Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category
OXFORD

Genome Analysis

Faucet: streaming de novo assembly graph construction

Roye Rozov [1], Gil Goldshlager [2], Eran Halperin [3]*, Ron Shamir [1]*

[1] Blavatnik School of Computer Science, Tel-Aviv University, Tel Aviv, Israel
[2] Department of Mathematics, MIT
[3] Departments of Computer Science, Anesthesiology and Perioperative Medicine, UCLA

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: We present Faucet, a 2-pass streaming algorithm for assembly graph construction. Faucet builds an assembly graph incrementally as each read is processed. Thus, reads need not be stored locally, as they can be processed while downloading data and then discarded. We demonstrate this functionality by performing streaming graph assembly of publicly available data, and observe that the ratio of disk use to raw data size decreases as coverage is increased.

Results: Faucet pairs the de Bruijn graph obtained from the reads with additional meta-data derived from them. We show these metadata - coverage counts collected at junction k-mers and connections bridging between junction pairs - contain most salient information needed for assembly, and demonstrate they enable cleaning of metagenome assembly graphs, greatly improving contiguity while maintaining accuracy. We compared Faucet's resource use and assembly quality to state of the art metagenome assemblers, as well as leading resource-efficient genome assemblers. Faucet used orders of magnitude less time and disk space than the specialized metagenome assemblers MetaSPAdes and Megahit, while also improving on their memory use; this broadly matched performance of other assemblers optimizing resource efficiency - namely, Minia and LightAssembler. However, on metagenomes tested, Faucet's outputs had 14-110% higher mean NGA50 lengths compared to Minia, and 2-11-*fold* higher mean NGA50 lengths compared to LightAssembler, the only other streaming assembler available.

Availability: Faucet is available at https://github.com/Shamir-Lab/Faucet

Contact: rshamir@tau.ac.il,eranhalperin@gmail.com

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Assembly graphs encode relationships among sequences from a common source: they capture sequences as well as the overlaps observed among them. When assembly graphs are indexed, their sequence contents can be queried without iterating over every sequence in the input. This functionality makes graph and index construction a prerequisite for many applications. Among these are different types of assembly - e.g., de novo assembly of whole genomes, transcripts, plasmids, etc. [1, 2] - and downstream applications - e.g., mapping reads to the graphs, variant calling, pangenome analysis, etc. [3, 4]

In recent years, much effort has been expended to reduce the amount of memory used for constructing assembly graphs and indexing them. Major advances often relied on index structures that saved memory by enabling subsets of possible queries: e.g., one could query what extensions a given substring s has, but not how many times s was seen in the input data. A great deal of success ensued in reducing the amount of memory needed to efficiently construct the central data structures used by most de novo assembly algorithms, namely, the de Bruijn and string graphs [5, 6, 7, 8]. Furthermore, efficient conversion of de Bruijn graphs to their *compacted* form (essentially string graphs with fixed overlap size) has been demonstrated [9, 10, 11].

In parallel to these efforts, streaming approaches were demonstrated as alternative resource-efficient means of performing analyses that had typically relied on static indices. Although appealing in terms of speed and low memory use, these approaches were initially demonstrated primarily for counting-centered applications such as estimating k-mer frequencies, error-correction of reads, and quantification of transcripts [12, 13, 14, 15, 16].

Recently, a first step towards bridging the gap between streaming approaches and those based on static index construction was taken, hinting at the potential benefits of combining the two. Matwali et al.[17] demonstrated a streaming approach to assembly by making two passes on a set of reads. The first pass subsamples k-mers in the de Bruijn graph and inserts them into a Bloom filter, and the second uses this Bloom filter to identify 'solid' (likely correct) k-mers, which are then inserted into a second Bloom filter. This streaming approach resulted in very high resource efficiency in terms of memory and disk use. However, LightAssembler finds solid k-mers while disregarding paired-end and coverage information, and thus is limited in its ability to resolve repeats and to differentiate between different possible extensions in order to improve contiguity.

In this work, we extend this approach with the aim of providing a more complete alternative to downloading and storing reads for the sake of de novo assembly. We show this is achievable via online graph and index construction. We describe the Faucet algorithm, composed of an online phase and an offline phase. During the online phase, two passes are made on the reads without storing them locally to first load their k-mers into a Bloom filter, and then identify and record structural characteristics of the graph and associated metadata essential for achieving high contiguity in assembly. The offline phase uses all of this information together to iteratively clean and refine the graph structure.

We show that Faucet requires less disk space than the input data, in contrast with extant assemblers that require storing reads and often produce intermediate files that are larger than the input. We also show that the ratio of disk space Faucet uses to the input data improves with higher coverage levels by streaming successively larger subsets of a high coverage human genome sample. Furthermore, we introduce a new cleaning step called *disentanglement* enabled by storage of paired junction extensions in two Bloom filters - one meant for pairings inside a read, and one meant for junctions on separate paired end mates. We show the benefit of disentanglement via extensive experiments. Finally, we compared Faucet's resource use and assembly quality to state of the art metagenome assemblers, as well as leading resource-efficient genome assemblers. Faucet used orders of magnitude less time and disk space than the specialized metagenome assemblers MetaSPAdes and Megahit, while also improving on their memory use; this broadly matched performance of other assemblers optimizing resource efficiency - namely, Minia and LightAssembler. However, on metagenomes tested, Faucet's outputs had 14-110% higher mean NGA50 lengths compared to Minia, and 2-11-*fold* higher mean NGA50 lengths compared to LightAssembler, the only other streaming assembler available.

2 Preliminaries

For a string s, we denote by $s[i]$ the character at position i, $s[i:j]$ the substring of s from position i to j (inclusive of both ends), and $|s|$ the length of s. Let $pref(s,j)$ be the prefix comprised of the first j characters of s and $suff(s,j)$ be the suffix comprised of the last j characters of s. We denote concatenation of strings s and t by $s \circ t$, and the reverse complement of a string s by s'.

