A graph extension of the positional Burrows–Wheeler transform and it's applications

S KRISHNA SANDEEP REDDY
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
sandeepcherry697@gmail.com

GUTTULA SAI NAGA TEJA
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
sainagateja555@gmail.com

MUCHARLA DEEKSHA
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
mdeeksha2002@gmail.com

LOKAVARAPU VARSHA
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Amritapuri, Kerala, India
varshalokavarapu743@gmail.com

***Abstract*—We recommend the graph PBWT, an interpretation of the "Positional Burrows–Wheeler transform (PBWT)" to genomic graphs. A genome graph is a compressed illustration of a graphed collection of genomes. A haplotype relates to a limited kind of walk in a genomic network. The graph PBWT is a compressed interpretation of a collection of these graph-encrypted haplotypes that enables sub-haplotype match examinations. For graph PBWT creation and query functions, we provide effective algorithms. We utilise the graph PBWT to analyse the quantity of haplotypes consistent with arbitrary moves in a genome graph and the pathways travelled by mapped reads as an example; the results imply that haplotype consistency data can be basically included into graph-based read mappers. We approximate that a single large compute node might store and search for haplotype queries on the order of 100,000 diploid genomes, as well as all kinds of fundamental variation, using the graph PBWT.**

***Index Terms*—Positional Burrows–Wheeler transform, Genomic graphs, Algorithms, Haplotype.**

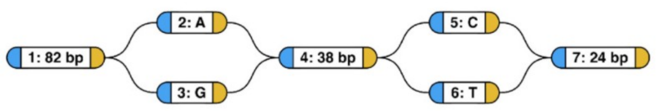
I. Introduction

In this paper,The Positional Burrows–Wheeler transform is a compact data model for preserving haplotypes that allows for fast sub haplotype match searches. The PBWT is an expansion of the standard Burrows–Wheeler transform, a technique for reducing string data that borrows some notions from the FM-index, a navigable version of the BWT. PBWT algorithms, example BGT, is used to acquire and retrieve the haplotypes of millions of specimens in a concise way. Current haplotype-based techniques can also use the PBWT to operate on far bigger data-sets of haplotypes than should have been possible. The haplotype address syndicate data-set, for instance, has 50,000 haplotypes, and PBWT-based equipment allows data at this level to easily guide phasing predictions on newly sequenced specimens with considerable speedups above other methodologies.

In a bidirected formulation, we define G= (V, E) as a genome graph. Each node in V contains a DNA sequence label, as well as a left (5) and right (3) side. In E, each edge is a pair of sides. The structure isn't really a multigraph because only single edge can link two sides, and therefore only single self-loop, or edge through one side towards itself, can exist on almost any specific side.

A primitive genome network can be built reasonably readily from a known sequences and a group of non - contiguous mutations, although more complex techniques are usually used in practise (identified as alternatives of a non-void substring of the source with a non-void another string). Begin by building a separate node that contains the complete reference chain. Split the nodes in the graph for every mutation to be introduced such that the variant's baseline genotype is reflected by a particular node. Build a node to symbolize the different genotype and connect the other allele's left and right sides to anything that is connected to the reference allele's left and right sides, correspondingly.

Fig. 1. Haplotypes in Genome graphs



We present the idea of an ambisequence to link up the worlds of bidirected graphs, where no alignment is better than almost any, and arithmetic, where integers subscripts are extremely useful. The alignment whereby an ambisequence is introduced is unimportant; a pattern as well as its overturn are indeed opposite and equal approaches of a certain fundamental ambisequence. A stick-shaped undirected graph is invariant to an ambisequence, as well as the alignments can be considered of as propagation of that graph from one side to another. A fundamental alignment is chosen randomly for each ambisequence s, and the apply the appropriate elements ”si” are the elements in that intentionally determined sequence. This alignment can also be used to determine concepts like ”previous” and ”after” in the perspective of an ambisequence.

