iScience, Volume *23*

Supplemental Information

Scalable Pairwise Whole-Genome Homology Mapping of Long Genomes with BubbZ

Ilia Minkin and Paul Medvedev

Scalable pairwise whole-genome homology mapping of long genomes with BubbZ
Supplemental Information

Ilia Minkin[*][†][1] and Paul Medvedev[‡][1, 2, 3]

[1]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, 16802, USA
[2]Department of Biochemistry and Molecular Biology, The Pennsylvania State University, University Park, PA, 16802, USA
[3]Center for Computational Biology and Bioinformatics, The Pennsylvania State University, University Park, PA, 16802, USA

1 Transparent Methods

1.1 Preliminaries

Strings and de Bruijn graphs

Let s be a string, indexed starting from 1. By s_i we denote the k-mer starting at position i of s. Given s and a positive integer k, we define a multigraph $G(s, k)$ as the *de Bruijn graph* of s. The vertex set consists of all substrings of s of length k, called k-mers. For each $(k + 1)$-mer substring x in s, we add a directed edge from u to v, where u is the prefix of x of length k and v the suffix of x of length k. Each occurrence of a $(k + 1)$-mer yields a unique multiedge, and every multiedge corresponds to a unique location in s. See Figure S1a for an example. Note that unlike some other definitions of a de Bruijn graph, a $(k + 1)$-mer that occurs multiple times in s will have multiple corresponding edges. The de Bruijn graph for a set of sequences S is $G(S, k) = \bigcup_{s \in S} G(s, k)$. That is, the vertex set of $G(S, k)$ is the union of all the vertex sets (where vertices with the same label are considered identical) and the multiedge set of $G(S, k)$ is the union of all the edge sets (but where each multiedge is preserved, even if it shares a label with another multiedge).

The set of a multiedges in a graph G is denoted by $E(G)$. We write (u, v) to denote a multiedge from vertex u to v. A *walk* w is a sequence of multiedges $((v_1, v_2), (v_2, v_3), \ldots, (v_{|w|}, v_{|w|+1}))$ where each multiedge (v_i, v_{i+1}) belongs to $E(G)$. The length of the walk w, denoted by $|w|$, is the number of multiedges it contains. A walk is *genomic* if it was generated by a substring in the input, that is, if the multiedges correspond to consecutive $(k + 1)$-mers in the input.

[*]ivminkin@gmail.com
[†]To whom correspondence should be addressed.
[‡]pashadag@cse.psu.edu

Chains

Consider two chromosome sequences s and t and their de Bruijn graph $G = G(\{s\} \cup \{t\}, k)$. There are several ways to mathematically define a homologous pair of substrings from s and t. The definition that lends itself to de Bruijn graph-based algorithms is that of a *chain* (Zhang *et al.*, 1994; Myers, 1995). In our context, a chain is, informally, a sequence of common k-mers that forms a sub-sequence (i.e. substrings allowing gaps) in both strings interleaved by potential point mutations or indels of bounded length. Formally, a *chain* c of *weight* n is two non-decreasing sequences of indices $(i_1, \dots, i_n)$ and $(j_1, \dots, j_n)$ such that $s_{i_x} = t_{j_x}$ and $i_x - i_{x-1} \leq b$ and $j_x - j_{x-1} \leq b$ and if $i_x = i_{x-1}$ then $j_x \neq j_{x-1}$, for all x. Each chain is associated with two genomic walks in G; specifically, the genomic walk corresponding to the substring of s starting from position i_1 and ending in position i_n, and, similarly, the genomic walk corresponding to the substring of t from position t_1 to t_n. See Figure S1a for an example of a chain.