A *k-mer* is a string of length k drawn from the DNA alphabet $\Sigma = \{A, C, G, T\}$. The de Bruijn graph $G(S,k) = (V,E)$ of a set of sequences S has nodes defined by consecutive k-mers in the sequences, $V = \bigcup_{s \in S} \bigcup_{i=0}^{|s|-k+1} s[i : i+k-1]$; E is the set of arcs defined by $(k-1)-mer$ overlaps between nodes in V. Namely, identifying vertices with their k-mers, $(u,v) \in E \iff suff(u,k-1) = pref(v,k-1)$. Each node v is identified with its reverse complement v', making the graph G bidirected, in that edges may represent overlaps between either orientation of each node [18]. When necessary, our explicit representation of nodes will use *canonical* node naming, i.e., the name of node (v,v') will be the lexicographically lesser of v and v'. *Junction nodes* are defined as k-mers having in-degree or out-degree greater than 1. *Terminal nodes* are k-mers having out-degree 1 and in-degree 0 or in-degree 1 and out-degree 0. Terminals and junctions are collectively referred to as *special nodes*. The *compacted de Bruijn graph* is obtained from a de Bruijn graph by merging all adjacent *non-branching nodes* (i.e., those having in-degree and out-degree of exactly 1). The string associated with merged adjacent nodes is the first k-mer, concatenated with the single character extensions of all following non-branching k-mers. Such merged non-branching paths are called *unitigs*.

Since a junction v having in-degree greater than 1 and out-degree 1 is identified with v' having out-degree greater than 1 and in-degree 1, we speak of junction directions relative to the reading direction of the junction's k-mer. Therefore, a *forward junction* has out-degree greater than 1, and a *back junction* has in-degree greater than 1. We refer to outbound k-mers beginning paths in the direction having out-degree greater than 1 as *heads*, and the sole outbound k-mer in the opposite direction as the junction's *tail*. It is possible that a junction may have no tail.

A Bloom filter B is a space-efficient probabilistic hash table enabling insertion and approximate membership query operations [19]. The filter consists of a bit array of size m, and an element x is inserted to B by applying h hash functions, $f_0, \ldots, f_{h-1}$ such that $\forall_{i \in [0,h-1]} f_i(x) \in [0, m-1]$, and setting values of the filter to 1 at the positions returned. For a Bloom filter B and string s, by $s \in B$ or the term 's in B' we refer to $B[s] = 1$, i.e., when the h hash functions used to load B are applied to s, only 1 values are returned. Similarly, $s \notin B$ or 's not in B' means that at least one of the h hash functions of B returned 0 when applied to s. For any s that has been inserted to B, $B[s] = 1$ by definition (i.e., there are no false negatives). However, false positives are possible, with a probability that can tuned by adjusting m or h appropriately.

3 Methods

We developed an algorithm called Faucet for streaming de novo assembly graph construction. A bird's eye view of its entire work-flow is provided in Figure 1. Below we detail individual steps.

Online Bloom filter loading Faucet begins by loading two Bloom filters, B_1 and B_2, as it iterates through the reads, using the following procedure: all k-mers are inserted to B_1, and only k-mers already in B_1 (i.e., those for which all hash queries return 1 from B_1) are inserted to B_2. Namely, for each k-mer s, if $B_1[s] = 1$ then we insert s into B_2, otherwise we insert into B_1. After iterating through all reads, B_1 is discarded and only B_2 is used for later stages. This procedure imposes a coverage threshold on the vast majority of k-mers so that primarily 'solid k-mers' [20] observed at least twice are kept. This process is depicted in Round 1 of Figure 1A. We note that a small proportion of singleton or false positive k-mers may evade this filtration. No count information is associated with k-mers at this round.

Online graph construction B_2, loaded at the first round, enables Faucet to query possible forward extensions of each k-mer. Faucet iterates through all reads a second time to collect information necessary for avoiding false positive extensions, building the compacted de Bruijn

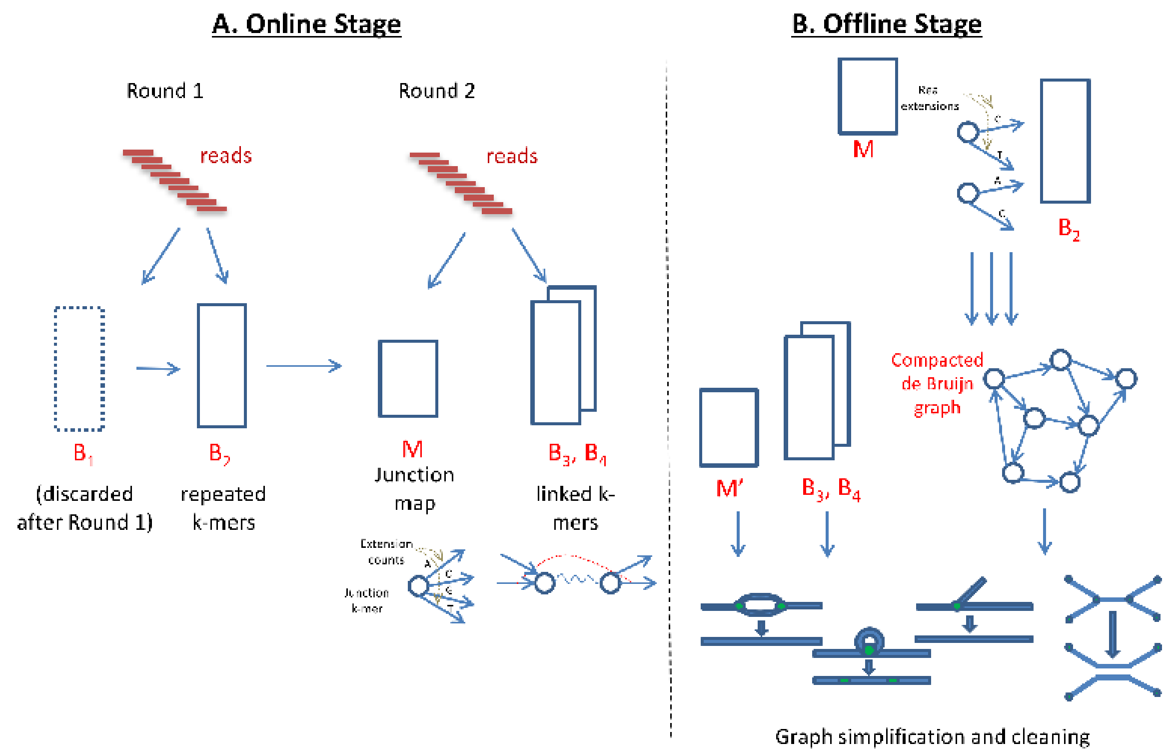

Fig. 1: Faucet work-flow. A. The online stage involves a first round of processing all reads in order to load Bloom filters B_1 and B_2, and a second round in order to build the junction map M and load additional Bloom filters B_3 and B_4. M stores the set of all junctions and extension counts for each junction, while B_3 and B_4 capture connections between junction pairs. The two online rounds capture information from and perform processing on each read, and the processing performed always depends on the current state of data structures being loaded. B. The offline stage uses B_2 and M, constructed during the online stage, in order to build the compacted de Bruijn graph by extending between special nodes using Bloom filter queries. ContigNodes (not shown) take the place of junctions and are stored in M', allowing access (via stored pointers) to Contigs out of each junction, and coverage information. An additional vector of coverage values at fake or past junctions is also maintained for each Contig. Then, B_3, B_4, and this coverage information are used together to perform simplifications on and cleaning of the graph.

graph, and later, cleaning the graph. The second round consists of finding junctions and terminal k-mers, recording their true extension counts, and recording k-mer pairs (Round 2 of Figure 1A).

