1 SUPPLEMENTARY MATERIAL

1.1 Algorithm: from alignments to compressed de Bruijn grpah

In this section we present an algorithm that constructs a compressed de Bruijn graph from the set of self-alignments of length $\geq k$ in the genome. We use mummer (Kurtz *et al.*, 2004) to preprocess the genome and efficiently locate all self-alignments in the genome whose lengths are at least k. Our alignment-based algorithm begins with a graph consisting of one large node to represent the entire genome. Then, the algorithm considers one alignment at a time. As each alignment is incorporated into the graph, the nodes are split to represent smaller subsequences of the genome. Occasionally, nodes are merged when a repetition is detected in the genome. Thus our algorithm achieves a runtime that is related to the number of self-alignments bounded by k. The number of self-alignments shrinks rapidly as k grows. In contrast there is no such advantage to using large values of k in an uncompressed de Bruijn graph because the initial number of nodes is fixed by the genome size.

Algorithm 2 depicts our alignment-based algorithm for constructing the compressed de Bruijn graph that represents a genome. We exclude implementation details that ensure the correctness of the algorithm. Each node captures a distinct subsequence of length $\geq k$ in the genome. This is stored as a set of start positions and a length. We maintain separately a sorted set of all start positions in the graph, with pointers to the nodes that represent them, so that we can quickly navigate to a start position in the graph and easily query whether there is a node with a particular start position. Each distinct subsequence of length $\geq k$ in the genome is represented by exactly one node in the compressed de Bruijn graph. Each k-mer, which we denote by its start position in the genome, is included in exactly one node in the graph. This invariant is true in the final graph as well as during construction. In the final graph, there is one leaf, representing the end of the genome. However, during construction, we allow many nodes to be leaves, representing suffixes of the genome. At the end of construction, each leaf (except possibly the shortest one) has its sequence truncated and becomes a parent to the first node whose sequence begins within the leaf's sequence.

We now summarize the procedure of our algorithm as it processes an alignment. We first insert the starting position of the first interval in the alignment, alignBeg1, to the appropriate node and then we add the starting position of the second interval, alignBeg2, to the same node. If another node already represents alignBeg2, the nodes are merged. Before merging nodes that begin with identical subsequences, we ensure that they represent sequences of the same length and precede a merge by a split (splitBackwards) if one node's sequence is a proper prefix of the other's. When a starting position is added to a node, we ensure that the subsequence is removed from any other node that already captured it, splitting nodes as appropriate. Thus we ensure that there are no redundancies in the graph.

When an alignment is considered, there are several scenarios that can occur when we insert alignBeg1.

1. **align1.beg is a starting position of a node in the graph.**

   a. **The existing node represents a longer subsequence than the alignment.** In this case, we split the existing node to form a new node whose sequence is a proper prefix of the existing node's. This uses the splitBackwards routine. Then a new start position of align2.beg is inserted to the new node [if it was not already there].

   b. **The existing node represents a shorter subsequence than the alignment.** In this case, we insert the beginning of the alignment by inserting a new start position of align2.beg to the new node. Then the alignment is trimmed at its left end and we continue by iterating through the rest of the alignment.

   c. **The existing node represents precisely the first interval of the alignment.** In this case, align2.beg is added as a start position for the node.

2. **align1.beg is not a starting position of any node in the graph.** In other words, alignBeg1 is implicitly included within a node.

   a. **The closest existing node with a start position before align1.beg ends at align1.end** In this case, we use splitForwards to split the closest node with a start position less than align1.beg into two nodes. Then align2.beg is inserted as a start position to the node that represents a suffix of the original node.

   b. **The closest existing node with a start position before align1.beg extends past align1.end** In this case, we use splitMiddle to split the closest node with a start position less than align1.beg. This creates two new nodes. align2.beg is inserted as a start position to the new node that represents the middle of the original node.

As the alignments are considered, nodes in the graph are merged and split. There are three ways in which a node is split, which we call splitBackwards, splitForwards, and splitMiddle. The splitBackwards routine is used when an alignment is a prefix to an existing node. It splits the existing node into two nodes. The splitForwards routine is used when an alignment is implicitly contained within an existing node, is not a prefix, and the alignment is a suffix of the existing node. It splits the existing node into two nodes. The splitMiddle routine is used when an alignment is implicitly contained within an existing node, is not a prefix, and the alignment ends earlier in the sequence than the existing node. It splits the existing node into three nodes. The splitting routines are depicted in Figure 1.

Self-overlapping alignments contributed additional complexity to our algorithm. Self-overlapping alignments are tandem repeats in the genome. We break down each self-overlapping alignment into its smallest repeating unit and create a node to capture the tandem repeat with all of its start positions. Then we create a separate node that bridges the occurrences of the tandem repeats, forming a cycle in the graph. We create an edge between these two nodes with multipflicity to represent all recurrences of the tandem repeat.

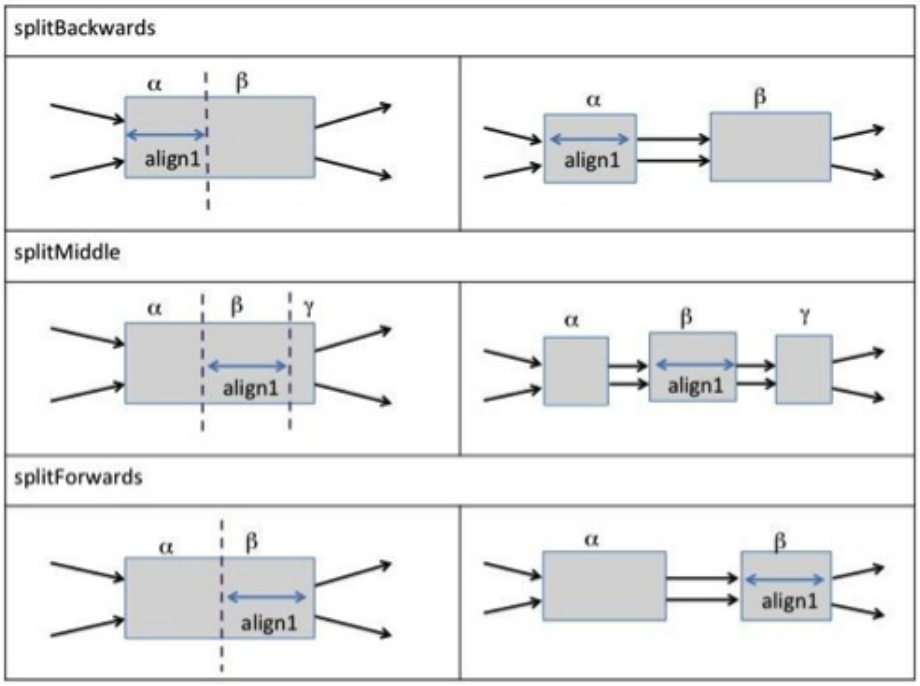

Fig. 1. The three splitting routines in our alignment-based algorithm. splitBackwards splits a node representing $\alpha\beta$ into separate nodes for α and β. splitMiddle splits a node representing $\alpha\beta\gamma$ into separate nodes for α, β and γ. splitForwards splits a node representing $\alpha\beta$ into separate nodes for α and β. Note that when a node representing the subsequence $\alpha\beta$ is split into separate nodes for α and β, the overlapping $k-1$ characters occur both at the end of the node for α and at the beginning of the node for β.

