` or a `1`. We also give each node two labels. The *path label* is the concatenation of all edge labels on the path from the root to the node. The *row label* is the subset of row indices $\{1, 2, \ldots, k\}$ corresponding to the rows that participate in a block whose reverse consensus pattern has the node's path label as a prefix. We say that these rows *support* the node. Figure S.1 shows an example of the extended block trie for a set of wildcard haplotypes at a column ℓ.

With the inclusion of single-character extensions in the trie, we have the following lemma.

Lemma 1. *Let T_ℓ be the extended block trie at ℓ. Every block in B_ℓ is represented as either a leaf node or an internal branching node in T_ℓ. That is, the reverse of the consensus pattern for every block in B_ℓ is the path label of either a leaf node or an internal branching node.*

Proof. Because T_ℓ is built using all blocks ending at ℓ, every block in B_ℓ certainly corresponds to a node in T_ℓ; we must show that such nodes are either branching nodes or leaves. We show that non-branching nodes cannot be blocks ending at ℓ.

Let u be an internal non-branching node in T_ℓ with path label p and depth d. Let $C \subseteq \{1, 2, \ldots, k\}$ be the row label of u and let v be the single child of u. Because u has only one child, all rows supporting u also support v. (If not, u would have a single-character extension as its other child.) Thus, the block $(C, \ell - d + 1, \ell)$ cannot be left-maximal, and so u cannot correspond to a block ending at ℓ. □

We note that leaf nodes that are not blocks must be single-character extensions. We call such nodes *failed extensions.*

To motivate the inclusion of single-character extensions in the trie, we also note that, without them, there could be a block corresponding to an internal node that does not branch, as in the following case. Consider a block (K, i, ℓ) whose consensus pattern P is a suffix of the consensus pattern for another block (K', i', ℓ) such that $K = K' \setminus h$ for some single sequence $s_h \in S$. Without loss of generality, assume that the string $\texttt{0}P$ is a suffix of the consensus pattern for (K', i', ℓ). Since only one sequence s_h supports (K', i', ℓ) but not (K, i, ℓ), the string $\texttt{1}P$ cannot be a suffix of any block ending at column ℓ, because blocks must be supported by at least two sequences. Thus, the node corresponding to (K, i, ℓ) would be an internal, non-branching node without the single-character extension $\texttt{1}P$. The block $(\{1, 3, 4\}, 2, 4)$ with consensus pattern `010` in Figure S.1 would be an internal, non-branching node if not for its single-character extension `1010`, as would be $(\{1, 3, 6\}, 2, 4)$.

We now prove properties of this trie that will help us to build it from the wildcard haplotype matrix.

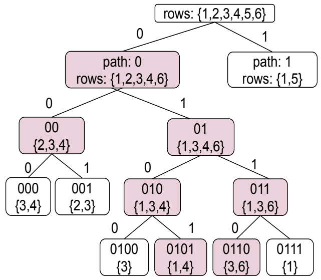
Lemma 2. *Let T_ℓ be the extended block trie at column ℓ built over k length n sequences of zeros, ones, and wildcards. If u is a node of T_ℓ whose parent is a branching node v, then u's row label is a strict subset of v's row label.*

s_1:	1	*	1	*	1	*
s_2:	1	1	0	0	1	*
s_3:	0	*	*	0	1	0
s_4:	1	0	*	0	0	1
s_5:	1	*	1	1	*	1
s_6:	0	1	1	0	0	*

(a) $\ell = 4$.