Faucet's Online stage has one main routine - Algorithm 1 - that calls upon two subroutines - Algorithm 2 and Algorithm 3. First, junction k-mers and their start positions are derived from a call to Algorithm 2. To find junctions, Algorithm 2 makes all possible alternate extension queries (Line 3-Line 4) to B_2 for each k-mer in the read sequence r. A junction k-mer j may have multiple extensions in B_2 - either because there are multiple extensions of j in G that are all real (i.e., present on some read), or because there is at least one real extension in G and some others in B_2 that are false positives. Accordingly, each k-mer possessing at least one extension that differs from the next base on the read is identified as a junction. Whenever one is found, its sequence along with its start position are recorded (Line 4), and the list of such tuples is returned. We note that each k-mer in the read is also queried for junctions in the reverse complement direction, but this is not shown in Algorithm 2.

Algorithm 1 then uses this set of junctions to perform accounting (Line 4-Line 7). All junctions are inserted into a hash map M that maps junction k-mers to vectors maintaining counts for each extension. For each junction of r, a count of 0 is initialized for each possible extension. These counters are only incremented based on extensions observed on reads - i.e., extensions due to Bloom filter outputs alone are not counted. As every real extension out of each junction must be observed on some read, and we scan the entire set of reads, an extension will have non-zero count only if it is real. This mechanism allows Faucet to maintain

Algorithm 1 $scanReads(R, B_2)$

Input: read set R, Bloom filter B_2 loaded from round 1, an empty Bloom filter B_3

Output: 1. a junction Map M comprised of $(key, value)$ pairs. Each key is a junction k-mer, and each $value \in \mathbb{N}^4$ is a vector $[c_A, c_C, c_G, c_T]$ of counts representing the number of times each possible extension of key was observed in R; 2. B_3 is loaded with linked k-mer pairs (i.e., specific 2k-mers - see text - are hashed in).

1: $M \leftarrow \emptyset$
2: **for** $r \in R$ **do**
3: $\quad juncs \leftarrow findJunctions(r, B_2)$ ▷ call to Algorithm 2
4: $\quad$ **for** $(junc, pos) \in juncs$ **do**
5: $\quad\quad$ **if** $junc \notin M$ **then**
6: $\quad\quad\quad M[junc] \leftarrow [0, 0, 0, 0]$
7: $\quad\quad$ *increment counter in M for* $r[pos + k]$
$\quad recordPairs(r, juncs, B_3)$ ▷ call to Algorithm 3
8: **return** M, B_3

coverage counts for all real extensions out of junctions. In later stages, only extensions having non-zero counts will be visited, but counts are stored for real extensions of false junctions as well. These latter counts are used to sample coverage distributions on unitig sequences at more points than just their ends. Proportions of real junctions vs. the totals stored after accounting are described in the section 'Solid junction counts' in the Appendix.

Algorithm 2 $findJunctions(r, B_2)$

Input: read r and Bloom filter B_2

Output: juncTuples, a list of tuples (seq, p) , where p is the start position of junction k-mer seq in r, in order of appearance on r

```
juncTuples ← ∅
for i ∈ [0, |r| − k] do kmer ← r[i : i + k − 1]
    for c ∈ Σ \ {r[i + k]} do
        if (suff(kmer, k − 1) ∘ c ∈ B2) then
            juncTuples ← juncTuples ∪ (kmer, i)
return juncTuples
```

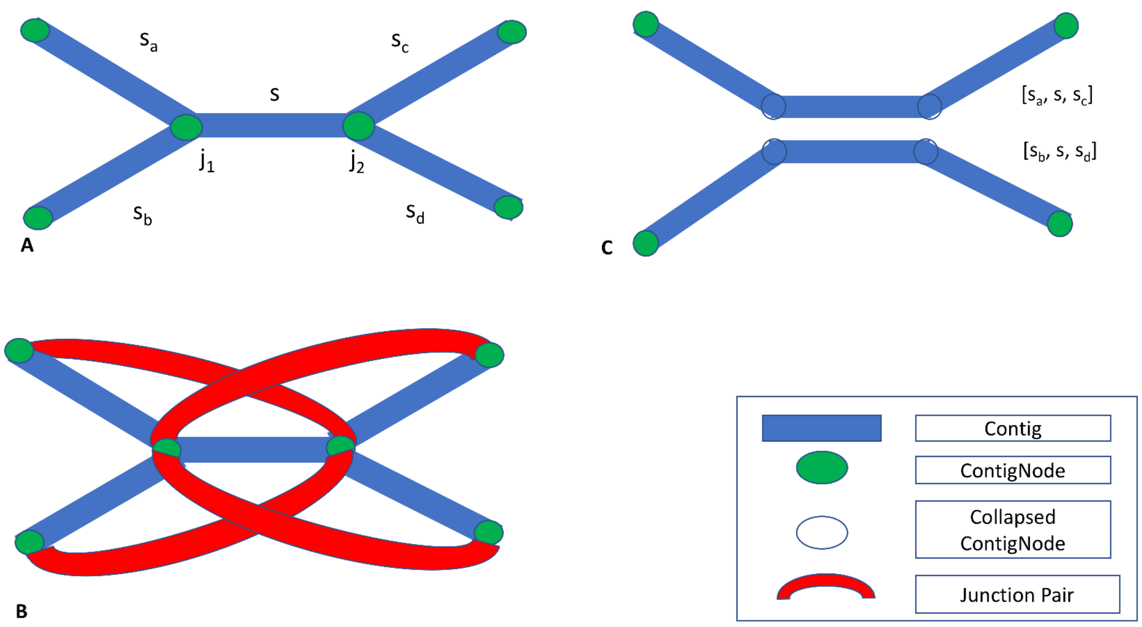

Fig. 2: Disentanglement. A. A *tangle* characterized by two opposite facing junctions j_1 and j_2, each with out-degree 2. B. Junction pairs linking extensions on s_a with s_c and s_b with s_d. Since no pairs link extensions on s_a with s_d or s_b with s_c, only one orientation is supported. C. the result of disentanglement: paths $[s_a, s, s_c]$ and $[s_b, s, s_d]$ are each merged into individual sequences, and junctions j_1 and j_2 are removed from M.