Algorithm 2 Construct Compressed de Bruijn Graph from Alignments

```
Input: genome sequence, k, set of self-alignments ≥ k.
Output: compressed forward de Bruijn graph of genome.
for all lines in mummerOutputFile do
    if splitInterval then
                                                  ▷ set align1 and align2 to second part of self-ovlerap
        splitInterval ← false
    else
                                                  ▷ load align1 and align2 from input file
        if self overlapping alignment then
                                                  ▷ split alignment to two parts
                                                  ▷ set align1 and align2 to first part of self-overlap
            splitInterval ← true
        else
            splitInterval ← false
        end if
    end if
    while ! intervalInserted do
        foundPos = findNodeBeginAtPos(align1.beg)
        if foundPos ≢ -1 then
            foundNode ← nodes[foundPos]
            if foundNode.length > alignLength then
                                                  ▷ foundNode is too long
                splitBackwards(foundNode, alignLength)
                intervalInserted ← true
            else if foundNode.length < alignLength then
                                                  ▷ foundNode is too short
                incToNextBegin ← foundNode.length −k +1
                align1.beg+= incToNextBegin
                align2.beg+= incToNextBegin
                intervalInserted ← false
            else
                                                  ▷ first interval is represented by foundNode
                intervalInserted ← true
            end if
        else
                                                  ▷ align1.beg not found, implicitly included in a node
            lastNode ← closest node with start before align1.beg
            if align1.end is end of lastNode then
                foundNode ← splitNodeForward(lastNode, align1.beg)
            else
                foundNode ← splitNodeMiddle(lastNode, align1.beg, align1.length)
            end if
                                                  ▷ foundNode represents align.beg
            createChild(newNode, align.beg)
            intervalInserted ← true
        end if
        addedStart ← addStartPosToNode(foundNode, align2.beg)
        if intervalInserted and addedStart then
            createChild(foundNode, align2.beg)
        end if
    end while
end for
updateLeaves()
```

Algorithm 3 Construct Repeat Nodes from MEM nodes in suffix tree in $O(n\ log\ n)$ time and space

```
1: procedure CREATEREPEATNODESFROMSUFFIXTREE                          ▷ recursive DFS of suffix tree
2:     CREATEREPEATNODESFROMMEM(root)
3: end procedure

4: procedure CREATEREPEATNODESFROMMEM(node)
5:     for all node children do
6:         CREATEREPEATNODESFROMMEM(node.child)
7:     end for
8:     if node.MEM then
9:         if node.parent ≠ root then                                 ▷ include path from root to MEM node
10:            extend node label left to include path label from root
11:        end if
12:        while node.strdepth ≥ k do
13:            LMAnode ← node.LMA
14:            if LMAnode ≠ null then                                 ▷ skip LMAnode.strdepth characters
15:                if skippedChars then
16:                    createRepeatNode for skipped segment of MEM
17:                end if
18:                numCharsToSkip ← LMAnode.strdepth −k + 1
19:            end if
20:            node ← node.suffixSkips[0]
21:            if numCharsToSkip > 0 then                             ▷ use suffix skips to traverse numCharsToSkip suffix links quickly
22:                numCharsToSkip−−
23:                if node.MEM then
24:                    break
25:                end if
26:                while numCharsToSkip > 0 do
27:                    slinkIndex ← floor(log(numCharsToSkip) / log(2))
28:                    slinkTraversing ← pow(2, slinkIndex)
29:                    if node.closestLMA[slinkIndex] ≠ null then
30:                        if node.closestLMAproximity[slinkIndex] < numCharsToSkip then
31:                            adjust numCharsToSkip to extend over skipped LMA
32:                        end if
33:                    end if
34:                    node ← node.suffixSkips[slinkIndex]
35:                    numCharsToSkip − = slinkTraversing
36:                end while
37:            end if
38:        end while
39:        if needLastNode then
40:            createRepeatNode for overhang beyond last embedded MEM
41:        end if
42:    end if
43: end procedure
```

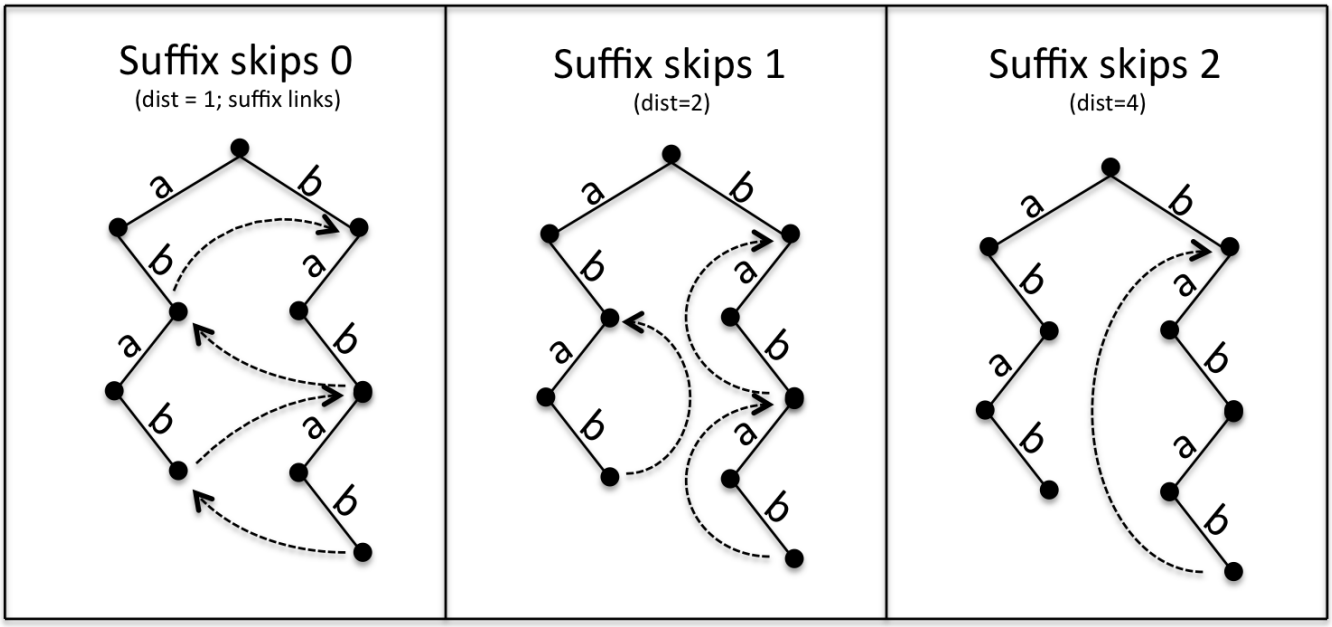

Fig. 2. Example suffix tree and suffix skips for the string "babab$". For clarity, only a subset of the suffix links and skips are displayed. Leaf nodes with $ characters are also not shown.

Table 1. The 9 *B. anthracis* and 9 *E. coli* strains included in our pan-genome analysis.