Let $c = ((i_1, \dots, i_n), (j_1, \dots, j_n))$ and $c' = ((i'_1, \dots, i'_m), (j'_1, \dots, j'_m))$ be two chains. The *concatenation* of c and c' is the pair of sequences

$$c \cdot c' = ((i_1, \dots, i_n, i'_1, \dots i'_m), (j_1, \dots, j_n, j'_1, \dots, j'_m))$$

Note that $c \cdot c'$ is a chain iff $i'_1 \geq i_n$, $j'_1 \geq j_n$ and $i'_1 - i_n, j'_1 - j_n \leq b$ and either $i'_1 \neq i_n$ or $j'_1 \neq j_n$. In practice, we will be interested in the concatenation operation only if the result is a chain. We say that a chain c is *right-maximal* if there is no other chain c' such that $c \cdot c'$ is a chain, *left-maximal* if there is no other chain c' such that $c' \cdot c$ is a chain, and *maximal* if it is both left- and right-maximal.

1.2 Problem formulation and recurrence solution

To formulate the pairwise whole-genome homology mapping problem, let us take as input two chromosome sequences s and t, and a positive integer parameter b. As we discussed, we define a pairwise homology as a chain. One could then formulate the problem as that of outputting all maximal chains. However, such an output would contain a lot of redundancy, because two chains can span similar regions in s and t but contain different shared k-mers. To remove some of the redundancy, and with an eye towards an efficient algorithm, we use the notion of *(i,j)-maximum* chains. A chain is *(i,j)-maximum* if it ends in positions i and j in s and t, respectively, and has the highest weight among all such chains. Our problem formulation is then:

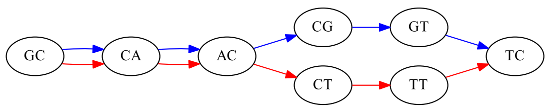
Problem Definition. *Given two sequences s and t and a positive integer b, output, for every $1 \leq i \leq |s|$ and $1 \leq j \leq |t|$, a maximal (i,j)-maximum chain, if it exists.*

This problem formulation lends itself naturally to dynamic programming, because (i,j)-maximum chains have an optimal substructure property. Formally,

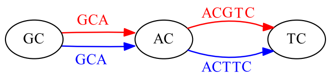
Property 1. *Let c be an (i,j)-maximum chain of length greater than one. Let us decompose it as $c = d \cdot ((i'), (j')) \cdot ((i), (j))$, where d may be empty. Let c' be any (i', j')-maximum chain. Then $c' \cdot ((i), (j))$ is an (i,j)-maximum chain.*

Proof. Let w be the weight of $d \cdot ((i'), (j'))$. The weight of c is $w + 1$. Since $d \cdot ((i'), (j'))$ ends in $((i'), (j'))$, any $((i'), (j'))$-maximum chain must have weight at least w. Hence the weight of c' is at least w, and the weight of $c' \cdot ((i), (j))$ is at least $w + 1$. Since this is the weight of c and c was (i,j)-maximum, $c' \cdot ((i), (j))$ is (i,j)-maximum as well. □

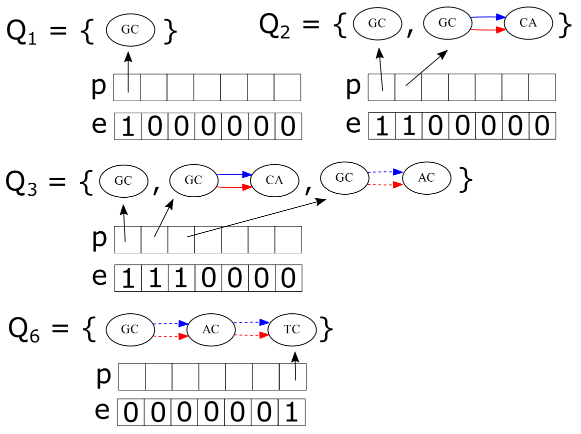
To determine the i' and j' of this Property, we will define the *predecessor* function. Consider a pair of positions i and j such that $s_i = t_j$. We define its possible *left-extensions* as the set of pairs