Following the accounting performed on observed junctions, Faucet records adjacencies between pairs of junctions using additional Bloom filters - B_3 and B_4. These adjacencies are needed for disentanglement - a cleaning step applied in Faucet's offline stage. Disentanglement, depicted in Figure 2, is a means of repeat resolution. Its purpose is to split paths that have been merged due to the presence of a shared segment - the repeat - in both paths. In order to 'disentangle,' or resolve the tangled region into its underlying latent paths, we seek to store sequences that flank opposite ends of the the repeat. Pairs of heads observed on reads provide a means of 'reading out' such latent paths by indicating which heads co-occur on sequenced DNA fragments. The application of disentanglement is presented in the section 'Offline graph simplification and cleaning,' while we now focus on the mechanism of pair collection and its rationale. To capture short and long range information separately, Bloom filter B_3 holds head pairs on the same read, while B_4 holds heads chosen such that each head is on a different mate of a paired-end read. Algorithm 3 is the process by which pairs are inserted into B_3, and insertion into B_4 is described in the Appendix.

In Algorithm 3, we aim to pair heads that are maximally informative. Informative pairs are those that allow us to 'read out' pairs of unitigs that belong to the same latent path. We specifically choose to insert heads because during the offline stage when disentanglement takes place, adjacencies between each unitig starting at an edge to a head and the unitig starting at the edge from the junction to its tail of are known and accessible via pointers to their sequences. Therefore, extension pairs capturing information of direct adjacencies provide no new information. The closest indirect adjacency that may be informative when captured from a read is that between two junctions that either face in the same direction, or when the first faces back and the second faces forward, as shown in Figure 3 A. Thus, when there are only two junctions on a read, their pair of heads is inserted as long as the two junctions are not facing each other. When there are at least three junctions on a read, every other junction out of every consecutive triplet is paired, as shown for a single triplet in Figure 3 B. This figure demonstrates that selecting every other head is preferable to selecting consecutive heads out of a triplet. This type of insertion is executed in Line 1-Line 5 ofAlgorithm 3 and ensures all unitigs flanking some triplet are potentially inferable. For reads having more than three junctions, applying the triplet rule for every consecutive window of size 3 similarly allows for all unitigs on the read to be included in some hashed pair.

Offline graph simplification and cleaning Given B_2, B_3, B_4 and M resulting from the online stage, the compacted de Bruijn graph is generated by traversing each forward extension out of every special k-mer, as well as traversing backwards in the reverse complement direction when the node has not been reached before by a traversal starting from another node. This is done by querying B_2 for extensions and continuing until the next

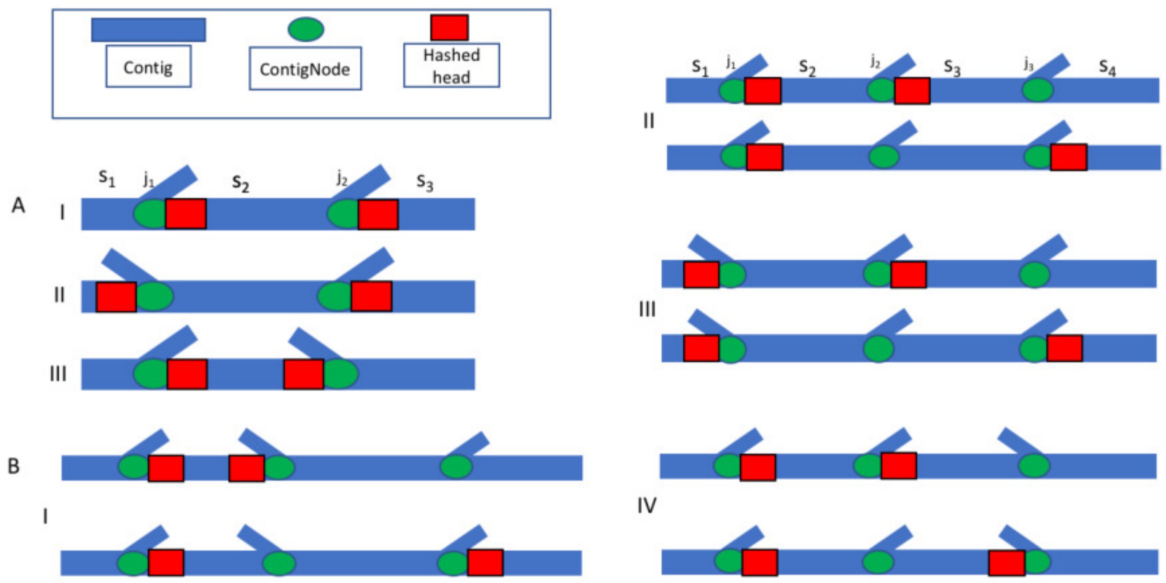

Fig. 3: Rationale for B_3 insertions. Pairs of junction heads (indicated by red rectangles adjacent to green junctions) observed on reads are inserted when they provide additional information to infer a path on the graph. A. Two junctions observed on a read. In cases I and II, it is beneficial to insert the pairs of heads into B_3, as in both cases each individual head allows inference of a different Contig. In case III, inserting the pair between two junctions facing each other is not beneficial because both heads lie on opposite ends of the same Contig. In all three cases, the full path can be inferred. B. Four possible arrangements of three consecutive junctions on a read. There are four more that are symmetrical reflections of those presented that are not shown. In each case, we compare the Contigs covered (i.e., either included by some head or inferable as a junction's back) when heads out of consecutive (top) and non-consecutive (bottom) junctions are chosen, assuming only one pair is inserted. Note that in cases I-III, Contig s_4 is not covered by any paired head or tail when inserting consecutive heads, while in case IV, all Contigs are covered by either the paired heads or some tail. Thus, in the first three cases it will not be possible to determine which head of j_3 occurred with an extension of j_1 or j_2 on some read unless this information is provided by some other hashed pair. In contrast, when non-consecutive heads are paired, every Contig is covered by either one of the inserted heads or a tail.