Strain	Size	Accession
B. anthracis A0248 uid33543	5178 KB	CP001598
B. anthracis A16R uid40353	5179 KB	CP001974
B. anthracis A16 uid40303	5179 KB	CP001970
B. anthracis Ames 0581 uid10784	5178 KB	AE017334
B. anthracis Ames uid309	5178 KB	AE016879
B. anthracis CDC 684 uid31329	5181 KB	CP001215
B. anthracis CI uid36309	5147 KB	CP001746
B. anthracis H9401 uid49361	5170 KB	CP002091
B. anthracis str Sterne uid10878	5180 KB	AE017225
E. coli 0127 H6 E2348 69 uid32571	4919 KB	FM180568
E. coli 042 uid40647	5193 KB	FN554766
E. coli 536 uid16235	4893 KB	CP000247
E. coli 55989 uid33413	5107 KB	CU928145
E. coli ABU 83972 uid38725	5083 KB	CP001671
E. coli APEC O1 uid16718	5034 KB	CP000468
E. coli APEC O78 uid184588	4753 KB	CP004009
E. coli BL21 DE3 uid20713	4516 KB	CP001509
E. coli BL21 DE3 uid28965	4516 KB	AM946981

Table 2. The number of nodes with suffixSkip[i] decreases rapidly. For 9 strains of *B. anthracis*, k-mer lengths of 25, 100 and 1000 bp, the longest MEM is 5227319 bp long.

	i	k=25	k=100	k=1000
B. anthracis	1	40151049	40151049	40151049
	2	30974258	22527962	6987800
	3	30974258	22527962	6987800
	4	30974258	22527962	6987800
	5	29713445	22527962	6987800
	6	25692642	22527962	6987800
	7	20529276	20529276	6987800
	8	15101442	15101442	6987800
	9	10308390	10308390	6987800
	10	6895641	6895641	6895641
	11	5234634	5234634	5234634
	12	4697489	4697489	4697489
	13	4567441	4567441	4567441
	14	4523377	4523377	4523377
	15	4461156	4461156	4461156
	16	4362852	4362852	4362852
	17	4166244	4166244	4166244
	18	3863600	3863600	3863600
	19	3339312	3339312	3339312
	20	2290736	2290736	2290736
	21	193584	193584	193584

Table 3. The number of nodes with suffixSkip[i] decreases rapidly. For 9 strains of *E. coli*, k-mer lengths of 25, 100 and 1000 bp, the longest MEM is 2235388 bp long.

	i	k=25	k=100	k=1000
E. coli	1	43523338	43523338	43523338
	2	36840466	35286760	31589536
	3	36840466	35286760	31589536
	4	36840466	35286760	31589536
	5	36584898	35286760	31589536
	6	35844384	35286760	31589536
	7	34929097	34929097	31589536
	8	33880911	33880911	31589536
	9	32781069	32781069	31589536
	10	31539198	31539198	31539198
	11	29765911	29765911	29765911
	12	26863423	26863423	26863423
	13	22433712	22433712	22433712
	14	16980038	16980038	16980038
	15	12225376	12225376	12225376
	16	9541879	9541879	9541879
	17	8103467	8103467	8103467
	18	6707025	6707025	6707025
	19	5204759	5204759	5204759
	20	4178743	4178743	4178743
	21	3130167	3130167	3130167
	22	5227319	5227319	5227319

Table 4. The 62 available strains of *E. coli* included in our scaling experiments. To highlight the maximum similarity between the genomes, seven of the strains were reverse complemented to be in the same orientation as the others.

Strain	Size	Accession	Orientation
E. coli 0127 H6 E2348 69 uid32571	4919 KB	FM180568	Forward
E. coli 042 uid40647	5193 KB	FN554766	Forward
E. coli 536 uid16235	4893 KB	CP000247	Forward
E. coli 55989 uid33413	5107 KB	CU928145	Forward
E. coli ABU 83972 uid38725	5083 KB	CP001671	Forward
E. coli APEC O1 uid16718	5034 KB	CP000468	Forward
E. coli APEC O78 uid184588	4753 KB	CP004009	Forward
E. coli BL21 DE3 uid20713	4516 KB	CP001509	Forward
E. coli BL21 DE3 uid28965	4516 KB	AM946981	Forward
E. coli BW2952 uid33775	4535 KB	CP001396	Forward
E. coli B REL606 uid18281	4586 KB	CP000819	Forward
E. coli CFT073 uid313	5182 KB	AE014075	Forward
E. coli C ATCC 8739 uid18083	4702 KB	CP000946	Forward
E. coli DH1 uid30031	4587 KB	CP001637	Reverse
E. coli DH1 uid52077	4578 KB	AP012030	Forward
E. coli E24377A uid13960	4933 KB	CP000800	Forward
E. coli ED1a uid33409	5161 KB	CU928162	Forward
E. coli ETEC H10407 uid42749	5105 KB	FN649414	Forward
E. coli HS uid13959	4600 KB	CP000802	Forward
E. coli IAI1 uid33373	4657 KB	CU928160	Forward
E. coli IAI39 uid33411	5084 KB	CU928164	Forward
E. coli IHE3034 uid43693	5060 KB	CP001969	Forward
E. coli JJ1886 uid218163	5082 KB	CP006784	Forward
E. coli KO11FL uid33875	4874 KB	CP002516	Reverse
E. coli KO11FL uid62299	4975 KB	CP002970	Reverse
E. coli K 12 substr DH10B uid20079	4642 KB	CP000948	Forward
E. coli K 12 substr MDS42 uid78215	3939 KB	AP012306	Forward
E. coli K 12 substr MG1655 uid225	4958 KB	U00096	Forward
E. coli K 12 substr W3110 uid16351	4603 KB	AP009048	Forward
E. coli LF82 uid33825	4728 KB	CU651637	Forward
E. coli LY180 uid203308	4790 KB	CP006584	Forward
E. coli NA114 uid66975	4925 KB	CP002797	Forward
E. coli O103 H2 12009 uid32511	5398 KB	P010958	Forward
E. coli O104 H4 2009EL 2050 uid81097	5204 KB	CP003297	Reverse
E. coli O104 H4 2009EL 2071 uid81099	5263 KB	CP003301	Reverse
E. coli O104 H4 2011C 3493 uid81095	5224 KB	CP003289	Reverse
E. coli O111 H 11128 uid32513	5321 KB	AP010960	Forward
E. coli O157H7 EDL933 uid259	5477 KB	AE005174	Forward
E. coli O157H7 uid226	5447 KB	BA000007	Forward
E. coli O157 H7 EC4115 uid27739	5520	CP001164	Forward
E. coli O157 H7 TW14359 uid30045	5476 KB	CP001368	Forward
E. coli O26 H11 11368 uid32509	5644 KB	AP010953	Forward
E. coli O55 H7 CB9615 uid42729	5336 KB	CP001846	Forward
E. coli O55 H7 RM12579 uid68245	5215 KB	CP003109	Forward
E. coli O7 K1 CE10 uid63597	5264 KB	CP003034	Forward
E. coli O83 H1 NRG 857C uid41221	4703 KB	CP001855	Forward
E. coli P12b uid59455	4889 KB	CP002291	Forward
E. coli S88 uid33375	4985 KB	CU928161	Forward
E. coli SE11 uid18057	4842 KB	AP009240	Forward
E. coli SE15 uid19053	4673 KB	AP009378	Forward
E. coli SMS 3 5 uid19469	5021 KB	CP000970	Forward
E. coli UM146 uid50883	4946 KB	CP002167	Reverse
E. coli UMN026 uid33415	5253 KB	CU928163	Forward
E. coli UMNK88 uid42137	5138 KB	CP002729	Forward
E. coli UTI89 uid16259	5018 KB	CP000243	Forward
E. coli W uid48011	4855 KB	CP002185	Forward
E. coli W uid62301	4852 KB	CP002967	Forward
E. coli Xuzhou21 uid45823	5336 KB	CP001925	Forward
E. coli BL21 Gold DE3 pLysS AG uid30681	4528 KB	CP001665	Reverse
E. coli clone D i14 uid52023	4991 KB	P002212	Forward
E. coli clone D i2 uid52021	4991 KB	CP002211	Forward
E. coli c321D uid215084	4600 KB	CP006698	Forward