(a)



(b)



(c)

Figure S1: (a) De Bruijn graph built from strings $s =$"GCACGTC" and $t =$"GCACTTC", with $k = 2$. The two strings are reflected by the blue and red walks, respectively. The whole graph is chain $((1, 2, 3, 6), (1, 2, 3, 6))$. (b) The compacted version of the graph from panel (a); substrings generating the corresponding edges are shown adjacent to them. Note that the pair of edges between vertices "GC" and "AC" correspond to a case of parallel edges generated by identical substrings, while the edges between "AC" and "TC" form a bubble caused by a point mutation. (c) State of the list Q as well as vectors e and p, after considering each position i of the string s at which the algorithm adds a chain. The pointer in $p[j]$ leads to an element $C(i, j)$ of Q; $e[j] = 1$ if $p[j]$ is not a null pointer; $e[j] = 0$ otherwise. Q_1 shows the contents of Q after processing vertex "GC" (there is only one chain consisting of the initial k-mer); Q_2 contains the extended chain; and Q_6 has the whole graph minus the first chain that was removed due to being too far from the current position. Related to Table 1.

(i', j') such that $((i'), (j')) \cdot ((i), (j))$ is a valid chain. We define the *predecessor function* $\pi(i, j)$ to be the left-extension (i', j') such that the concatenation of an $((i'), (j'))$-maximum chain with $((i), (j))$ results in the chain of the highest weight. Formally,

$$\pi(i, j) = \arg\max\{\text{weight of } (i', j')\text{-maximum chain} \mid (i', j') \text{ is a left-extension of } (i, j)\}$$

Ties are broken by choosing the chain with a smaller value of j', and then with a smaller i' if the tie still exists. If there are no left extensions, we set $\pi(i, j) = \emptyset$. The predecessor function gives rises to our dynamic programming matrix C, where each entry $C(i, j)$ stores an (i, j)-maximum chain, as follows:

$$C(i, j) := \begin{cases} \emptyset & \text{if } s_i \neq t_j, \\ ((i), (j)) & \text{else if } \pi(i, j) = \emptyset, \\ C(\pi(i, j)) \cdot ((i), (j)) & \text{else} \end{cases} \tag{1}$$

The predecessor function can be computed by checking the value of C for all possible left-extensions. A solution to our problem is then to compute C and output every $C(i, j)$ that is also maximal.

Observe that an (i, j)-maximum chain is by definition left-maximal, so it suffices to check if $C(i, j)$ is right-maximal. This can be done easily, as follows. Observe that $C(i, j)$ is not right-maximal if and only if there are some offsets $1 \leq \alpha \leq b$ and $1 \leq \beta \leq b$ such that $s_{i+\alpha} = t_{j+\beta}$. In practice, b is quite small, when appropriate data structures are maintained (details omitted), we can check if a chain is right-maximal quickly. In what follow, we will therefore focus on just computing C.

1.3 High-level algorithm

Equation (1) immediately lends itself to a naive dynamic programming algorithm that uses a table where each cell corresponds to a row i and a column j. Such an algorithm can compute all $C(i, j)$ but will use $\Omega(|s||t|)$ memory, which is prohibitive. Instead, we present an algorithm that exploits the sparseness and structure of C as well as the fact that the maximum gap is limited by parameter b.

Let $C(i', j') = C(\pi(i, j))$ denote the predecessor chain of $C(i, j)$. First, we observe that if we compute the values of $C(i, j)$ in increasing order of i, we are guaranteed that the predecessor chain has already been computed, i.e. $i' < i$. Second, by definition of a valid chain, the predecessor chain must lie within the b previous columns, i.e. $i' \geq i - b$. Hence, it is not necessary to retain the whole table in memory, but rather, just the previous b columns. Third, the matrix is mostly sparse, since it only contains values when $s_i = t_j$. Therefore, storing it as a matrix is impractical. Instead, we will store the elements of the previous b columns in a queue Q that supports the lookup operation, $\mathit{Lookup}(Q, (i, j)) = C(i, j)$, if $C(i, j)$ is in Q. We will describe the implementation of the lookup function in Section 1.4.