Algorithm 3 $recordPairs(r, juncs, B_3)$

Input: read r, $juncs$ - a list of pairs (j, p) , where p is the start position of junction j in r, and Bloom filter B_3. We also make use of a subroutine $getOutExt(j_i, p_i, r)$ that for a junction j_i returns $pref(j_i, k-1) \circ r[p_i - k]$ if j_i is a back junction, and $suff(j_i, k-1) \circ r[p_i + k]$ otherwise.
Output: Bloom filter B_3, loaded with select *linked k-mer pairs*

```
if len(juncs) > 2 then
    for i ∈ [0, len(juncs) − 2] do
        back ← getOutExt(j_i, p_i, r)
        front ← getOutExt(j_{i+2}, p_{i+2}, r)
        insert(back ∘ front, B_3)          ▷ insert the concatenation into B_3
else if (len(juncs) = 2) ∧ (¬(j_0 is a forward junction ∧ j_1 is a back junction)) then
    back ← getOutExt(j_0, p_0, r)
    front ← getOutExt(j_1, p_1, r)
    insert(back ∘ front, B_3)
return B_3
```

special node is reached. During each such traversal from special node u to special node v, a unitig sequence s_{uv} is constructed. s_{uv} is initialized to the sequence of u, and a base is added at each extension until v is reached.

New data structures are constructed in the course of traversals in order to aid later queries and updates. A *ContigNode* structure is used to represent a junction that points to *Contigs*. ContigNodes are structures possessing a pointer to a Contig at each forward extension, as well as one backwards pointer. This backwards pointer connects the junction to the sequence beginning with the reverse complement of the junction's k-mer. Contigs initially store unitig sequences, but these may later be concatenated or duplicated. They also point to one ContigNode at each end. To efficiently query Contigs and ContigNodes, a new hashmap M' is constructed having junction k-mers as keys, and ContigNodes that represent those junctions as values. Isolated contigs formed by unitigs that extend between terminal nodes are stored in a separate set data structure.

Once the raw graph is obtained, cleaning steps commence, incorporating tip removal, chimera removal, collapsing of bulges, and disentanglement. Coverage information and paired-junction links are crucial to these steps. Briefly, tip removal involves deletion of Contigs shorter than the input read length that lead to a terminal node. Chimera and bulge removal steps involve heuristics designed to remove low coverage Contigs when a more credible alternative (higher coverage, or involved in more sub-paths) is identified. These first three steps proceed as described in [21], thus we omit their full description here.

Disentanglement relies on paired junction links inserted into B_3 and B_4. We iterate through the set of ContigNodes to look for 'tangles' - pairs of opposite-facing junctions joined by a repeat sequence - as shown in Figure 2. Tangles are characterized by tuples (j_1, j_2, s) where j_1 is a back junction, j_2 is a forward junction (or vice-versa), and there is a common Contig s pointed to by the back pointers of both j_1 and j_2. Junctions j_1 and j_2 each have at least two outward extensions. We restrict cleaning to tangles having exactly two extensions at each end. Let s_a and s_b be the Contigs starting at heads of j_1, and s_c and s_d be the Contigs starting at heads of j_2. By disentangling, we seek to pair extensions at each side of s to form two paths. The possible outputs are paths $[s_a, s, s_c]$ together with $[s_b, s, s_d]$ or $[s_a, s, s_d]$ together with $[s_b, s, s_c]$.

Thus, each such pair straddling the tangle -e.g., having one head on s_a and the other on s_c - lends some support to the hypothesis that the correct split is that which pairs the two. To decide between the two possible split orientations, we count the number of pairs supporting each by querying B_3 or B_4 for all possible junction pairings that are separated by a characteristic length associated with the pairs inserted to each. For example, B_3 stores heads out of non-consecutive junction pairs on the same read. Therefore, for each junction on s_a we count each pairing accepted by B_3 with a junction on s_c that is at most one read length away. Specifically for B_3, we also know that inserted pairs are always one or two junctions away from the starting junction, based on the scheme presented in Figure 3. To decide when a tangle should be split, we apply XOR logic to arrive at a decision: if the count of pairs supporting both paths in one orientation is greater than 0, and the count of both paths in the other orientation is 0, we disentangle according to the first, as shown in Figure 2. Similar yet more involved reasoning is used for junction links in B_4, using the insert size between read pairs (see Appendix). Once we arrive at a decision, we add a new sequence to the set of Contigs that is the concatenation of the sequences involved in the original paths. We note one of the consequences of this simplification step is that the graph no longer represents a de Bruijn graph, in that each k-mer is no longer guaranteed to appear at most once in the graph. Furthermore, the XOR case presented is the most frequently applied form of disentanglement out of a few alternatives. We discuss these alternatives in the Appendix.

Optimizations and technical details Here we discuss some details omitted from the above descriptions for the sake of completeness. Based on the description of Algorithm 1 and Algorithm 2, it is possible that false positive extensions out of terminal nodes will ensue. This is possible because the mechanism described for removing false positive junctions can differentiate between one or multiple extensions existing in G for a given node, but can not differentiate between one or none. This may lead to assembly errors at sink nodes.

To overcome such effects, we store distances between junctions seen on the same read with the distance recorded being assigned to the extension of each junction observed on the read. When an outermost junction on a read has not been previously linked to another junction, we record its distance from the nearest read end - this solves the problem mentioned previously as long as paths to sinks are shorter than read length. To obtain accurate measurements of distances on longer non-branching paths, we also introduce artificial 'dummy' junctions whenever a pre-defined length threshold is surpassed. In effect, this means that reads with no real junctions are assigned dummy junctions.

Once distances and dummy junctions are introduced, an additional benefit is gained: the speed of the read-scan can be improved by skipping between junctions that have been seen before. Once distances are known, if we see a particular extension out of a junction, and then a sequence of length ℓ without any junctions, then, wherever else we see that junction and extension, it must be followed by the exact same ℓ next bases. Otherwise, there would be a junction earlier. So we store ℓ when we see it, and skip subsequent occurrences.

Finally, we note that Faucet can benefit from precise Bloom filter sizing. When a good estimate of dataset parameters is known, the algorithm can do the 2-pass process above. Otherwise, to determine the numbers of distinct k-mers and the number of singletons in the dataset in a streaming manner, we have used the tool ntCard [15]. This requires an additional pass over the reads (for a total of three passes). The added pass does not increase RAM or disk use. In fact, in tests on locally stored data, we found it only adds negligible time.

No. of files	Time (hrs)	RAM (GB)	Disk (GB)	Data size (GB)	Comp. ratio
10	26.3	48.3	19.0	29.6	0.64
20	47.7	84.3	34.3	59.2	0.58
37	98.2	144.7	50.0	108.4	0.46

Table 1.