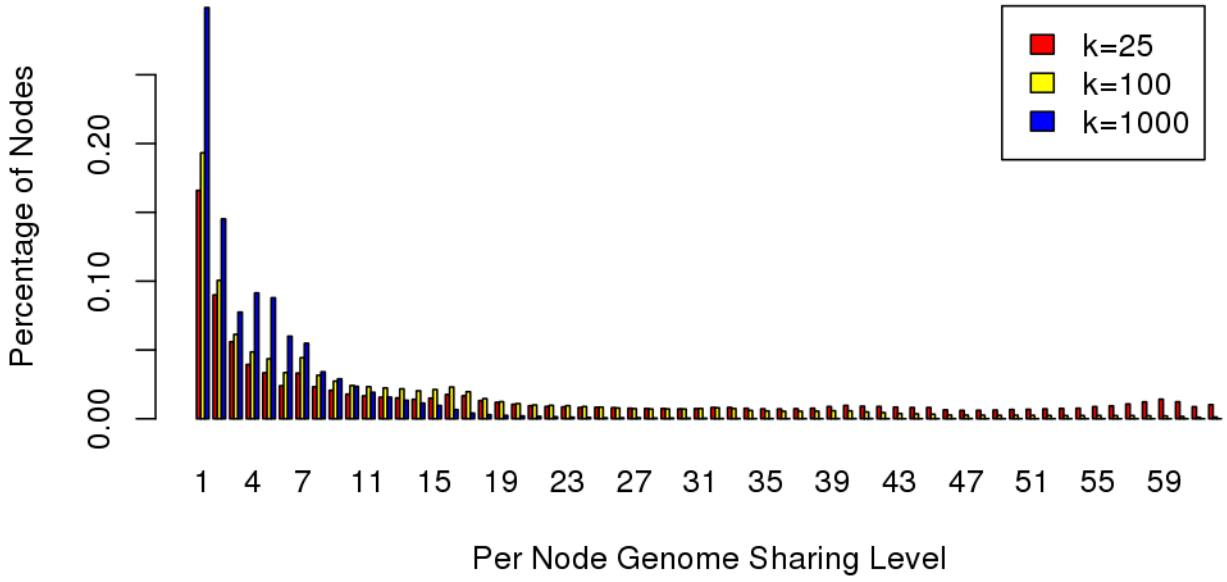

Fig. 3. Levels of genome sharing in the nodes of the pan-genome graph of all 62 strains of *E. coli*. The distribution is approximately exponential in shape, although with an extended tail of highly conserved sequences.

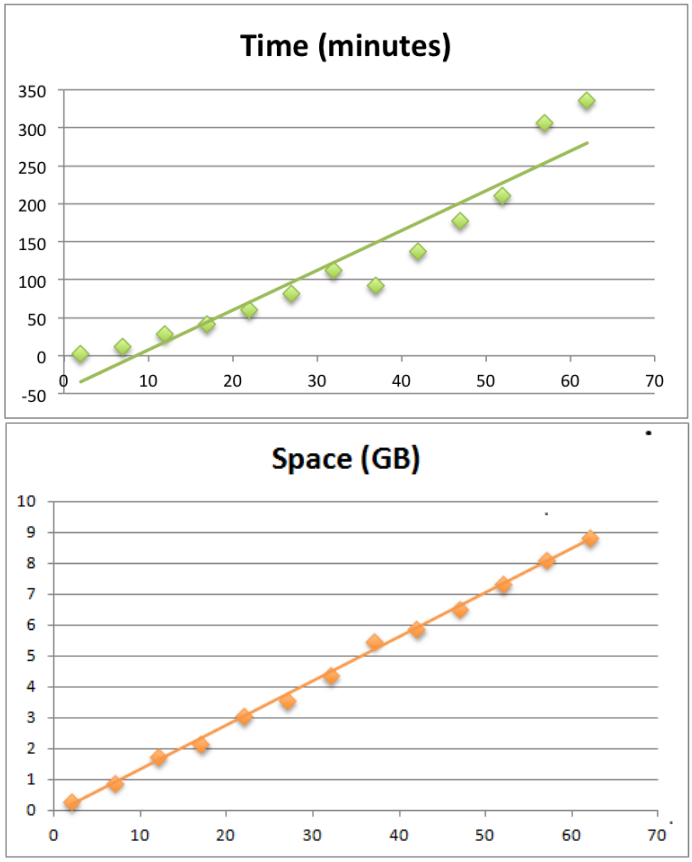

Fig. 4. The running time and peak memory of Sibelia (Minkin *et al.*, 2013) on the pan-genome graphs of increasing numbers of *E. coli* strains. Each point represents the minimum value recorded over 5 trials to reduce measurement noise introduced by competing activity of the server. The line represents the linear regression of the points. Following the recommended settings, we used commands of the form `sibelia.py -s loose ecoli.XXXstrains.fa` where `ecoli.XXXstrains.fa` was a multifasta file containing the selected `XXX` genomes.