The pseudocode of our method is in Algorithm 1. The outer for loop iterates over all the values of i. The inner for loop iterates over all values of j where $C(i, j) \neq \emptyset$. Lines 4 through 8 implement the logic of Equation (1). When column i is finished, lines 10 through 14 update Q by removing all chains from the now outdated column $i - b$ and, for those that are right-maximal, outputting them. Figure S1c shows an example of the contents of Q after several iterations.

Let us use $C(i, *)$ as shorthand for $C(i, j)$ for all j. The correctness of the algorithm follows from the previous discussion and the following theorem. For clarity, the pseudocode and the theorem omit some corner cases (e.g. when $i' = i$ or when we hit the end of the strings).

Algorithm 1 *Find-chains*

Input: strings s and t, graph $G(\{s\} \cup \{t\}, k)$, integers b and m

Output: the set of all chains in C that are right-maximal.

```
 1: $Q \leftarrow$ an empty doubly-linked list                         ▷ The set of current chains $C(i,j)$
 2: for $i \leftarrow 1$ to $|s|$ do
 3:     for all $j$ such that $t_j = s_i$ do                    ▷ Consider all position of $k$-mer $s_i$ in $t$
 4:         if $\pi(i,j) \neq \emptyset$ then
 5:             $r \leftarrow$ Lookup($Q$, $\pi(i,j)$)                         ▷ Equation (1)
 6:             PushBack($Q$, $r \cdot ((i),(j))$)
 7:         else
 8:             PushBack($Q$, $((i),(j))$)
 9:     let $c \leftarrow$ Front($Q$) and denote the end of $c$ as $(i',j')$
10:     while $i' < i - b$ do                                   ▷ Cleaning-up and outputting $Q$
11:         if $c$ is right-maximal then
12:             output $c$
13:         PopFront($Q$)
14:         let $c \leftarrow$ Front($Q$) and denote the end of $c$ as $(i',j')$
```

Theorem 1 (Correctness of Algorithm 1)**.** *At the end of the i-th iteration,*

1. *Q is the set of $C(i',*)$ for all $i - b \leq i' \leq i$, in front-to-back order of non-decreasing i'.*
2. *The algorithm's output has been the chains $C(i',*)$ which are right-maximal and for which $i' < i - b$.*

Proof. For the base case ($i = 0$), the statement holds since Q is empty. For the general case, the induction hypothesis tells us that at the start of the i-th iteration, Q contains $C(i',*)$ for all $i - b - 1 \leq i' \leq i - 1$, in order. To show (1), we will show that during the iteration, $C(i - b - 1, *)$ are popped from the front and $C(i,*)$ are pushed to the back. $C(i - b - 1, *)$ are popped from the front of Q during the while loop, using the fact that Q is in order. $C(i,*)$ are computed in the inner while loop using the logic of Equation (1), so all we need to show is that if $\pi(i,j) \neq \emptyset$, then $C(\pi(i,j)) \in Q$. Let $(i',j') = \pi(i,j)$. Because the gap between indices in a chain cannot exceed b, we have $i' \geq i - b$. By part (1) of the induction hypothesis, $C(i',*)$ is in Q, and hence $C(i,j)$ is pushed to the back of Q during the inner for loop.