4 Results

Assembling while downloading As a demonstration of streaming assembly, we ran Faucet on publicly available human data, SRR034939, used for benchmarking in [6]. To assess resource use at different data volumes, we ran Faucet on 10, 20, and 37 paired-end files out of 37 total. Streaming was enabled using standard Linux command line tools: wget was used for commencing a download from a supplied URL, and streamed reading from the compressed data was enabled by the bzip2 utility. Downloads were initiated separately for each run. The streaming results are shown in Table 1.

We emphasize that Faucet required less space than the size of the input data in order to assemble it, while most assemblers generate files during the course of their processing that are larger than the input data. Also, the ratio of input data to disk used by Faucet decreased as data volume increased, reflecting the tendency of sequences to be seen repeatedly with high coverage. We also note that Faucet's outputs effectively create a lossy compression of the read data, in that the choice of k value inherently creates some ambiguity for read substrings larger than k. This compression format is also queryable, in that given a k-mer in the graph, its extensions can be found: indeed, this is the basis of Faucet's graph construction and cleaning.

Disentanglement assessment To gauge the benefits of disentanglement on assembly quality, we compared Faucet's outputs with and without each of short- and long-range pairing information, provided by Bloom filters B_3 and B_4, on SYN 64 - a synthetic metagenome produced to provide a dataset for which the ground truth is known comprised of 64 species (data set sizes and additional characteristics are provided in the Appendix). The results of this assessment are presented in Table 2. We measured assembly contiguity by the NGA50 measure. NG50 is defined as "the contig length such that using equal or longer length contigs produces x% of the length of the reference genome, rather than x% of the assembly length" in [22]. NGA50 is an adjustment of the NG50 measure designed to penalize contigs composed of misassembled parts by breaking contigs into aligned blocks after alignment to the reference. We found that disentanglement more than doubled contiguity measured by mean NGA50 values, with greater gains as more kinds of disentanglement were enabled. This was also reflected by corresponding gains in the genome fractions, and in the number of species for which at least 50% of the genome was aligned to, allowing NGA50 scores to be reported. More applications of disentanglement also increased the number of misassemblies reported and the duplication ratio, however two thirds of the maximum misassembly count is already seen without any disentanglement applied.

	No disent.	B_3 only	B_4 only	both B_3, B_4
Genome fraction (%)	76.4	79.9	80.3	82.3
Dup. ratio	1.00	1.01	1.02	1.02
Mean NGA50	13048	21703	26356	29066
Misassemblies	388	480	521	572
Species reported	54	56	56	56

Table 2.

Tools comparison We sought to assess Faucet's effectiveness in assembling metagenomes, and its resource efficiency. For the former, we compared Faucet to MetaSPAdes [23] and Megahit [24], state of the art metagenome assemblers in terms of contiguity and accuracy that require substantial resources. To address resource efficiency, we also compared Faucet to two leading resource efficient assemblers, Minia 3 (Beta) [6] and LightAssembler [17]. We note these last two were not designed as metagenome assemblers, but they perform operations similar to what Faucet does - both in the course of their graph construction steps, and in their cleaning steps. They differ from Faucet in that neither is capable of disentanglement, as they do not utilize paired-end information, but counter this advantage with more sophisticated traversal schemes. All tools were run on two metagenome data sets - SYN64 and HMP - a female tongue dorsum sample sequenced as part of the Human Microbiome Project. Both datasets were used for testing in [23]. To achieve a fair comparison, runs were performed with a single thread on the same machine, as Faucet does not currently support multi-threaded execution. Full details of the comparison, including versions, parameters, and data accessions, are presented in the supplement.

	Metaspades	Megahit	LightAssembler	Minia	Faucet
Genome fraction (%)	89.1	90.1	75.6	76.5	82.3
Dup. ratio	1.02	1.02	1.01	1.00	1.02
Mean NGA50 (kb)	167	99.0	2.60	14.6	30.7
Median NGA50 (kb)	71.1	57.6	2.30	10.5	23.7
Misassemblies	785	949	314	395	572
Species reported	59	61	55	52	56
Time (hrs)	41.2	10.9	1.63	0.97	2.61
Memory (GB)	26	9.1	2.7	4.8	6.0
Disk (GB)	43.1	14.3	28.2	1.84	1.59
Genome fraction (%)	46.9	48.6	23.4	27.8	27.9
Dup. ratio	1.05	1.12	1.02	1.01	1.05
Mean NGA50 (kb)	28.3	36.8	3.18	6.25	7.12
Median NGA50 (kb)	28.3	36.8	3.18	6.25	7.12
Misassemblies	504	602	100	184	202
Species reported	12	12	5	3	6
Time (hrs)	30.5	13.0	3.35	0.99	2.30
Memory (GB)	14	8.3	3.4	3.7	7.3
Disk (GB)	53.2	11.5	23.5	1.30	1.61

Table 3. Tool comparison on two metagenomes. Top values in each cell are for SYN 64 data, and bottom values are for HMP. Duplication ratio is the ratio between the total aligned length to the combined length of all references aligned to. The mean and median NGA50 values are calculated on based on species sufficiently covered by all assemblers to yield an NGA50 value (i.e., 50% of the genome is covered). Species reported are those for which an NGA50 value is reported. In the HMP data, only 2 species were reported for all, making the mean and median NGA50 values equal. Disk and memory use are those reported by the Linux time utility, and Disk use is the total amount written to disk during the course of a run.

Table 3 presents the full results for the tools comparison. There was a strong advantage to Megahit and MetaSPAdes over the three lightweight assemblers (Minia, LightAssembler, and Faucet) in terms of contiguity achieved (shown by NGA50 statistics), but this came at a large cost in terms of memory, disk space, and time, particularly in the case of MetaSPAdes. Among the lightweight assemblers, Minia used by far the most disk space, and differences in other resource measures were less pronounced. Among these three, Faucet had a large advantage in NGA50 statistics relative to the other two. This is highlighted by the trend of Table 3, and shown by its 14-110% advantage in the mean of NGA50 relative to Minia, and 2-11 *fold* advantage relative to LightAssembler.

5 Discussion

Streaming de novo assembly presents an opportunity to significantly ease some of the burdens introduced by the recent deluge of second generation sequencing data. We posit the main applications of streaming assembly will be de novo assembly of very large individual datasets (e.g., metagenomes from highly diverse environments) and re-assembly of pangenomes derived from many samples. In both cases, very large volumes of data must be digested in order to address the relevant biological questions behind these assays. Therefore, streaming graph assembly presents an attractive alternative to data compression: instead of attempting to reduce the size of data, the aim is to keep locally only relevant information in a manner that is queryable and that allows for future re-analysis.