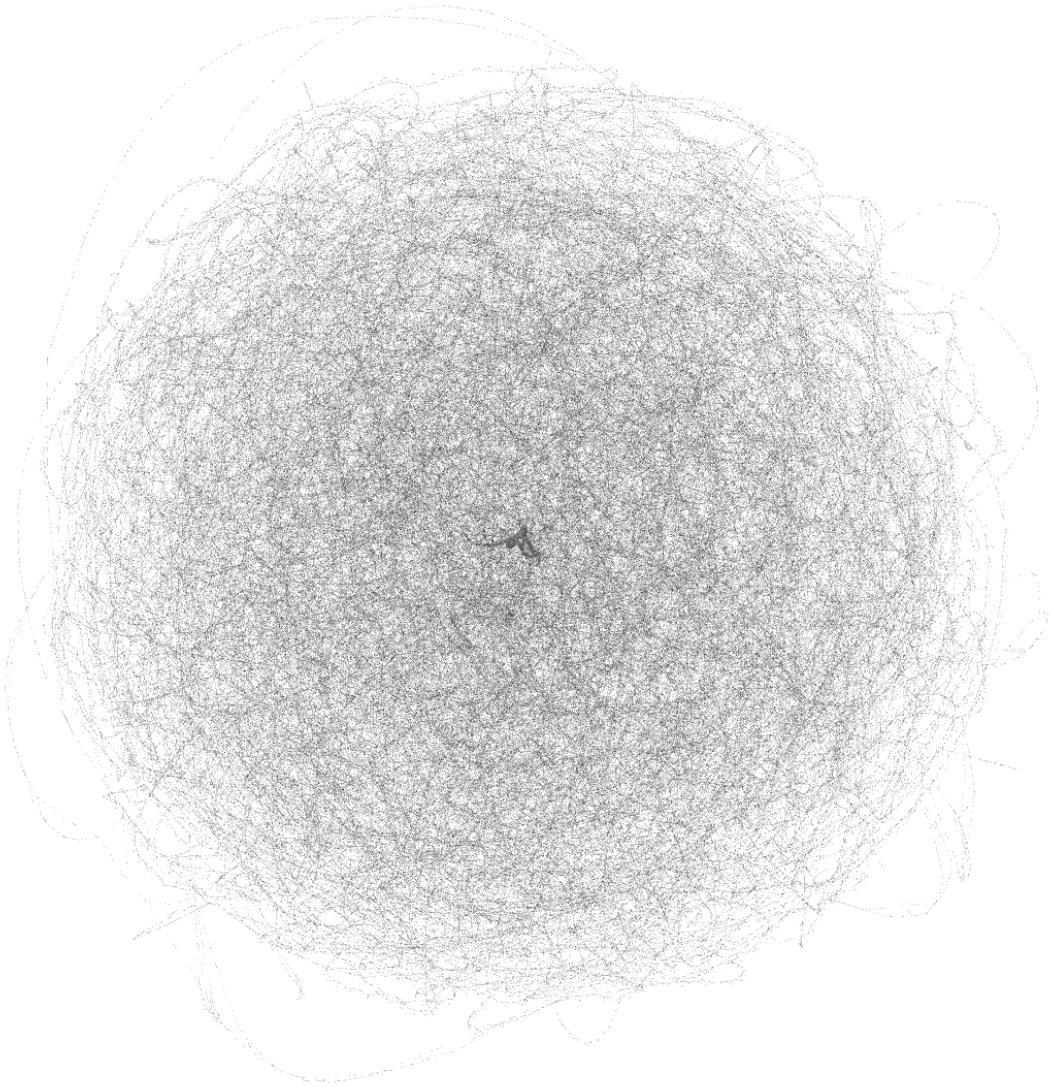

Fig. 5. The compressed de Bruijn graph for the B. anthracis pan genome with k=25 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

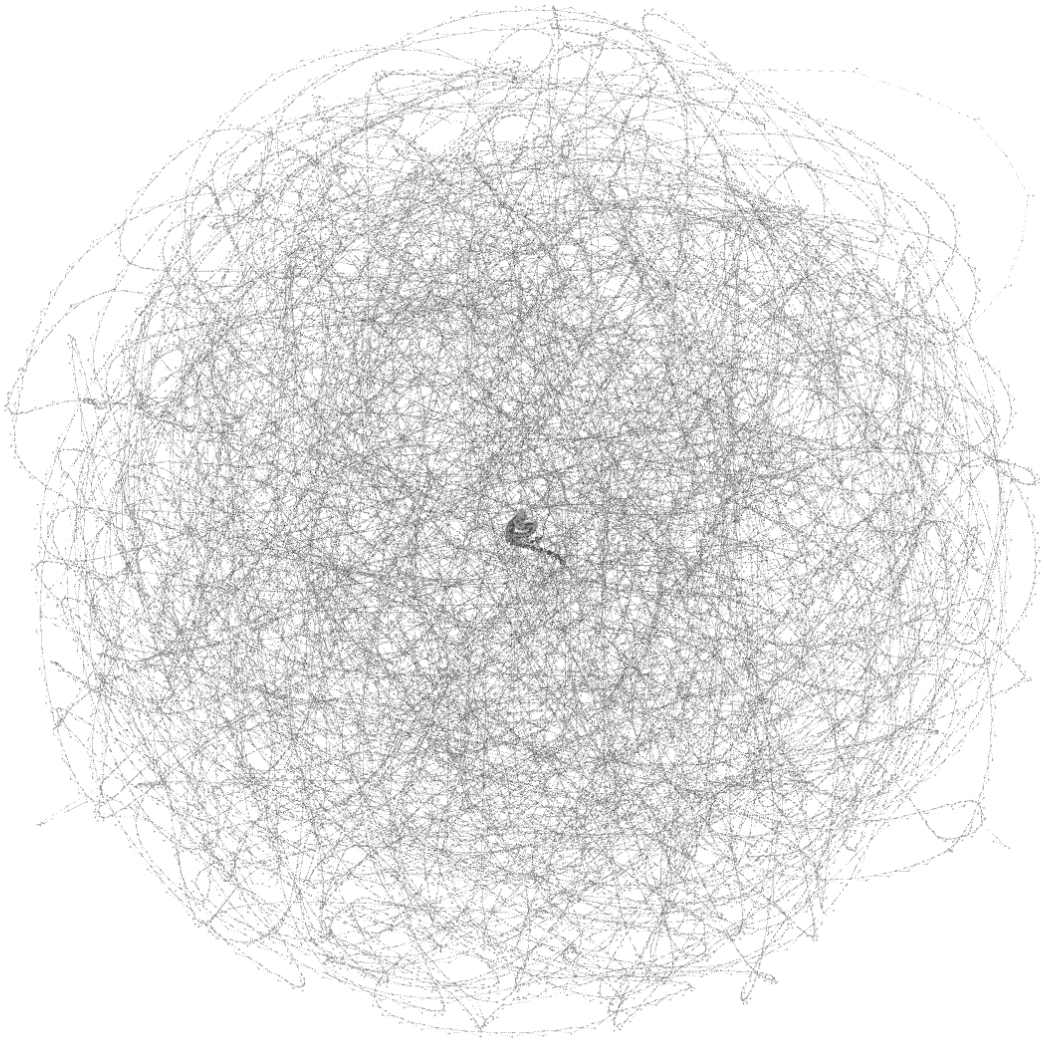

Fig. 6. The compressed de Bruijn graph for the B. anthracis pan genome with k=100 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