Next we show (2). By induction, before the i-th iteration the output was $C(i',*)$ for all $i' < i - b - 1$. We need to then show that during the i-th iteration, the output is $C(i - b - 1, *)$. By part (1) of the induction hypothesis, the front of Q contains $C(i - b - 1, *)$. These elements will be popped during the while loop and output if they are right-maximal. □

1.4 Important details

There are additional aspects that the pseudocode does not address. We described the algorithm considering only the single strand of DNA. To handle both strands, we run a slightly modified version of our algorithm on the graph $G_{\text{comp}}(s,k) = G(s,k) \cup G(\bar{s},k)$, where $\bar{s}$ is reverse complement of s (Minkin *et al.*, 2017). We also preprocess the graph by removing all k-mers occurring more than a times, where a is a parameter. High-frequency k-mers can clog up our data structures and slow down the algorithm. We allow the user to set a, thereby controlling the trade-off between speed and potential decrease in accuracy. Finally, to save space, we do not store the actual chains in Q, but only their starting and ending coordinates, since this is what the final mapping will return anyway.

To support the computation of π and the lookup operation for Q (in Line 5), we need a specialized index. To quickly iterate over all left extensions of $((i), (j))$ we keep a bit vector e such that $e[j'] = 1$ if and only if Q contains a chain $C(i', j')$, for some i'. We also keep a vector p where $p[j']$ contains a pointer to an element $C(i', j')$ if one exists. If there are several such elements, we choose the one with the biggest i'.

Using a special machine instruction returning the number of trailing 0-bits, we can find all j' in the range of $j - b \leq j' \leq j$ such that $e[j'] = 1$ using using $\max(m, b/64)$ operations, where m is the number of ones in the range. In the GCC compiler the instruction is designated as __builtin_ctzll. Once we identify such values of j', we the use pointers in $p[j']$ to access the actual chains and select the one that yields the best predecessor for $C(i, j)$.[1]. Due to the nature of the de Bruijn graph, we expect the vector e to be sparse which results in efficient lookups. Figure S1c contains an example of state of vectors e and p during several iterations of running Algorithm 1

1.5 Adaptation of the algorithms to the compacted graph

For simplicity of exposition, we described our algorithm in terms of the regular de Bruijn graph. Our implementation, however, operates on the compacted de Bruijn graph which we build using TwoPaCo (Minkin *et al.*, 2017). The vertex set of the compacted graph is a subset of vertices of the regular graph, called *junctions*. Intuitively, a vertex is a junction if it is either a branching vertex or is the start or end of an input string (for an exact definition, please see Minkin *et al.* (2017)). The reason we can consider only junctions is that one can show that there is a one-to-one correspondence between the maximal chains in the ordinary graph and the ones in the compacted one. Particularly, any maximal chain starts and ends with a junction. Since the number of junctions is usually much smaller than the total number of k-mers, using only junctions greatly speeds up the algorithm and saves space, while not affecting the output of the algorithm.

A pair of vertices can be connected in the compacted graph by a pair of edge-disjoint genomic walks in two ways. These walks are either a pair of parallel edges representing a stretch of identical k-mers, or two walks forming a so called "bubble" which correspond to a sequence of point mutation or a short indels. Figure S1b shows an example of the compacted graph containing a pair of parallel edges and a bubble. To adapt our algorithm to the compacted graph, we modify the definition of chain such that it now consists of junction k-mers that are connected either by a pair of parallel edges or a bubble of size at most b.

Formally, two pairs of indices (i_1, j_1) and (i_2, j_2) are compatible if $s_{i_1} = t_{j_1}, s_{i_2} = t_{j_2}$ and either or both of the following holds: (1) $i_2 - i_1 \leq b$ and $j_2 - j_1 \leq b$; (2) $i_2 - i_1 = j_2 - j_1$ and $s_{i+p} = t_{j+p}$ for $i_1 \leq p \leq i_2$. The first condition models a bubble of size at most b, while the second one represents a stretch of identical k-mers corresponding to a pair of parallel edges in the compacted graph. Note that we have to handle the case of parallel edges separately because they might correspond to more than b k-mers and we should allow to "skip" over such pair of edges regardless of its length. A chain then is a pair of non-decreasing sequences of indices $(i_1, \ldots, i_n)$ and $(j_1, \ldots, j_n)$ such that $s_{i_x} = t_{j_x}$, each s_{i_x} (t_{j_x}) is a junction, (i_x, j_x) and (i_{x+1}, j_{x+1}) are compatible for $1 \leq x < n$ and if $i_x = i_{x-1}$ then $j_x \neq j_{x-1}$, for all x.

