Supplement

7 Illustration of Succinct Colored de Bruijn Graph

Below we illustrate the succinct colored de Bruijn graph data structure that was presented by Muggli *et al.* (2017) and is used in this paper.

Fig. 2: **Left:** A colored de Bruijn graph consisting of two individual graphs, whose edges are shown in yellow and green. The nodes are present in either the yellow graph, the green graph, or both graphs and thus, are shown in lime. **Center:** The co-lexicographically sorted nodes, with the number of incoming edges shown on their left and the labels of the outgoing edges shown on their right. The edge labels are shown in yellow or green if the edges occur only in the respective graph, or lime if they occur in both. **Right:** The Vari representation of the colored de Bruijn graph: the edge-BWT is the list of edge labels. B_F and B_L are bit vectors to track the incoming and outgoing node degree, respectively. Finally, the binary array C (shown transposed) indicates which edges are present in which individual graphs.

8 k-mer Recovery

If we know the range $\mathsf{B}_L[i..j]$ of k-mers whose starting nodes end with a pattern P of length less than $(k-1)$, then we can compute the range $\mathsf{B}_F[i'..j']$ of k-mers whose ending nodes end with Pc, for any character c, as follows:

$$\begin{aligned} i' &= |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| \\ &\quad +|\{h : \mathsf{Edge\text{-}BWT}[h] = c,\ h < i\}| \\ j' &= |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| \\ &\quad +|\{h : \mathsf{Edge\text{-}BWT}[h] = c,\ h \leq j\}| - 1\,. \end{aligned}$$

Thus, we can find the interval in B_L containing v's outgoing edges in $O(k \log\log \sigma)$-time, provided there is a directed path to v of length at least $k-1$. Thus, we add extra nodes and edges to the graph to ensure there is a directed path of length at least $k-1$ to each original node. More formally, we augment the graph so that each new node has a $(k-1)$-mer that is prefixed by one or more copies of a special symbol \$ not in the alphabet and lexicographically strictly less than all others. When new nodes are added, we are assured that the node with k-mer $\$^{k-1}$ is always first in colex order and has no incoming edges. Lastly, we augment the graph in a similar manner by adding an extra outgoing edge, labeled \$, to each node with no outgoing edge.

9 Pseudocode for the Naive Merge Algorithm

Below is a detailed sketch of the naive merge algorithm.

Algorithm 3 Naive Merge Algorithm. Because L_1 and L_2 are explicitly constructed, a large amount of memory is needed.

$\mathsf{L}_1 \leftarrow \emptyset$
$\mathsf{L}_2 \leftarrow \emptyset$
Populate L_1 and L_2 (See "k-mer Recovery" of Subsection 3)
Merge L_1 and L_2 into L_M.
Create $\mathsf{Edge\text{-}BWT}(G)_M$, B_{LM}, B_{FM} from L_M
Create C_M from C_1 and C_2

10 Illustration of merge plans

We provide an example of merge plans in two conceptual edge lists L_1 and L_2 from graphs G_1 and G_2 in Figure 3, where $k = 4$. As mentioned before the merge plan is defined as $P_1 = \{[0, p_1^1], \ldots, [p_i^1, |\mathsf{L}_1| - 1]\}$ where each $p_1^1, \ldots, p_i^1$ is an index in L_1, and $P_2 = \{[0, p_1^2], \ldots, [p_i^2, |\mathsf{L}_2| - 1]\}$, where each $p_1^2, \ldots, p_i^2$ is an index in L_2. Next, all revisions of P_1 and P_2 are computed iteratively. In this example, we start with the red merge plans P_1 and P_2 covering the whole L_1 and L_2 respectively. In the next iteration, based on the next letter (the order is right to left), we revise every interval into at most five subintervals (five being the number of alphabets: {\$, A, C, G, T}). In this example see how P_1 and P_2 in red are revised to P_1' and P_2' in green each with five intervals. Similarly, every interval in P_1' and P_2' are revised to at most five subintervals making P_1'' and P_2''.