We propose the term of a thread inside the graph G, that can be used to symbolize a haplotype or haplotype segment. A thread t on G is a non - void set ambisequence of sides where t2i and t2i+1 are opponents of one another 0iN sides and G includes an edge linking any set of t2i and t2i+1. To put it another way, a thread is the ambisequence variant of a walk across the graph's sides that combines visiting nodes and visiting edges and begins and finishes with nodes. It's worth noting that a thread can't be reversed because it's an ambisequence. The ”reversal” of a thread, on the other hand, is indeed one of the two possibilities.

We assume G to get a group of linked strands, indicated T, connected with it. With T provided G, we proposed an improved caching and retrieval methodology.

II. Background Study

Most of the graph's edges are (arbitrarily) sorted in relation to each other. The void edge, 0, is defined as a ratio that correlates to no distinct edge in the graph yet is smaller than any valid side. The opposite of an edge s is denoted by the expression s, that denotes the edge of s's node that is not s Lastly, the node to which an edge s relates is denoted by the form n(s).

Every point (which corresponds to a gene mutation) is a binary component in the PBWT, and the points are completely sorted. The PBWT accepts binary strings as source haplotypes, among each component denoting the status of a point. Every given haplotype is a traversal in a generic bidirected graph, or genomic graph, in the generalisation we provide. Because graph-based techniques to genomics challenges like mapping and variant recognition have indeed been demonstrated to outperform linear-reference-based methodologies, adopting the PBWT to a graph framework should be beneficial. There were other adaptations of BWT-based methods to the graph framework, but they mostly focus with the substring problem instead of the challenge of preserving and retrieving haplotypes.

Haplotypes can be incomplete (starting and ending at random nodes) and navigate random structural divisions using the PBWT generalisation provided here. Reduction doesn't really imply that the places (links in the graph) have a biologically meaningful arrangement. Although these generalisations, the PBWT's core characteristics remain intact. The basic data structures are equal, the reduction primarily relies on genetic similarity, and the haplotype matching technique is nearly identical. This generalisation of the PBWT is projected to support large, nested haplotype plates to guide read-to-graph orientation, graph-based mutation detection, and graph-based genomic data presentation, introducing the PBWT's advantages to genome graphs.

A. A graph extension of the positional Burrows–Wheeler transform

The elevated technique is to preserve T by clustering threads that have visited the very similar sequences of edges and collecting the upcoming edges that all those processes will examine in a single location. We regard the modern past of a thread to be a reliable predictor of where the thread is likely to move ahead, just as we do with the positional Burrows–Wheeler transformation, that is used to record haplotypes it against linear baseline, and the conventional Burrows–Wheeler transform. We may utilise optimal compression algorithms (including such run-length compression algorithms) and gain remarkable reduction by clustering together the next side data so that close elements are capable of exchanging data.
To put it another way, our strategy is as such. We consider an occurrence of side in an even-numbered location 2i a site inside an alignment; a thread might examine a specific side several instances in one or both configurations. (We establish it because a thread includes both the left and right sides of every node it contacts, but we always desire single visit to represent both.) Consider all visits of thread configurations in T to a side s. Consider the series of sides that came before this entry at s in the thread and invert this for every session, and afterwards arrange the entries lexicographically by these series of sides, removing connections using an alternative global sorting of the threads. After which, for every encounter, go two steps ahead through its thread (previous s and s) to the edge chosen to represent the very next visit, and concatenate it to a list (or indeed the void edge when there is no next encounter). Using that list and make the collection Bs [] for side s following recurring for all the ordered entries to s. Figure 1 shows an illustrative B[] array and its interpretations. (Remember that collections are stored from 0 continuously and can construct their dimensions on command.)

E possesses two configurations for every unoriented side s, s (s, s) and (s, s). Consider c () be a function of such oriented sides, as such c (s, s) is the least value in Bs [] of a visit of s that gets at s while visiting s, s for an oriented edge (s, s). Observe that c (s0, s) c (s1, s) for s0s1 being near to s because of the global processing of edges and the sort methods specified for Bs [] previously. A working illustration of a group of B [] arrays and the related c () method values can be found in Figure 2 and Table 1.