block/ extension	reverse pattern
$(\{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{6}\}, \mathbf{4}, \mathbf{4})$	0
$(\{1, 5\}, 4, 4)$	1
$(\{\mathbf{2}, \mathbf{3}, \mathbf{4}\}, \mathbf{3}, \mathbf{4})$	00
$(\{\mathbf{1}, \mathbf{3}, \mathbf{4}, \mathbf{6}\}, \mathbf{3}, \mathbf{4})$	01
$(\{3, 4\}, 2, 4)$	000
$(\{3, 6\}, 2, 4)$	001
$(\{\mathbf{1}, \mathbf{3}, \mathbf{4}\}, \mathbf{2}, \mathbf{4})$	010
$(\{\mathbf{1}, \mathbf{3}, \mathbf{6}\}, \mathbf{2}, \mathbf{4})$	011
$(\{3\}, 1, 4)$	0100
$(\{\mathbf{1}, \mathbf{4}\}, \mathbf{1}, \mathbf{4})$	0101
$(\{\mathbf{3}, \mathbf{6}\}, \mathbf{1}, \mathbf{4})$	0110
$(\{1\}, 1, 4)$	0111

(b) All blocks (bold) and failed extensions at $\ell = 4$.



(c) T_4.

Figure S.1: An example extended block trie at column $\ell = 4$ for the six input sequences shown in S.1a. The trie is built over the seven blocks and five failed extensions shown in S.1b. Nodes that correspond to blocks are shown in pink in S.1c. Leaf nodes for patterns 001 and 1 are not continued because they are not right-maximal; the leaf node for pattern 000 is not continued because it does not satisfy the consensus condition. Related to Figure 1.

Proof. Let p be the path label of v. Because v is a branching node, p is a pattern of some block (K, i, ℓ) in B_ℓ. We must consider two cases: the case when u's path label is a prefix of the reverse of some other block (K', i', ℓ) with $i' > i$, and the case when u is a failed extension (and thus not a block).

Suppose that u's path label is a prefix of the reverse of some other block, as described above. Let p' be the path label of u. Since u is a child of v, p must be a prefix of p'. Thus, every row that matches to p must also match to p', and so $K' \subseteq K$. Now we must show that $K' \neq K$. Aiming for a contradiction, suppose that the same set of rows match p' as p, so $K' = K$. Then, (K, i, ℓ) is not left-maximal, contradicting the fact that it is a block.

Now suppose that u is a failed extension. By definition, the rows supporting u are a subset of the rows supporting v. As in the other case, if the two sets were equal then u could not be left-maximal, and thus u could not correspond to a block. □

We define a *terminal block node* as either a leaf in the trie that represents a block or a parent of an extension (and so is also a block).

Lemma 3. *Given k length n sequences S over the alphabet $\{0, 1, *\}$, the number of terminal block nodes in the extended block trie at column ℓ is at most $2^{\min(k-2, \ell)}$.*

Proof. Clearly ℓ is trivial bound on the depth of any trie node, as the levels in the trie correspond to SNPs $1 \ldots \ell$, so there at most 2^ℓ leaves. We observe that any path in the trie to a terminal block node goes through at most $k - 2$

branching nodes, since each branch eliminates at least one row from the path (Lemma 2) and we must have at least two remaining rows at the node since it is a block. If we collapse all paths of non-branching nodes and remove extensions, the trie becomes a binary tree with maximum depth $k-2$; it can have at most 2^{k-2} leaves. □

Corollary 1. *The maximum number of blocks ending at column ℓ is at most $2^{\min(k-1,\ell+1)}-1$.*

Proof. The number of internal, non-terminal block nodes is one less than the number of terminal block nodes. □

Consequently, an upper bound on the number of blocks possible in a data set with k rows (sequences) and n columns (SNPs) is

$$\sum_{\ell=1}^{n} 2^{\min(k-1,\ell+1)} \quad - n.$$

We remark that this bound is probably not tight since it grows significantly faster than Equation (1), the number of blocks found in the somewhat pathological construction shown in Figure 1.