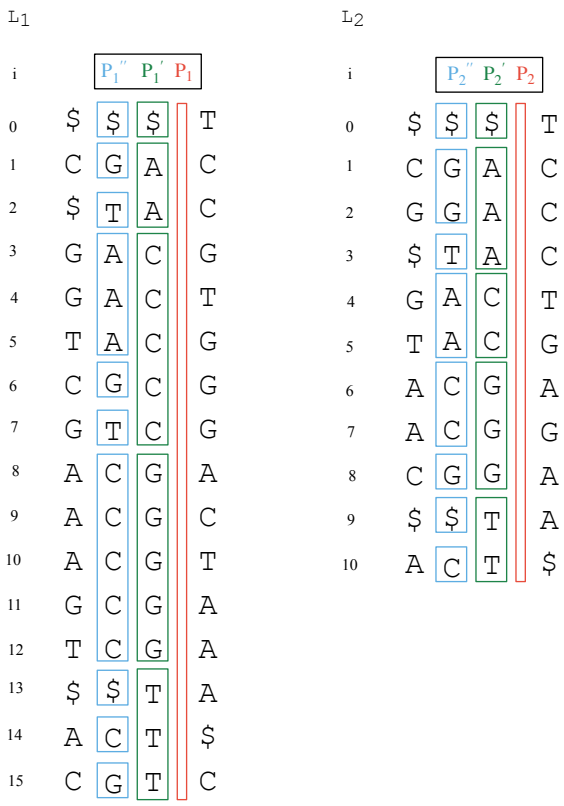

Fig. 3: Three merge plans. The initial merge plans are shown in red, where P_1 is initialized $\{[0, 15]\}$ and P_2 is initialized to $\{[0, 10]\}$. Green is the first refinement of P_1 and P_2. Thus, P_1' and P_2' consist of the continuous range of elements that have \$, A, C, G, and T in the next character position, e.g. $P_1' = \{[0, 0], [1, 2], [3, 7], [8, 12], [13, 15]\}$ and $P_1' = \{[0, 0], [1, 3], [4, 5], [6, 8], [9, 10]\}$. Blue is the third merge plan which is based on the refinement of P_1' and P_2'. P_1'' breaks each continuous range in P_1' into five (possibly empty) continuous ranges based on the next character position. For example, the first continuous range of P_1' is $\{[0, 0]\}$ broken into five ranges in P_1'', e.g., one for \$, A, C, G, and T. Yet, four of these five ranges are empty since there exists only a single element with \$ in the next character position.

11 Illustration of computing the next character of L

We provide an example of how we can navigate the set of edge k-mers without explicitly storing them. We compute the next character at each iteration with only three columns present in memory (shown with colors blue, red and orange).