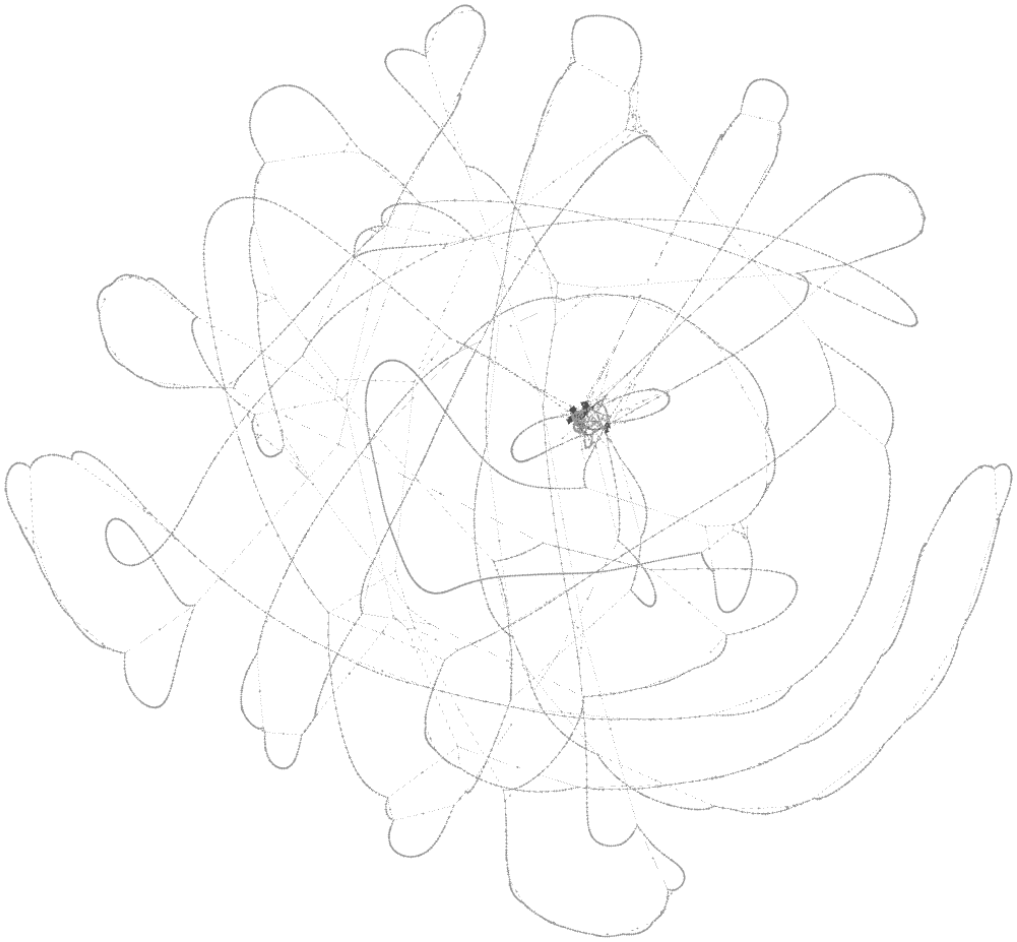

Fig. 7. The compressed de Bruijn graph for the B. anthracis pan genome with k=1000 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

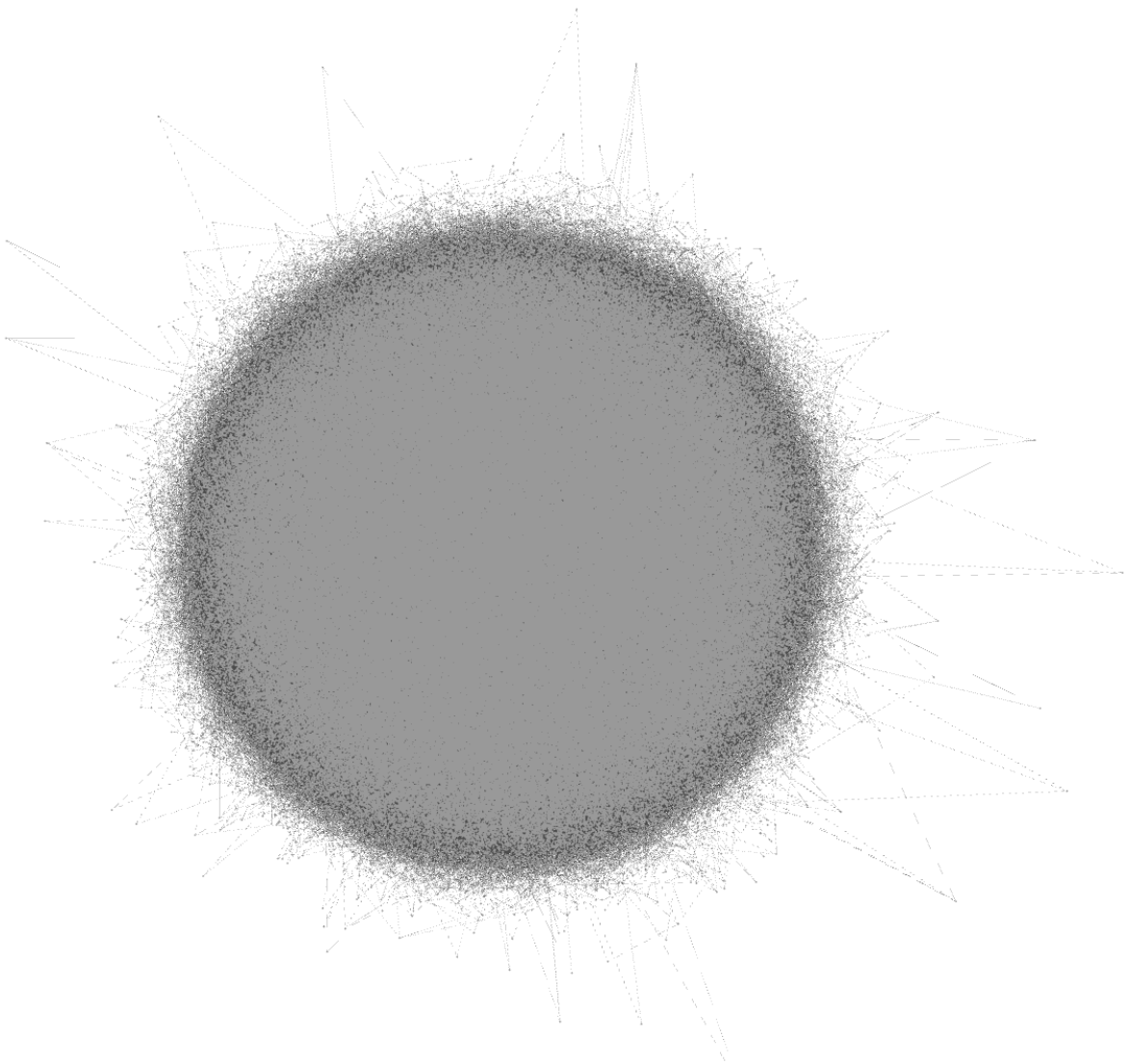

Fig. 8. The compressed de Bruijn graph for the E. coli pan genome with k=25 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