Here, we have demonstrated a mechanism for performing streaming graph assembly and described some of its characteristics. First, we showed that assembly can be achieved without ever storing raw reads locally. By assembling the graph, an intermediate by-product of many assemblers, we show this technique is generally applicable. By refining the graph and showing better assembly contiguity than competing resource efficient tools on metagenome assembly, we showed this method can also be applied in the setting when sensitive recovery of rare sequences is crucial.

In future work, we aim to expand the capabilities of Faucet in a number of ways. Multi-threaded processing will reduce run times and make the tool more applicable to large data volumes. We believe further refinements of cleaning and contig generation can be achieved by adopting a statistical approach to making assembly decisions. In addition, beyond graph cleaning, we aim to apply Faucet's data structures to path generation, as done with paired end reads in [25, 26, 27]. Both have the potential to greatly improve contiguity and accuracy.

Beyond this, the present work raises several remaining challenges pertaining to what one may expect of streaming assembly. For instance, it is immediately appealing to ask if streaming assembly can be achieved with a just a single pass on the reads, and if so, what inherent limitations exist. In [12], a simple solution is proposed wherein the first 1M reads are processed to provide a succinct summary for the rest, but such an approach is more suited to high coverage or low entropy data, and thus unlikely to perform well on diverse metagenomes or when rare events are of particular interest. Another issue raised by the performance comparison herein is that of capturing the added value that iterative (multi-k value) graph generation provides. We have given a partial solution by capturing subsets of junction pairs within each read, and between mates of paired-end reads. Although it is possible to iteratively refine the graph with more passes on the reads, each time for the collection of k-mers at different lengths, this becomes unwieldy with large data volumes. Identifying the contexts for which such information would be useful in the graph and indexing the reads to allow for querying of such contexts may provide more efficient means of extracting such information.

6 Acknowledgments

This work was supported in part by the Israel Science Foundation as part of the ISF-NSFC joint program to RS. RS was supported in part by the Raymond and Beverley Chair in Bioinformatics at Tel Aviv University. EH was supported in part by the United States-Israel Binational Science Foundation (Grant 2012304) and EH and RR were supported in part by the Israel Science Foundation (Grant 1425/13). RR was supported in part by a fellowship from the Edmond J. Safra Center for Bioinformatics at Tel Aviv University, an IBM PhD fellowship, and by the Center for Absorption in Science, the Israel Ministry of Immigrant Absorption. EH is a Faculty Fellow of the Edmond J. Safra Center for Bioinformatics at Tel Aviv University. GG was supported by the MISTI MIT-Israel program at MIT and Tel Aviv University.

References

[1]Mihaela Pertea, Geo M Pertea, Corina M Antonescu, Tsung-Cheng Chang, Joshua T Mendell, and Steven L Salzberg. StringTie enables improved reconstruction of a transcriptome from RNA-seq reads. *Nature Biotechnology*, 33(3):290–295, feb 2015.

[2]Roye Rozov, Aya Brown Kav, David Bogumil, Naama Shterzer, Eran Halperin, Itzhak Mizrahi, and Ron Shamir. Recycler: an algorithm for detecting plasmids from <i>de novo</i> assembly graphs. *Bioinformatics*, 28(4):btw651, dec 2016.

[3]Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232, jan 2012.

[4]Adam M Novak, Glenn Hickey, Erik Garrison, Sean Blum, Abram Connelly, Alexander Dilthey, Jordan Eizenga, M. A. Saleh Elmohamed, Sally Guthrie, André Kahles, Stephen Keenan, Jerome Kelleher, Deniz Kural, Heng Li, Michael F Lin, Karen Miga, Nancy Ouyang, Goran Rakocevic, Maciek Smuga-Otto, Alexander Wait Zaranek, Richard Durbin, Gil McVean, David Haussler, and Benedict Paten. Genome Graphs. *bioRxiv*, 2017.

[5]J. Pell, a. Hintze, R. Canino-Koning, a. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012.

[6]Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms in Bioinformatics*, pages 236–248, 2012.

[7]Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12), 2010.

[8]Chengxi Ye, Zhanshan Sam Ma, Charles H Cannon, Mihai Pop, and Douglas W Yu. Exploiting sparseness in de novo genome assembly. *BMC bioinformatics*, 13 Suppl 6(6):S1, 2012.

[9]Ilia Minkin, Son Pham, and Paul Medvedev. TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics (Oxford, England)*, page btw609, 2016.

[10]Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev. On the representation of de bruijn graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8394 LNBI, pages 35–55, 2014.

[11]Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.

[12]Li Song, Liliana Florea, and Ben Langmead. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome Biology*, 15(11):509, nov 2014.

[13]Adam Roberts and Lior Pachter. Streaming fragment assignment for real-time analysis of sequencing experiments. *Nature Methods*, 10(1):71–73, nov 2012.

[14]Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C. Titus Brown. These are not the K-mers you are looking for: Efficient online K-mer counting using a probabilistic data structure. *PLoS ONE*, 9(7):e101271, jul 2014.

[15]Hamid Mohamadi, Hamza Khan, and Inanc Birol. ntCard: a streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, page btw832, jan 2017.

[16]P. Melsted and B. V. Halldorsson. KmerStream: streaming algorithms for k-mer abundance estimation. *Bioinformatics*, 30(24):3541–3547, dec 2014.

[17]Sara El-Metwally, Magdi Zakaria, and Taher Hamza. LightAssembler: Fast and memory-efficient assembly algorithm for high-throughput sequencing reads. *Bioinformatics*, 32(21):3215–3223, 2016.

[18]Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of Models for Sequence Assembly. In *Algorithms in Bioinformatics*, pages 289–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[19]Burton H. Bloom and Burton H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, jul 1970.

[20]P A Pevzner, H Tang, and M S Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–53, 2001.

[21]Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son Pham, Andrey D. Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max a. Alekseyev, and Pavel a. Pevzner. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[22]A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, apr 2013.

[23]S. Nurk, D. Meleshko, A. Korobeynikov, and P. Pevzner. metaSPAdes: a new versatile de novo metagenomics assembler. *aRxiv*, (2004):arXiv:1604.03071, 2016.

[24]Dinghua Li, Chi Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak Wah Lam. MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, 2014.