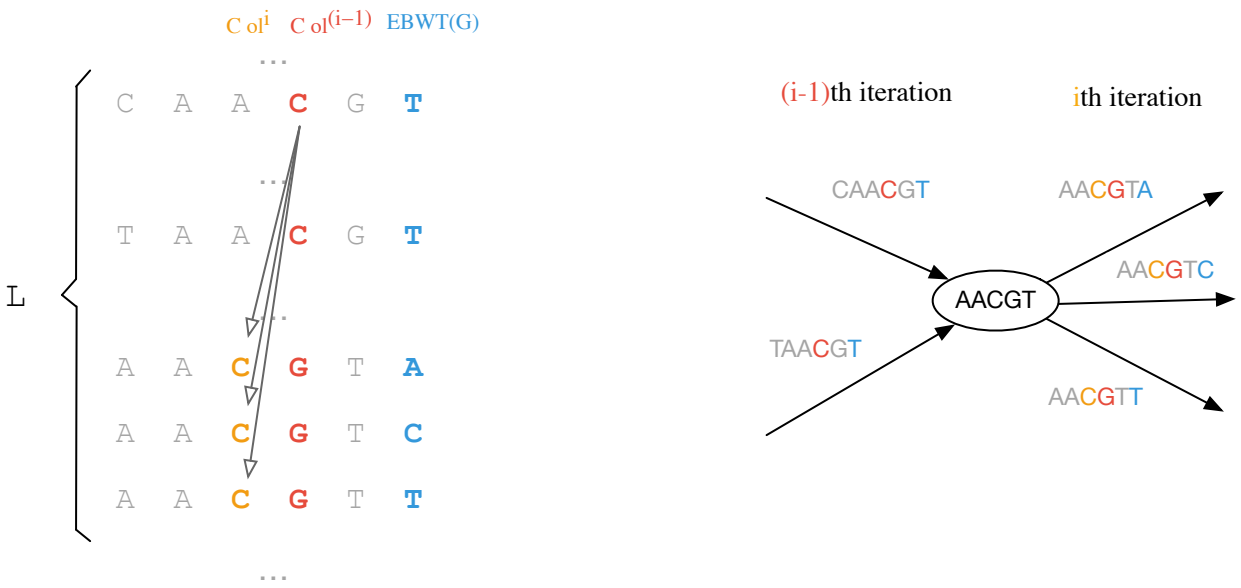

Fig. 4: Method for populating Col^i based on Col^{i-1} and graph navigation. Colored (blue, red and orange) nucleotides represent data that is in memory and valid. Grey represents data that is stored in external memory in Vari but is computed as needed and only exists ephemerally in VariMerge. Thus, only three columns are ever present in memory, which is a significant memory savings relative to the full set of edge k-mers. The three resident vectors are 1.) Edge-BWT(G) (which is always present and used for navigation), 2.) Col^{i-1} which is already completely populated when a new column to the left, 3.) (Col^i) is being generated. As one can see in the associated graph representation on right, the successor edges have almost the same k-mer but shifted by one position. e.g. the red colored C in $i-1$-th iteration in position 4 is shifted to position 3 and turned into orange in i-th iteration.

Algorithm 4 GetCol(i, $PrevCol$, G)

if $i = k$ **then**
  return $EBWT(G)$
else
  if $i = 1$ **then**
    $PrevCol \leftarrow EBWT(G)$
  end if
  $ResultCol \leftarrow []$
  for all $e \in \{1..|E|\}$ **do**
    $a \leftarrow \text{select}(\text{rank}(|\{d : d \in E,\ \mathsf{label}(d) \prec \mathsf{label}(e)\}| + 1, \mathsf{B}_{L_1}) + r - 1, \mathsf{B}_{L_1})$
    $b \leftarrow \text{select}(\text{rank}(|\{d : d \in E,\ \mathsf{label}(d) \prec \mathsf{label}(e)\}| + 1, \mathsf{B}_{L_1}) + r, \mathsf{B}_{L_1})]$
    for all $j \in \{a..b\}$ **do**
      $ResultCol[j] \leftarrow PrevCol[e]$
    end for
  end for
  return ResultCol
end if

12 Merging the secondary components

Delimiting common origin with B_{LM}. We prepare to produce B_{LM} in the planning step by preserving a copy of the merge plan after $k-1$ refinement iterations as S_{k-1}. After $k-1$ refinement steps, our plan will demarcate a pair of edge sets where their k-mers have identical $k-1$ prefixes. Thus, whichever merged elements in $\mathsf{Edge\text{-}BWT}(G)_M$ result from those demarcated edges will also share the same $k-1$ prefix. Therefore, while executing the primary merge plan, we also consider the elements covered by S_{k-1} concurrently, advancing a pointer into $\mathsf{Edge\text{-}BWT}(G)_1$ or $\mathsf{Edge\text{-}BWT}(G)_2$ every time we merge elements from them. We form B_{LM} by appending a delimiting 1 to B_{LM} (again, indicating the final edge originating at a node) whenever both pointers reach the end of an equal rank pair of intervals in S_{k-1}'s lists.

