MATH-469 Project Checkin

Eric Biggers

March 23, 2014

The goal of my project is to implement an algorithm that can assemble a genome, given a set of reads that came from it. The key idea in the algorithm I am implementing is the construction of a string graph where the vertices correspond to reads and the edges are labeled with DNA sequences, and a path through the graph corresponds to a set of reads that assemble together consistently.

I have implemented the initial stages of the algorithm. Each step has been implemented as a separate executable program. This means that the overall assembly process will iteratively transform the data (e.g. the string graph) on-disk into the final assembly. The main currently implemented programs are:

- `compute-overlaps`: Takes as input a set of reads and outputs all pairwise overlaps between reads of a minimum length specified by the `--min-overlap-len` option. The algorithm avoids a pairwise comparison of all reads by using a hash table to find reads sharing a seed. Only exact overlaps are considered at this point.

- `remove-contained-reads`: Given a set of reads and the overlaps that were computed from them, finds all reads that are fully contained by another read and discards them, along with the corresponding overlaps. If a pair of reads is identical, only one copy of the read is kept.

- `build-directed-string-graph`: Given a set of reads and the overlaps that were computed from them, build the *directed* string graph that models the genome assembly.

- `build-bidirected-string-graph`: Given a set of reads and the overlaps that were computed from them, build the *bidirected* string graph that models the genome assembly.

- `transitive-reduction`: Given a string graph, remove edges $v \to x$ where there exist edges $v \to w \to x$, provided that $v \to x$ is labeled by the same sequence as that of $v \to w$ concatenated with $w \to x$.

- `collapse-unbranched-paths`: Given a string graph, collapse chains of vertices that have in-degree 1 and out-degree 1.

- `map-contained-reads`: Map the reads that were removed in `remove-contained-reads` back into the string graph to mark edges that are supported by more than one read and therefore are more likely to need to be traversed multiple times.

The modular design allows for multiple possible data flows, but a possible flow is `compute-overlaps` → `remove-contained-reads` → `build-directed-string-graph` → `transitive-reduction` → `map-contained-reads` → `collapse-unbranched-paths` → `digraph-to-bidigraph`, which will produce a transitively-reduced, collapsed bidirected string graph produced from overlaps between non-contained reads.

Even with the final stages not implemented, the algorithm is already very effective when used on completely random genomes. Since there are no spurious overlaps in a completely random genome with adequate length reads, the transitively-reduced string graph is a chain of vertices with no branching, which then collapses into a single edge that will be labeled with almost all the original sequence.

However, genomes with repeats will produce tangled graphs that will require the network flow and Eulerian path parts of the overall algorithm to be implemented.

What follows is an explanation of the initial stages of the algorithm on a very simple case. The diagram below shows a 64-base-pair "genome" (G.) that has had three 32-base-pair error-free reads taken from it (1., 2., and 3.). Note that the genome is two-stranded. `A` always pairs with `T` and `C` always pairs with `G`, but the two strands run in opposite directions. Reads 1 and 2 come from the top ("reverse") strand, while read 3 comes from the bottom ("forward") strand. The reads are directed as shown (e.g. read 1 begins with `ACAGC`, reading from right to left.)

```
1. ←TGAGCTTAAGGCTTATCTATCTTCAGACGACA←
2.              ←TTATCTATCTTCAGACGACTATTATAGCGCGG←
G. ←TGAGCTTAAGGCTTATCTATCTTCAGACGACTATTATAGCGCGGCCAAGACTACGCGGAGCCCC←
G. →ACTCGAATTCCGAATAGATAGAAGTCTGCTGATAATATCGCGCCGGTTCTGATGCGCCTCGGGG→
3.                           →TCTGCTGATAATATCGCGCCGGTTCTGATGCG→
```

From looking at the above diagram, it is apparent that read 1 overlaps with read 2 by 20 base pairs, read 2 overlaps with read 3 by 20 base pairs, and read 1 overlaps with read 3 by 8 base pairs. The task of the `compute-overlaps` program is to determine these overlaps. Note that reverse-complement overlaps (e.g. reads 2 and 3) are permissible.

Before building the string graph, `remove-contained-reads` is run to remove reads that are fully contained by another read. However, there are no such reads in this example.