S.2. A DFS-Based Algorithm

We make use of a depth-first search approach to implicitly search the trie described in Section S.1 in order to identify blocks. For each SNP column ℓ we would like to search for blocks that end at this column. As explained above, we can view blocks as paths in a {0,1}-trie, where the path specifies the reverse order of the consensus string of the block. We define the set $\text{set}_0(\ell) = \{i \in \{1,2,\ldots,k\} : s_i[\ell] = 0\}$ to indicate the rows (sequences) that are 0 for SNP ℓ. Similarly, we also define sets $\text{set}_1(\ell)$ and $\text{set}_*(\ell)$. Pseudocode is given in Algorithms 1 and 2. The main idea of the DFS function is to implicitly search the trie for new blocks that end at the given column, ℓ. The argument *rows* is a set containing the sequences that reach the current node in the trie, corresponding to the row label of the extended block trie described in Section S.1. In order for an edge to exist in the trie it must have at least one row (sequence) that supports it in the sense that row agrees with the edge's SNP setting. Also, trie branches are not followed if the block they represent is not right-maximal, or if there is a column in the block that only contains wildcards. These conditions are checked prior to making a recursive call to further explore the trie from the current node. It is easy to parallelize the loop in Algorithm 1 by allocating a separate `cons.cnt` array for each thread.

Note that each path in the trie is less than n in depth and terminates in a found block. As noted in Lemma 3, there are at most $k-1$ branching nodes along any path in the trie. We keep count of the number of non-wildcard (consensus) values at each SNP level in the trie path currently being explored. As rows get deleted along a path, these counts can be updated in $O(n)$ time per row deleted. The path is only continued if these counts stay positive. At most $k-1$ rows can deleted along any path, so the $O(kn)$ time bound until the next block is discovered holds in general. Thus the running time of the algorithm is $O(kn \cdot T)$, where T is the total number of blocks in the solution.

References

Alanko, J., Bannai, H., Cazaux, B., Peterlongo, P., Stoye, J., 2019. Finding all maximal perfect haplotype blocks in linear time, in: 19th International Workshop on Algorithms in Bioinformatics (WABI 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Cunha, L., Diekmann, Y., Kowada, L., Stoye, J., 2018. Identifying maximal perfect haplotype blocks, in: Brazilian Symposium on Bioinformatics, Springer. pp. 26–37.

Algorithm 1: FindBlocks

```
int $\ell$;
int cons.cnt[n];
for $\ell = 1, 2, \ldots, n$ do
    cons.cnt[$\ell$] = 0;
    DFS($\ell, \{1, 2, \ldots, k\}, \ell$, cons.cnt);
end
```

Algorithm 2: int DFS($i, \mathrm{rows}, \ell, \mathrm{cons.cnt}$)

```
branchs := 0;
for $b = 0, 1$ do
    kp := rows $\cap$ (set$_b(i) \cup$ set$_*(i)$);
    rm := rows $\cap$ set$_{1-b}(i)$;
    cons.cnt[$i$] := $\|$rows $\cap$ set$_b(i)\|$;
    ok = (cons.cnt[$i$] $> 0$);
    if $ok$ then
        branchs := branchs + 1;
    end
    for $r \in rm$ do
        for $c = i + 1, \ldots \ell$ do
            if $s_r[c] \neq *$ then
                cons.cnt[$c$] := cons.cnt[$c$] $- 1$;
                if $cons.cnt[c] \leq 0$ then
                    ok = false;
                end
            end
        end
    end
    ok := ok & ($\ell = n$ $\|$ (kp $\cap$ set$_0(\ell + 1) \neq \emptyset$ & kp $\cap$ set$_1(\ell + 1) \neq$
     $\emptyset$) $\|$ kp $\subset$ set$_*(\ell + 1)$);
    if $ok$ & $\|kp\| \geq 2$ & ($i = 1$ $\|$ $DFS(i - 1, kp, \ell, cons.cnt) \neq 1$) then
        found block: (kp, $i, \ell$);
    end
    for $r \in rm$ do
        for $c = i + 1, \ldots \ell$ do
            if $s_r[c] \neq *$ then
                cons.cnt[c] := cons.cnt[c] + 1;
            end
        end
    end
end
return $branchs$
```