Genome Analysis

Faucet: streaming de novo assembly graph construction

Roye Rozov¹, Gil Goldshlager², Eran Halperin³*, Ron Shamir¹*

¹Blavatnik School of Computer Science, Tel-Aviv University, Tel Aviv, Israel
²Department of Mathematics, MIT
³Departments 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 $s \| t$, and the reverse complement of a string $s$ by $s^\prime$.

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 < k-1} \{s[i..i+k-1] : (k-1)-mer overlaps between nodes in $V$. Namely, identifying vertices with their k-mers, $(u, v) \in E \iff \text{suff}(u, k-1) = \text{pref}(v, k-1)$.

Each node $v$ is identified with its reverse complement $v^\prime$, 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 $(v, v^\prime)$ will be the lexicographically lesser of $v$ and $v^\prime$. 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 same character extensions of all following non-branching k-mers. Such merged non-branching paths are called units.

Since a junction $v$ having in-degree greater than 1 and out-degree 1 is identified with $v^\prime$ 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, $f_0, \ldots, f_h$, such that $\forall i \in [0, m-1], f_i(x) \in [0, m-1]$, 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 both $B_1$) are inserted to $B_2$. Namely, for each k-mer $s$, if $B_2[|s|] = 1$ then we insert $s$ into $B_2$, otherwise we insert into $B_1$. After iterating through all reads, $B_2$ 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
A. Online Stage

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.

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 $\in N^4$ is a vector $[c_1, c_2, c_3, c_4]$ 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. \hspace{1em} juncs $\leftarrow$ findJunctions($r, B_2$) \hspace{1em} \triangleright$ call to Algorithm 2
4. \hspace{2em} for $(\text{junc, pos}) \in \text{juncs}$ do
5. \hspace{3em} if junc $\notin M$ then
6. \hspace{4em} $M[\text{junc}] \leftarrow [0, 0, 0, 0]$
7. \hspace{3em} increment counter in $M$ for $r[pos + k]$ \hspace{1em} \triangleright$ call to Algorithm 3
8. recordPairs($r, \text{juncs}, B_3$) \hspace{1em} \triangleright$ call to Algorithm 3
9. 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.
Following the accounting performed on observed junctions, Faucet records adjacencies between pairs of junctions using an additional Bloom filter - B\textsubscript{3} and B\textsubscript{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 the repeat - in both paths. In order to do this, we first identify the heads that are inserted into the repeat. For each observed junction, we query the Bloom filter B\textsubscript{2} with each of its k-mers to find the closest indirect adjacency that may be informative when captured from a read. This is done by querying the Bloom filter for each junction and continuing until the next junction. Then, for each junction, we identify its paired junction by 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\textsubscript{2} for extensions and continuing until the next

Algorithm 2 \texttt{findJunctions}(r, B_2)

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{Input:} read $r$ and Bloom filter $B_2$
\State \textbf{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 $r$
\Function{juncTuples}{$r$, $B_2$}
\State $juncTuples \leftarrow \emptyset$
\For{$i \in [0, |r| - k]$ do $kmer \leftarrow r[i : i + k - 1]$}
\For{$c \in \Sigma \setminus \{r[i + k]\}$ do}
\If{$(\text{suffix}(kmer, k - 1) \circ c \in B_2)$ then}
\State $juncTuples \leftarrow juncTuples \cup (kmer, i)$
\EndIf
\EndFor
\EndFor
\State \Return $juncTuples$
\EndFunction
\end{algorithmic}
\end{algorithm}

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 $s_b$ and $s_c$ to $s_d$. Since no pairs link extensions on $s_a$ with $s_d$ or $s_c$ with $s_d$, only one orientation is supported. C. The result of disentanglement: paths $[s_a, s_c, s_d]$ and $[s_a, s_b, s_c]$ are each merged into individual sequences, and junctions $j_1$ and $j_2$ are removed from $M$.
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 $s_3$ 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 $s_3$ 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$(j, p, r)$` that for a junction $j$, returns `pref$(j, k-1) \circ r[p_1-k]$` if $j$ is a back junction, and `suff$(j, k-1) \circ r[p_1+k]$` otherwise.