[25]Andrey D. Prjibelski, Irina Vasilinetc, Anton Bankevich, Alexey Gurevich, Tatiana Krivosheeva, Sergey Nurk, Son Pham, Anton Korobeynikov, Alla Lapidus, and Pavel A. Pevzner. ExSPAnder: A universal repeat resolver for DNA fragment assembly. *Bioinformatics*, 30(12), 2014.

[26]Rahul Nihalani and Srinivas Aluru. Effective Utilization of Paired Reads to Improve Length and Accuracy of Contigs in Genome Assembly. In *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 355–363, 2016.

[27]Wenyu Shi, Peifeng Ji, and Fangqing Zhao. The combination of direct and paired link graphs can boost repetitive genome assembly. *Nucleic acids research*, page gkw1191, 2016.

[28]M. Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, oct 2002.

[29]Esteban Walker and Amy S Nowacki. Understanding equivalence and noninferiority testing. *Journal of general internal medicine*, 26(2):192–6, feb 2011.

Appendix

Sizing Bloom filters

We used the tool ntCard to estimate the cardinality F_0 of the set of k-mers and the number of singletons f_1. These counts are used for optimizing both runtime and memory use by allowing us to minimize the size of Bloom filters m and the number of hash functions h used for both B_1 and B_2, the largest filters used. B_1 and B_2 share these parameters (and the same set of hash functions) to allow insertions of each s into B_2 for which $B_1(s) = 1$ without recalculating h hash values. We use the fact that elements inserted into B_2 are either non-singletons or false positives due to B_1. Thus, the expected number of elements n_2 in B_2, is bound by their sum, i.e.,

$$n_2 \leq (F_0 - f_1) + f_1 p_1 \tag{1}$$

where p_1 is the false positive rate of B_1. We note that since p is the *effective* false positive rate after all elements are inserted into B_1, this bound holds strictly and may be overly pessimistic regarding the number of false positives inserted into B_2, however it provides a simple means of setting parameters. To do so, we first recall that B_1 is discarded after loading, while B_2 is maintained and thus its false positive rate p_2 is the rate that affects all downstream queries. A default false positive rate of of $p_2 = 0.01$ is used to work backwards to derive a higher rate p_1, and Bloom filter parameters for both filters were set based on this derived value, using knowledge of F_0 and f_1. To derive p_1, we paired the expressions for the expected false positive rates with the expression for the optimal number of hash functions for a given false positive rate [28]:

$$p_1 = (1 - e^{\frac{F_0 h}{m}})^h \tag{2}$$

$$p_2 = (1 - e^{\frac{n_2 h}{m}})^h \tag{3}$$

$$h = \frac{m \ln(2)}{F_0}. \tag{4}$$

By plugging the value of h from equation 4 into equation 2, we arrive at $m = \frac{-F \ln(p_1)}{\ln(2)^2}$. Combining this and the above expressions, we arrive at

$$0 = -\ln(2) ln(p_2) - \ln(p_1) \ln(1 - 2^{-\frac{F_0 + (1-p_1) f_1}{F_0}}) \tag{5}$$

for which root-finding methods can be applied to finally extract p_1, the sole remaining unknown.

Currently, we have not yet found similar means of optimizing the sizes of filters B_3 and B_4, as it is unclear how to estimate the number of elements that will be inserted into them in advance. We therefore define their sizes based on empirical observations. For diverse metagenomes, where the number of singletons f_1 may be very close to the cardinality F_0, we expect there to be few junctions, as a junction k-mer must by definition occur at least twice in the data. Based on this observation, we set the expected number of elements in both B_3 and B_4 to be $\frac{F_0}{10}$ and found that this bound was not exceeded on tested datasets. For higher coverage data, where a significantly larger proportion of junctions is expected relative to F_0, we set the size of both filters to be $\frac{F_0}{2}$.

Solid junction counts

Total junction counts listed in the table below include real junctions, those due to false positives, and dummy junctions inside long linear stretches. We posit that the SYN 64 data set included many more fake (false positive and dummy) junctions as a result of having a much larger proportion of linear stretches, as reflected in the much larger genome fraction and N50 size (relative to HMP) output by Faucet.

	HMP	SYN 64
Total junctions (M)	7.11	9.23
Real junctions (M)	4.55	1.34
genome fraction (%)	27.9	82.3
N50	2290	16707

Table 4.

Inserting into B_4

When inserting into B_4, both the distance and relative orientation between paired-end mates is unknown. Therefore, a tiling scheme such as that seen in Figure 3 cannot be applied. Instead, we seek to ensure that in most cases when querying approximately one insert size away from a given junction u, there will be another junction v such that an extension of u will be paired with an extension of v in B_4. To achieve this end, and to avoid long run times due to pair insertions, we apply the following logic: for each junction u on the first mate, we only insert extensions of a new pair (u, v) if u has no pair in B_4. When a new pair must be inserted, v is chosen to be the first junction found on the second mate. During the insertion process, this logic allows us to break the querying process whenever one previously inserted pair is encountered, and lets us avoid inserting too many pairs into B_4, and thus risking increasing B_4's effective false positive rate.

Additional disentanglement

Other forms of disentanglement include resolution of loops and disentanglement by coverage. Loops are encountered when, e.g., s_a and s_c in Figure 2 are the same unitig, and disentanglement requires unwinding the loop and duplicating the s's sequence to arrive at the walk $[s_b, s, s_c, s, s_d]$. Disentanglement by coverage is allowed only when s is deemed too long for there to be support by junction pairs flanking opposite ends of s, and is applied when the coverage distributions of Contig pairs supporting a certain orientation (e.g., s_a paired with s_c and s_b with s_d for the case presented in Figure 2) is significantly similar, as determined by Two One Sided Tests [29] for each pair. To smooth coverage levels when this test is applied, coverage values are updated each time a cleaning step such as bulge removal is applied. For example, if a bubble includes one low coverage Contig s_1 and one high coverage Contig s_2, as extensions flanked by the same ContigNodes j_L and j_R, and s_2's coverage is sufficiently higher than s_1's, Contig s_1 will be removed, and its average coverage will be assigned to all (expired or fake) junctions on Contig s_2.

Tools comparison details

Tools and flags:
Faucet was run with k= 31
MetaSPAdes 3.9.0, default parameters
Megahit 1.1.1, default parameters
Minia 3 Beta, git commit 4b0a83a, k = 31
LightAssembler, no version information available, downloaded 1/17 from GitHub k = 31
MetaQUAST, 4.4.0, –fragmented flag

Data Sets:
SYN 64 (SRA accession SRX200676), 109M 100 bp paired end mates, I.S. 206
HMP (SRX024329), 149.6 M 100 bp paired end mates, I.S. 213