The conjunction of the c () method and the B [] matrices is referred to as a graph positional Burrows–Wheeler transformation for a particular G and T. (graph PBWT). We argue that a graph PBWT can represent T, and that calculating the number

Fig. 2. Graph extension of the positional Burrows–Wheeler transform

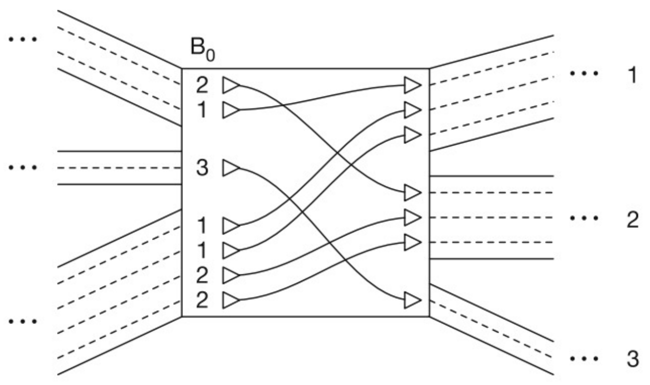
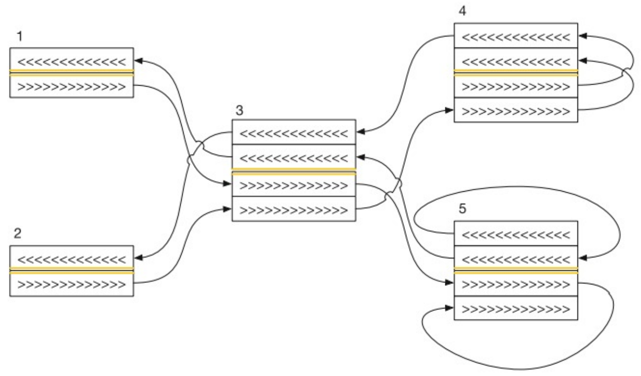


Fig. 3. Graph extension of the positional Burrows–Wheeler transform



of threads in T that incorporate a particular new thread as a sub thread is also effective.

III. Methods and it's applications

A. *Algorithm 1. Algorithm for extracting threads from a graph.*

Take each side s in G in turn to recreate T and the gPBWT. And we must create threads that begin at s that takes the minimum of c (x, s) to all the sides x which will be adjacent to s. When s consists of no edges, we take the length of Bs [], and we name this number b. For the new thread to begin, the i must exclusively run from 0 to b which will have the side S. And if it is on the null side of the thread, we must avoid traversing and obtain the thread and start from the original node s again with i in which the next value will be less then b. If at all it's not on the null side, we traverse to a new side s' and then calculate the sum of initial or arrival index of i' (new value of i) and number of entries in Bs []. Hence, it obtains the index in s' when the thread is being extracted. This algorithm process will help in enumerating all the threads in a graph, in a thread it takes place twice which is once from one end and the other from the other end, these threads must be duplicated into two new enumerated threads which in turn gives us one thread.

B. *Algorithm 2. Algorithm for embedding a thread in a graph.*

Here is an efficient algorithm for gPBWT construction where the process of construction of gPBWT is reduced into the embedding of an thread. Every thread is embedded by two orientations in which it follows a proper order. In order to embed a thread t we must primarily look at the node n entering the thread. We then for every new entry fit in a Bto [] which lengthens the array. The new item is near the beginning, before all the entries for visits coming by edges, with the precise location defined by the thread orientations' arbitrary order. Therefore, if no alternate order of thread orientations emerges, the order established by their addition to the graph will serve, in which case the new item can be added to Bt0 []. Now name k as the location of the entry. If the thread isn't done, we first add one to c(s, t2) for each side s next to t2 and after t1 in the global side ordering. This changes the c() function to reflect the addition into Bt2 [] that is about to happen. After this is done, this adds up to the iteration into Bt2 []. Then, knowing that the present visit in t occurs at position k in Bt0 [], we determine the position at which the following visit in t should have its insertion in Bt2 []. Placing k to this value, we can go through all the steps again to embed into t until it gets exhausted and places its final step with a null side entry.