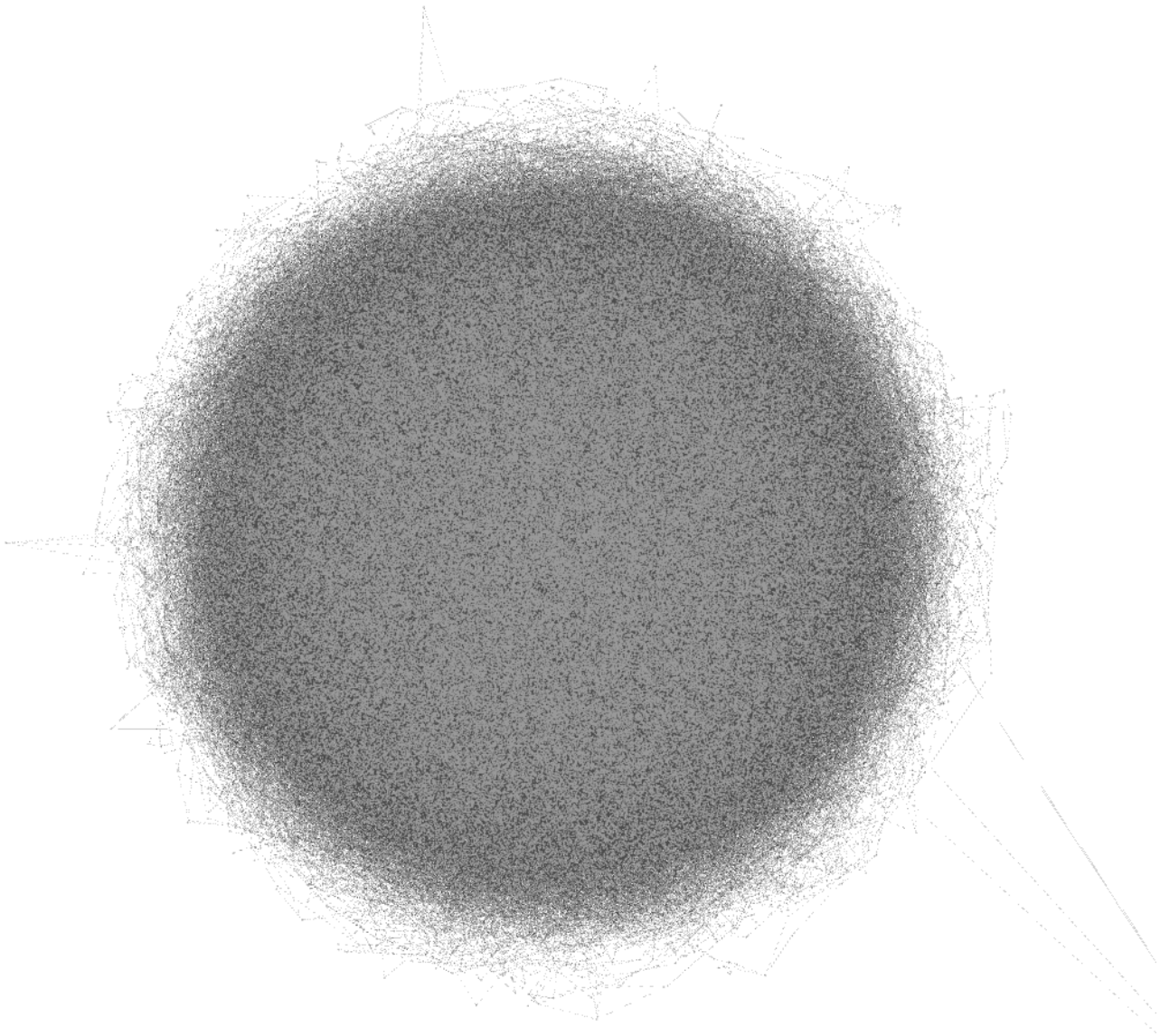

Fig. 9. The compressed de Bruijn graph for the E. coli pan genome with k=100 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

Fig. 10. The compressed de Bruijn graph for the E. coli pan genome with k=1000 artistically rendered in Gephi using the ForceAtlas 2 placement algorithm.

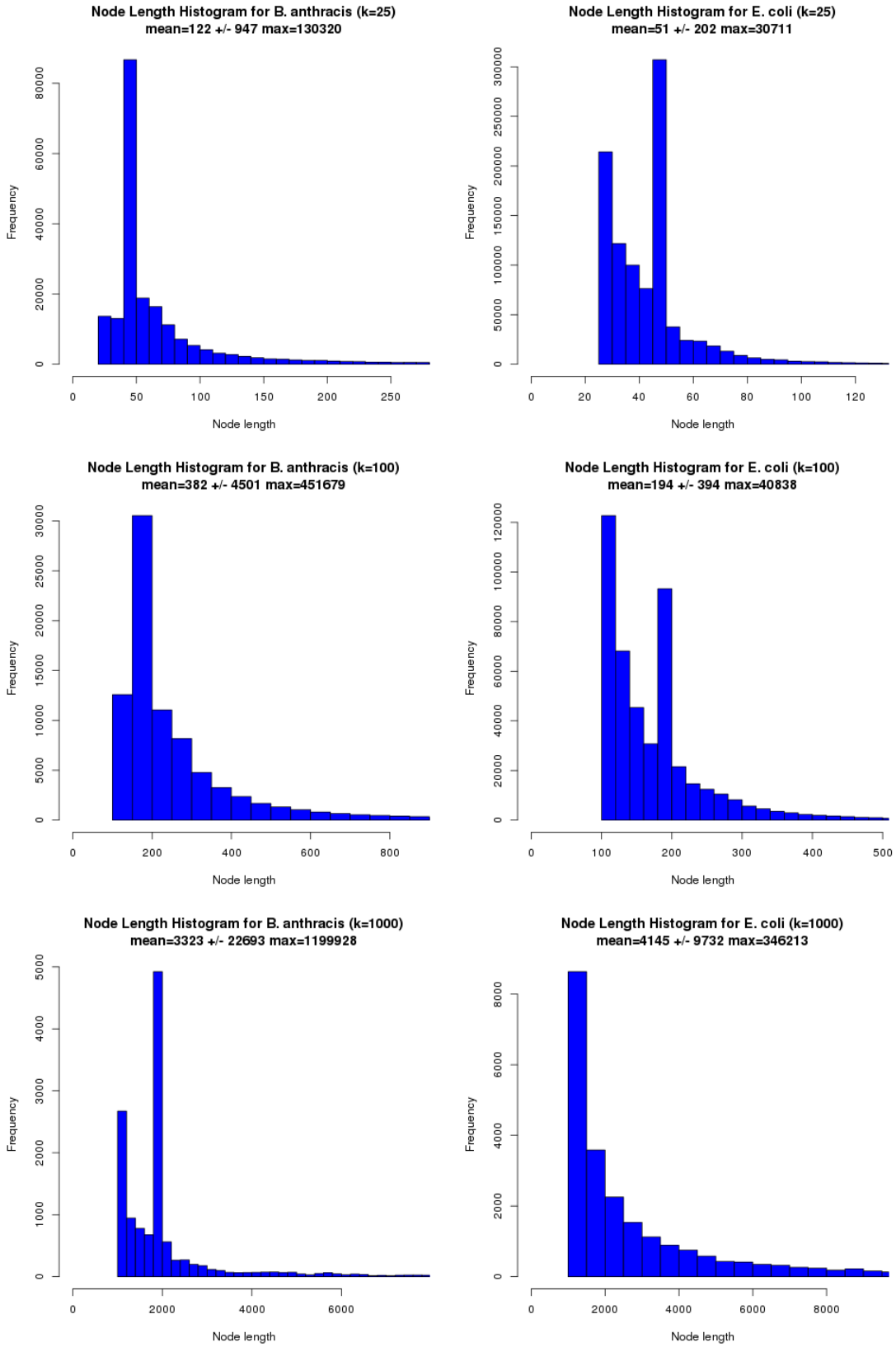

Fig. 11. Distributions of node lengths in the compressed de Bruijn graphs for the pan-genomes of 9 strains of *E. coli* and 9 strains of *B. anthracis*.