**Output:** Bloom filter $B_3$, loaded with select linked k-mer pairs

```plaintext
1: if len(juncs) > 2 then
2:   for $i \in [0, \text{len(juncs)}-2]$ do
3:     back ← getOutExt$(j_i, p_1, r)$
4:     front ← getOutExt$(j_{i+2}, p_{i+2}, r)$
5:     insert(back $\circ$ front, $B_3$)
6:   end for
7: else if (len(juncs) = 2) ∧ ($j_0$ is a forward junction ∧ $j_1$ is a back junction) then
8:   back ← getOutExt$(j_0, p_0, r)$
9:   front ← getOutExt$(j_1, p_1, r)$
10:  insert(back $\circ$ front, $B_3$)
11: return $B_3$
```

A 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
cruel these steps. Briefly, trim 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 $E_3$ and $E_4$. 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 $j_1$, 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 $[s_a, s, s_b]$ together with $[s_c, s, s_d]$ or $[s_c, s, s_a]$ together with $[s_d, s, s_b]$. Then, each such pair straddling the tangle - e.g., having one head on $s_a$ and the other on $s_b$ - 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 $E_3$ or $E_4$ for all possible junction pairings that are separated by a characteristic length associated with the pairs inserted to each. For example, $E_3$ 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 $E_3$ with a junction on $s_c$ that is at most one read length away. Specifically for $E_3$, 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 $E_4$, 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 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 $\ell$ without any junctions, then, wherever else we see that junction and extension, it must be followed by the exact same $\ell$ next bases. Otherwise, there would be a junction earlier. So we store $\ell$ when we see it, and skip further 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 mCard [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 added negligible time.

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 a given 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 $E_3$ and $E_4$ 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]. NG50 is an adjustment of the NGS50 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 NG50 values, with greater gains as more disentanglement were enabled. This was also reflected by corresponding gains in the genome fractions, and in the number of species for each of which at least 50% of the genome was aligned to, allowing NG50 scores to be reported. More applications of disentanglement also increased

<table>
<thead>
<tr>
<th>No. of files</th>
<th>Time (hrs)</th>
<th>RAM (GB)</th>
<th>Disk (GB)</th>
<th>Data size (GB)</th>
<th>Comp. ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>47.7</td>
<td>84.3</td>
<td>54.3</td>
<td>79.2</td>
<td>0.58</td>
</tr>
<tr>
<td>37</td>
<td>98.2</td>
<td>144.4</td>
<td>90.0</td>
<td>108.4</td>
<td>0.46</td>
</tr>
</tbody>
</table>

Table 1.
the number of misassemblies reported and the duplication ratio, however two thirds of the maximum misassembly count is already seen without any disentanglement applied.

<table>
<thead>
<tr>
<th>Tools comparison</th>
<th>Misassemblies</th>
<th>Genome fraction (%)</th>
<th>Mean NGAs0 (kb)</th>
<th>Median NGAs0 (kb)</th>
<th>Dup. ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetaSPAdes</td>
<td>175</td>
<td>90.0</td>
<td>14.6</td>
<td>26.5</td>
<td>1.02</td>
</tr>
<tr>
<td>Megahit</td>
<td>158</td>
<td>90.1</td>
<td>14.4</td>
<td>25.6</td>
<td>1.02</td>
</tr>
<tr>
<td>LightAssembler</td>
<td>142</td>
<td>90.2</td>
<td>14.7</td>
<td>25.5</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table 2.
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 Beverly Chair in Bioinformatics at Tel Aviv University. EH was supported in part by the United States-Israel Binational Science Foundation (Grant 20122304) and EH and RR were supported in part by the Israel Science Foundation (Grant 2012304). RR was supported in part by a Faculty Fellowship of the Edmond J. Safra Center for Bioinformatics at Tel Aviv University, and 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


Appendix

Sizing Bloom filters

We used the tool BenCov to estimate the cardinality $F_0$ of the set of k-mers and the number of singlets $f_1$. These counts are used for optimizing both runtime and memory use by allowing us to minimize the size of Bloom filters $n_2$ and the number of hash functions $h$ used for both $B_2$ and $B_3$, 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 $B_2(s) = 1$ without recalculating the hash values. We use the fact that elements inserted into $B_2$ are either non-singlets or false positives due to $E_1$. Thus, the expected number of elements $n_2$ in $B_2$ is bound by their sum, i.e.:

$$n_2 \leq (F_0 - f_1) + f_1 p_2$$

where $p_2$ is the false positive rate of $B_2$. We note that since $p_2$ 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 $p_2 = 0.01$ 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]:

\[ p_1 = (1 - e^{-|\mathcal{H}|})^h \]  
\[ p_2 = (1 - e^{-|\mathcal{H}|})^k \]  
\[ h = m \ln(2) \frac{2 - (1 + 1/2)^k}{F_0} \]

By plugging the value of \( h \) from equation 4 into equation 2, we arrive at
\[ m = \frac{2 \ln(p_1)}{\ln(2)} \]

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 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 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 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 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 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 sole remaining unknown.