C. *Algorithm 3. Algorithm for searching for a sub thread in the graph.*

Some of the original PBWT's efficient haplotype search features are preserved in the generalised PBWT data structure provided here. We start by naming fi and gi as the first and the before last of the positions for the range of different threads. The very first step is to initialize f0 to 0 and go to length of Bt0 after initializing they all visit the node through t0. From the next steps we calculate f(i+1) and g(i+1) from fi and gi respectively by using the "WHERETO()" function. The process must take place until which we can come to conclusion whether the threads in the graph have no matches to t. Furthermore, given the final range obtained by counting the occurrences of a thread t, we may estimate the occurrences of any larger threads beginning with t by simply repeating the process with the further entries in the extended thread.

IV. Results

This process will enumerate all threads in the graph, and will enumerate each such thread twice (once from each end). The threads merely need to be reduplicated (such that two enumerated threads produce one actual thread, as the original collection of embedded threads may have had duplicates) in order to produce the collection of embedded threads T. Pseudo-code for thread extraction is shown in Algorithm 1.

Setting k to this value, we can then repeat the preceding steps to embed t2, t3, etc. until t is exhausted and its embedding terminated with a null-side entry. Pseudo-code for this process is shown in Algorithm 2.

Fig. 4. Algorithm for extracting threads from a graph.

```
Algorithm 1. Algorithm for extracting threads from a graph.
function STARTING_AT($Side$, $G$, $B[]$, $c()$)
    ▷ Count instances of threads starting at $Side$.
    ▷ Replace by an access to a partial sum data structure if appropriate.
    if $Side$ has incident edges then
        return $c(s, Side)$ for minimum $s$ over all sides adjacent to $Side$.
    else
        return LENGTH($B_{Side}[]$)
function RANK($b[]$, $Index$, $Item$)
    ▷ Count instances of $Item$ before $Index$ in $b[]$.
    ▷ Replace by RANK of a rank-select data structure if appropriate.
    $Rank \leftarrow 0$
    for all index $i$ in $b[]$ do
        if $b[i] = Item$ then
            $Rank \leftarrow Rank + 1$
    return $Rank$
function WHERE_TO($Side$, $Index$, $B[]$, $c()$)
    ▷ For thread visiting $Side$ with $Index$ in the reverse prefix sort order, get the
corresponding sort index of the thread for the next side in the thread.
    return $c(\overline{Side}, B_{Side}[Index]) + \text{RANK}(B_{Side}[], Index, B_{Side}[Index])$
function EXTRACT($G$, $c()$, $B[]$)
    ▷ Extract all oriented threads from graph $G$.
    for all Side $s$ in $G$ do
        $TotalStarting \leftarrow$ STARTING_AT($s, G, B[], c()$)
        for all $i$ in $[0, TotalStarting)$ do
            $Side \leftarrow s$
            $Index \leftarrow i$
            $Thread \leftarrow [s, \overline{s}]$
            $NextSide \leftarrow B_{Side}[Index]$
            while $NextSide \neq$ null do
                $Thread \leftarrow Thread + [NextSide, \overline{NextSide}]$
                $Index \leftarrow$ WHERE_TO($Side, Index, B[], c()$)
                $Side \leftarrow NextSide$
                $NextSide \leftarrow B_{Side}[Index]$
            yield $Thread$
```

