MATH-469 Project Checkin

Eric Biggers

November 9, 2012

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 edges are labeled with DNA sequences and the vertices correspond to reads, 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 binary program. This means that the overall assembly process will iteratively transform the data (including the graph) on-disk into the final assembly. The currently implemented programs are:

- `convert-reads`: Imports and merges a set of reads into a concise binary representation.
- `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.
- `print-overlaps`: Prints a textual representation of a set of overlaps, given a file containing the overlaps in binary format.
- `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 discard them, along with the corresponding overlaps. If a pair of reads is identicaly, 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.

- `print-string-graph`: Prints a textual representation of a directed or bidirected string graph, or prints statistics about the graph.

- `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 $v \to x$. This program currently only works on directed string graphs, but the bidirected case is in progress.

- `collapse-unbranched-paths`: Given a string graph, collapse chains of vertices that have in-degree 1 and out-degree 1. This program currently only works on directed string graphs, but the bidirected case is in progress.

- `digraph-to-bidigraph`: Convert a directed string graph into a bidirected string graph.

The modular design allows for multiple possible data flows, but a possible flow is `convert-reads` → `compute-overlaps` → `remove-contained-reads` → `build-directed-string-graph` → `transitive-reduction` → `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 effective when used on random genomes because with no spurious overlaps, the transitively-reduced string graph is a chain of vertices with no branching, which then collapses into a single edge in `collapse-unbranched-paths`. This is true even if the genome is 15 million random base pairs, given 100 bp reads sampled uniformly from either strand with no errors. (Such a genome takes about 1 minute to assemble with maximum memory usage 1.8 GB, with the main time and memory bottleneck being the `compute-overlaps` program).

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" that has had three 32-base-pair error-free reads at a constant separation of 12 base pairs. 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, not `TGAGC`).

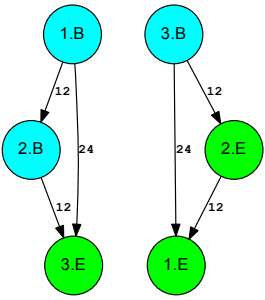
```
1. ←TGAGCTTAAGGCTTATCTATCTTCAGACGACA←
2.             ←TTATCTATCTTCAGACGACTATTATAGCGCGG←
G. ←TGAGCTTAAGGCTTATCTATCTTCAGACGACTATTATAGCGCGGCCAAGACTACGCGGAGCCCC←
G. →ACTCGAATTCCGAATAGATAGAAGTCTGCTGATAATATCGCGCCGGTTCTGATGCGCCTCGGGG→
3.                          →TCTGCTGATAATATCGCGCCGGTTCTGATGCG→
```

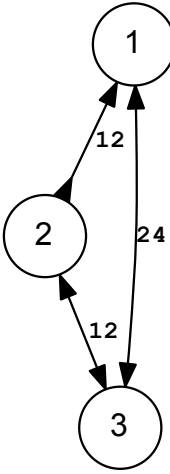
From looking at the diagram, it is apparent that Read 1 overlaps with Read 2 by 20 base pairs, Read 1 overlaps with Read 2 by 20 base pairs, and Read 1 overlaps with Read 3 by 8 base pairs. These overlaps are computed by the `compute-overlaps`

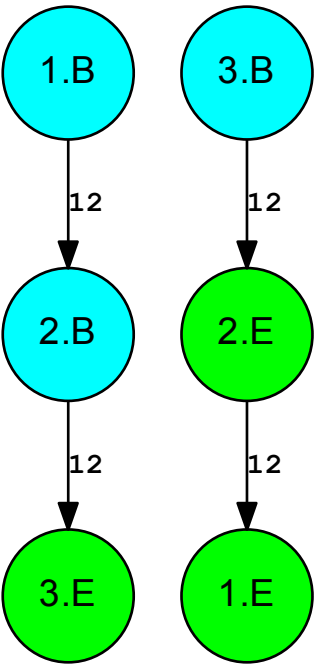
program. Note that an overlap may match either the forward or reverse-complement sequence.

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

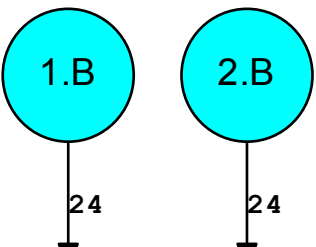
Given the set of overlaps, the next step is to build a bidirected or directed string graph. Both types of graphs represent how reads are

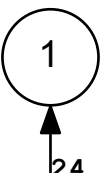

1.B
3.B
12
12
2.B
24
24
2.E
12
12
3.E
1.E


1
12
2
24
12
3


1.B
3.B
12
12
2.B
2.E
12
12
3.E
1.E


1
12
2
12
3


1.B
2.B
24
24


1
24