Delimiting common destination with flags_M. We produce flags in a similar fashion to B_{LM} but create a temporary copy of S_{k-2} in the planning stage after $k-2$ refinement iterations instead of $k-1$. In this case, the demarcated edges are not strictly those that share the same destination; only those edges that are demarcated and share the same final symbol. Thus, in addition to keeping pointers into $\mathsf{Edge\text{-}BWT}(G)_1$ or $\mathsf{Edge\text{-}BWT}(G)_2$, we also maintain a vector of counters which contain the number of characters for each (final) symbol that have been emitted in the output. We reset all counters to 0 when a pair of delimiters in S_{k-2} is encountered. Then, when we append a symbol onto $\mathsf{Edge\text{-}BWT}(G)_M$, we consult the counters to determine if it is the first edge in the demarcated range to end in that symbol. If so, we will not output a flag for the output symbol; otherwise, we will.

13 Proof of Theorem 1

Theorem 1. Given two colored de Bruijn graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ constructed for k such that $|E_1| \geq |E_2|$. It follows that our merge algorithm constructs the merged colored de Bruijn graph G_M in $O(|E_1| \cdot \max(k, t))$-time, where t is the number of colors in G_M.

Proof. In our merge algorithm, we will perform k refinements of P_1 and P_2 after they are initialized. We know by definition and Observation 1 that $|P_1| \leq |\mathsf{L}_1|$, $P_2 \leq |\mathsf{L}_2|$, $Col_1^i = |\mathsf{L}_1| = |E_1|$ and $Col_2^i = |\mathsf{L}_2| \leq |E_1|$ at each iteration i of the algorithm. Further, it follows from Observation 1 that a constant number of operations are performed to P_1, P_2, Col_1^i and Col_2^i. Thus, the planning step will require $O(|E_1|k)$-time since there will k refines of P_1 and P_2, each of which has size at most $|E_1|$.

Executing the merge plan requires constructions of $\mathsf{Edge\text{-}BWT}$, B_F, B_L, and C_M from the merge plan. Construction of $\mathsf{Edge\text{-}BWT}$, B_F and B_L requires $O(|E_1|)$-time since a constant number of operations are needed for each item of the merge plan and there are at most $|E_1|$ items in P_1 and P_2. Next, in order to construct C_M, we perform a constant number of operations for each element of C_M. Since C_M is a binary matrix of size $2|E_1| \times t$ this will require $O(|E_1|t)$-time. Thus, the total merge algorithm requires $O(|E_1|k + |E_1|t)$-time which is equal to $O(|E_1| \cdot \max(k, t))$.

14 Details Concerning Strains of E. coli K-12

Table 4 describes the accession number, sub-strain and genome length of the E.coli genomes used for validation of our construction and bubble-calling.

Accession Number	Sub-strain	Genome Size
AP009048	W3110	4,646,332 bp
CP009789	ER3413	4,558,660 bp
CP010441	ER3445	4,607,634 bp
CP010442	ER3466	4,660,432 bp
CP010445	ER3435	4,682,086 bp
U00096	MG1655	4,641,652 bp

Table 4. Characteristics of the substrains of E. coli K-12 used to test the performance and accuracy of VariMerge

.

We validate VariMerge by generating two succinct colored de Bruijn graphs with three *E. coli* assemblies each, merge them, and verify the correctness of the merged graph. First, we generated all k-mers for each reference genome, counted all unique k-mers with KMC2 Deorowicz *et al.* (2015), constructed two de Bruijn graphs of three assemblies each using Vari, and merged them into a six-color graph using VariMerge. Independently, we constructed a second colored de Bruijn graph using Vari on all six assemblies in one run and then compared these two graphs. We found VariMerge produced files on disk that were bit-for-bit identical to those generated by Vari, demonstrating they construct equivalent graphs and data structures.