Fig. 5. Algorithm for embedding a thread in a graph.

```
Algorithm 2. Algorithm for embedding a thread in a graph.
procedure INSERT($b[]$, $Index$, $Item$)
    ▷ Insert $Item$ at $Index$ in $b[]$.
    ▷ Replace by INSERT of a rank-select-insert data structure if appropriate.
    LENGTH($b[]$) $\leftarrow$ LENGTH($b[]$) $+ 1$ ▷ Increase length of the array by 1
    for all $i$ in $(Index, \text{LENGTH}(b[]) - 1]$, descending do
        $b[i] \leftarrow b[i-1]$
    $b[Index] = Item$
procedure INCREMENT_C($Side$, $NextSide$, $c()$)
    ▷ Modify $c()$ to reflect the addition of a visit to the edge $(Side, NextSide)$.
    for all side $s$ adjacent to $NextSide$ in $G$ do
        if $s > Side$ in side ordering then
            $c(s, NextSide) \leftarrow c(s, NextSide) + 1$
procedure EMBED($t$, $G$, $B[]$, $c()$)
    ▷ Embed an oriented thread $t$ in graph $G$.
    ▷ Call this twice to embed it for search in both directions.
    $k \leftarrow 0$ ▷ Index we are at in $B_{t_{2i}}[]$
    for all $i$ in $[0, \text{LENGTH}(t)/2)$ do
        if $2i + 2 < \text{LENGTH}(t)$ then
            ▷ The thread has somewhere to go next.
            INSERT($B_{t_{2i}}[], k, t_{2i+2}$)
            INCREMENT_C($t_{2i+1}, t_{2i+2}, c()$)
            $k \leftarrow$ WHERE_TO($t_{2i}, k, B[], c()$)
        else
            INSERT($B_{t_{2i}}[], k$, null)
```

Assuming that the B[] arrays have been indexed for O(1) rank queries, the algorithm is O(N) in the length of the subthread t to be searched for, and has a run time independent of the number of occurrences of t. Pseudocode is shown in Algorithm 3.

Fig. 6. Algorithm for searching for a sub thread in the graph.

```
Algorithm 3. Algorithm for searching for a subthread in the graph.
function COUNT($t$, $G$, $B[]$, $c()$)
    ▷ Count occurrences of subthread $t$ in graph $G$.
    $f \leftarrow 0$
    $g \leftarrow$ LENGTH($B_{t_0}[]$)
    for all $i$ in $[0, \text{LENGTH}(t)/2 - 1)$ do
        $f \leftarrow$ WHERE_TO($t_{2i}, f, B[], c()$)
        $g \leftarrow$ WHERE_TO($t_{2i}, g, B[], c()$)
        if $f \geq g$ then
            return 0
    return $g - f$
```

The gPBWT is a compressible representation of a set of these graph-encoded haplotypes that allows for efficient subhaplotype match queries. We give efficient algorithms for gPBWT construction and query operations. We describe our implementation, showing the compression and search of 1000 Genomes data. As a demonstration, we use the gPBWT to quickly count the number of haplotypes consistent with random walks in a genome graph, and with the paths taken by mapped reads; results suggest that haplotype consistency information can be practically incorporated into graph-based read mappers

V. Conclusion

The core idea behind PBWT is to sort haplotypes in reversed prefix order. As a result of this sorting, it is possible to find haplotype matches quickly. Durbin (2014) uses Algorithm 1 to create a "positional prefix array," which is simply a list of haplotype indices that would arrange the haplotypes in inverted prefix order at position k. It's worth noting that the original BWT was created by Burrows and Wheeler (1994) for string data compression rather than search, and it's the basis for the bzip compression technique. We also looked at how to insert and delete particular haplotypes using efficient techniques.

References

[1] Adam M. Novak, Erik Garrison Benedict Paten, "A graph extension of the positional Burrows–Wheeler transform and its applications", July 2017.

[2] Guillaume Filion, Cesar H. Comin, A tutorial on Burrows-Wheeler indexing methods, July 2016.

[3] Khalid Sayood, "Data Compression", Encyclopedia of Information Systems 2003

[4] Phillip Compeau and Pavel Pevzner, "Bioinformatics algorithm, An active learning Approach", Vol.1. and Vol. 2 , 2015.

[5] JinXiong, "Essential Bioinformatics", Cambridge University Press, 2006

[6] Integer Linear Programming in Computational and Systems Biology, Dan Gusfield, University of California, Davis

[7] Karthik Raman, an Introduction to Computational Systems Biology (Systems Level Modeling of Cellular Networks), CRC Press, 2021.

[8] Gerald Karp, Chapter 15- Cell Signaling and Signal Transduction: Communication Between Cells, In Cell and Molecular Biology: Concepts and Experiments, 7e, Wiley, 2013