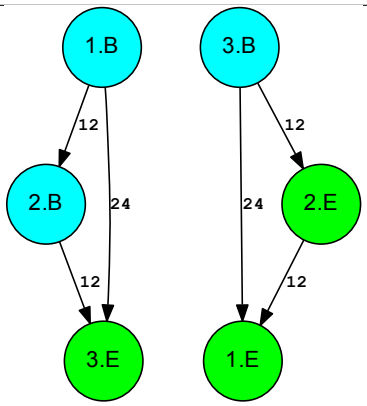
Given the set of overlaps, the next step is to build the bidirected or directed string graph. For this example, both these graphs are shown on the next page. In the directed graphs, a node $n.B$ means the beginning of read n, while a node $n.E$ means the end of read n. In the bidirected graphs, node n simply corresponds to read n. In both the directed and bidirected graphs, the edges are labeled by DNA sequences; however, in the drawings, only the lengths of the sequences are shown.

Consider graph (2A), the directed string graph after transitive reduction, and consider the walk $1.B \rightarrow 2.B \rightarrow 3.E$. This represents a way in which reads 1, 2, and 3 assemble together consistently. In the simplest case, all the nodes in a path would be E nodes, and this would correspond to a sequence of reads that assemble together as they are, with no reverse-complementation. However, in this case, we walk from the *beginning* of read 1 to the *beginning* of read 2, indicating that they are laid out in reverse-complement order, before reaching the *end* of read 3, which is in forward order. This reconstruction is consistent with how the reads were actually sampled from the genome (shown earlier).

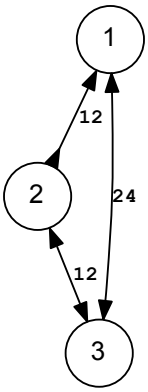
The other component of graph (2A) produces an equivalent assembly, but it lays out the reads in the opposite order and assembles the other strand of the genome.

Graph (2B) is equivalent to (2A), but combines every node $n.B$ and $n.E$ into one node n to deal with the double-strandedness in a more concise manner. The edges are bidirected. A bidirected edge may be traversed either way in a walk, but if a vertex is entered through an inward head, it must be left through an outward head, and vice versa.

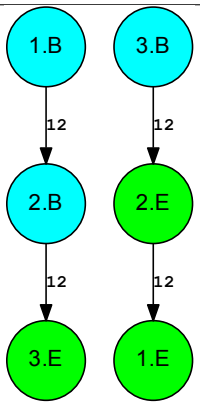
Graphs (3A) and (3B) result from collapsing unbranched paths in (2A) and (2B), respectively. Read 2 is preceded or succeeded only by reads 1 and 3, so its vertice(s) can be removed, and the adjacent edges can be squashed together. As mentioned before, this step can theoretically produce an edge labeled with nearly the entire genome if the genome is sufficiently short or random, as in this case. However, in the general case, due to genomic repeats, there will be many branches in the graph, and this needs to be taken into account by the later stages of the algorithm (see the last page for a slightly more complicated graph).



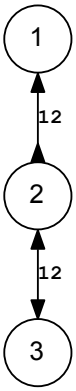
1A. Directed string graph of original data



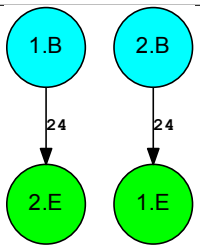
1B. Equivalent bidirected string graph to (1A)



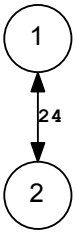
2A. Graph (1A) after transitive reduction. Read 2 is between reads 1 and 3, so there is no need for edges directly between 1 and 3.



2B. Graph (1B) after transitive reduction, or the equivalent bidirected string graph to (2A)



3A. Graph (2A) after collapsing unbranched paths



3B. Graph (2B) after collapsing unbranched paths, or the equivalent bidirected string graph to (3A).

Figure 1: Transitively-reduced, collapsed bidirected string graph for error-free 100bp reads uniformly sampled from the first 250000 base pairs of the genome of *E. coli*, which includes some repetitive sequences. The network flow algorithm must be able to determine how many times each edge should be traversed, then a generalized Eulerian path through the bidirected graph must be found to produce a possible assembly. In the general case, the final reconstruction may have to take the form of a set of paths rather than just one path.