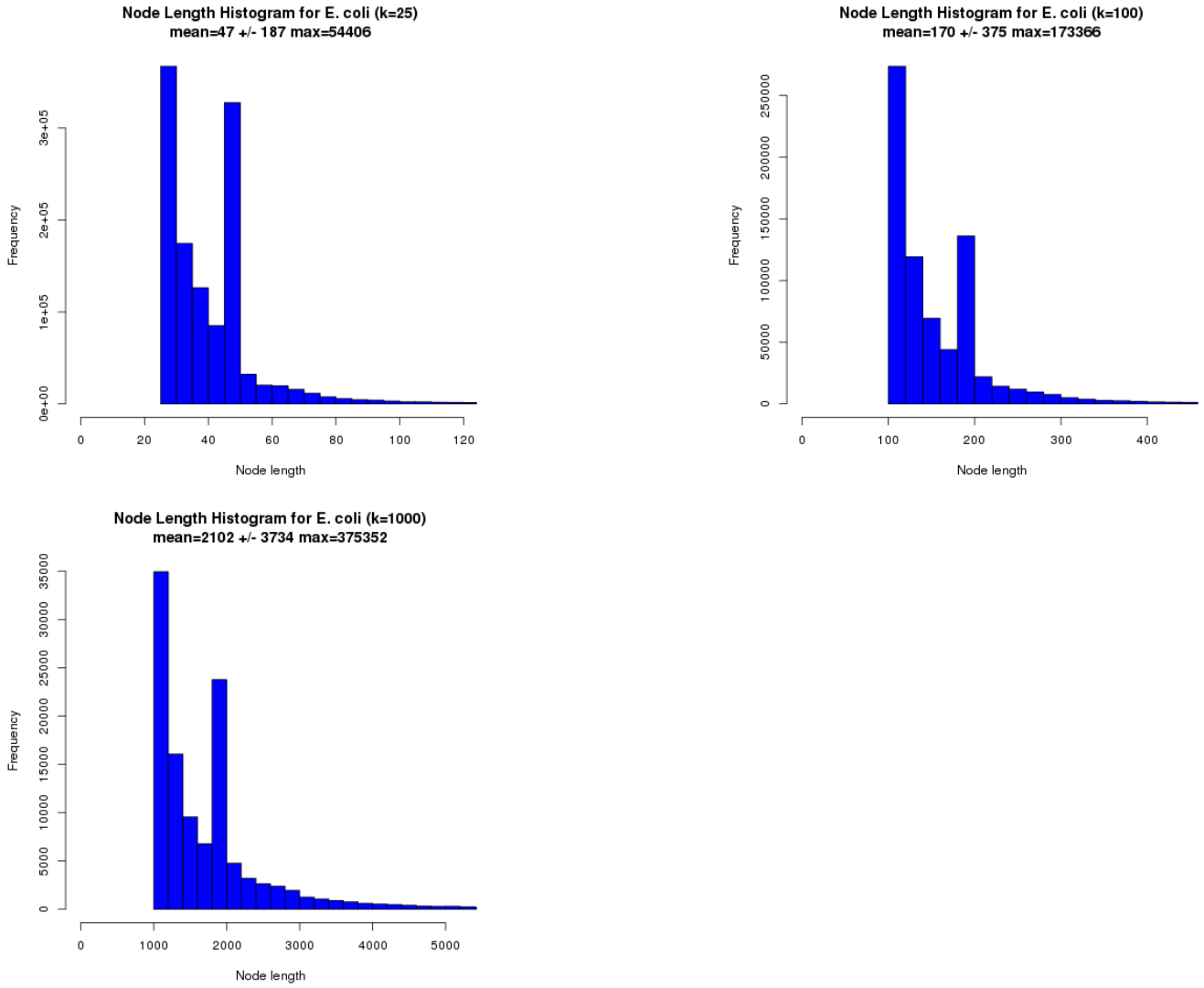

Fig. 12. Distributions of node lengths in the compressed de Bruijn graphs for the pan-genomes of all 62 strains of *E. coli.*

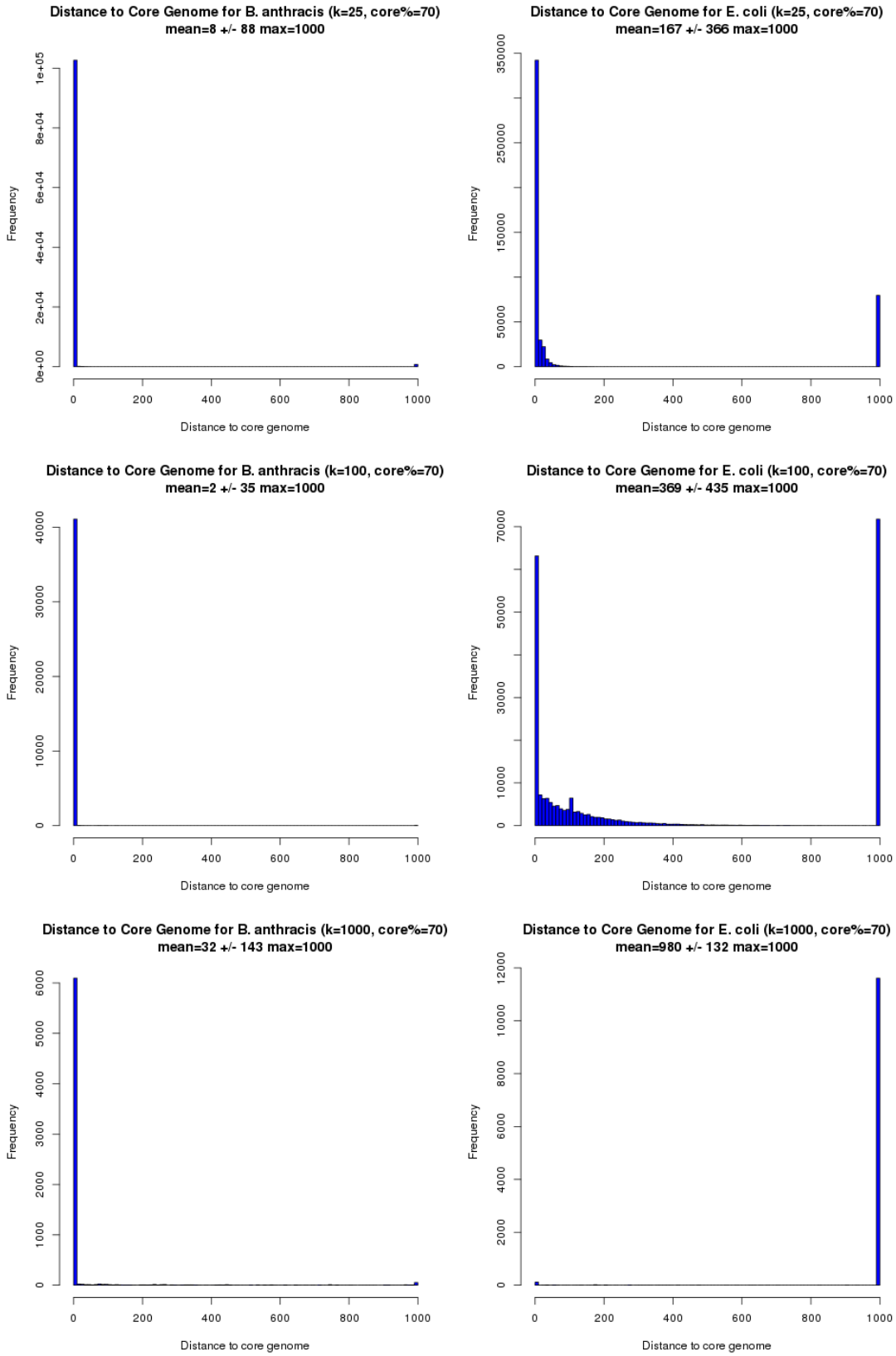

Fig. 13. Distributions of distances to the core genome in the compressed de Bruijn graphs for the pan-genomes of 9 strains of *E. coli* and 9 strains of *B. anthracis.*