We adjust the code of as follows. In the loop in Lines 2 to 14 of Algorithm 1 we iterate over junctions of $\ell(s)$ instead of ordinary k-mers. We also modify our lookup procedure to take the new

[1]Our actual implementation does not store the vector p explicitly; instead we use a mapping from the k-mer set to the data structure Q.

definition of chain into account, as well the clean-up procedure in Lines 10 to 14. Particularly we handle two separate cases for extension of a chain: by a pair of parallel edges, or by a “bubble.”

1.6 Computational complexity

Here we will analyze the complexity of Algorithm 1. Let the maximum degree of a vertex considered by our algorithm be a. This could just be the maximum degree in the graph or it could be the parameter a set by the user. The number of iterations of the inner loop in Lines 4 to 8 is bounded by $a|s|$. Computing π and doing the lookup operation in Line 5 takes $O(b)$ operations in the worst case, as described in Section 1.4. The processing of Q in Lines 10 to 14 takes a total of $O(a|s|)$ time over the course of the algorithm, since this is the number of elements pushed into Q. As the result, the total time complexity is $O(ab|s|)$. The space complexity is dominated by the data structures to store the mappings for the shared k-mers. The amount of memory is strongly dependent on the structure of the input, and we therefore did not perform a worst case analysis.

1.7 Modes of operation

Algorithm 1 can also be used to find chains within a single genome, corresponding to duplications. To do this, the user should give as input a pair of identical sequences. To handle this case, we modify our algorithm to forbid chains that overlap with themselves. To implement this, we perform additional checks before concatenating chains in Line 5 (details omitted).

Algorithm 1 can also be used to compute an all-against-all mapping for a set of chromosomes $S = \{s_1, \ldots, s_{|S|}\}$. Rather than performing $\Theta(|S|^2)$ runs of the algorithm, we can modify the algorithm to run only $\Theta(|S|)$ times, at a potential cost of more memory, as follows. We first compute the de Bruijn graph from all of S, i.e. $G(S, k)$. Then we run Algorithm 1 $|S|$ times; in the i^{th} run, s_i plays the role of s and the chromosomes $\{s_i, \ldots, s_{|S|}\}$ play the role of t. In our pseudocode, t is a single string; but, we can easily modify it to allow t to be a set of strings by considering positions in all sequences of $\{s_i, \ldots, s_{|S|}\}$ in Line 2. It may also be that the underlying graph $G(S, k)$ could have vertices from some chromosome s_j that is not part of the comparison (i.e. $j < i$); however, since the algorithm only looks at k-mers that appear in s or t, those extra k-mers would not effect the execution of the algorithm. This approach to all-against-all mapping will give the same results as the naive $O(|S|^2)$ runs approach. However, it does have an associated memory cost, since we must maintain in memory a de Bruijn graph of $|S|$ sequences, rather than just the graph of 2 sequences. This strategy also lends itself to parallelization, by executing these $|S|$ runs in parallel using multithreading. Finally, note that the same strategy applies to all-against-all mapping of multiple multi-chromosomal genomes, since our algorithm does not distinguish between chromosomes on the same vs. different genomes.

References

Minkin, I., Pham, S., and Medvedev, P. (2017). Twopaco: an efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, **33**(24), 4024–4032.

Myers, G. (1995). Chaining multiple-alignment fragments in sub-quadratic time.

Zhang, Z., Raghavachari, B., Hardison, R. C., and Miller, W. (1994). Chaining multiple-alignment blocks. *Journal of Computational Biology*, **1**(3), 217–226.