Overlap Consensus Assembly Project

Ken Reese, Kevin Boehme, Shaun Miller

Methods

Error Handling

The overlap consensus approach to contig assembly has an innate and flexible method for overcoming sequencing error. Our overlap consensus assembler performs a pairwise comparison between all reads, calculating a maximum overlap score between the two and generates a NxN matrix where N is the number of reads to assemble. This overlap scoring approach inherently accounts for errors in the reads by allowing for high overlap scores even in the presence of a few rare errors.

Branching Node

Our algorithm was an implementation of a greedy travelling salesman algorithm. We arbitrarily chose the first contig in the matrix and chose its highest scoring overlap (that wasn't itself). After choosing a best scoring overlapping contig we remove all other branches from the original contig. Then, we select the newly chosen contig and perform the same operation on it, namely choosing the highest scoring overlapping contig (that is still available) and then removing all other branches. We broke contigs when the overlap score was sufficiently low. This approach maximizes each consecutive overlap score starting at any arbitrary contig until reaching each contig. It does not, however, guarantee that the absolute maximum path will be found.

Assembly Quality

In order to assess the quality of our assembly we focused on the following informative metrics: number of contigs, largest contig size, N50, and mean contig size. The results for the following datasets are found below, with a discussion on the quality in the next section.

<u>small.easy.fasta</u>

Number of contigs: 1
N50: 903
Mean contig length: 903.00
Minimum contig length: 903
Maximum contig length: 903

<u>synthetic.small.noerror.fasta</u>

Number of contigs: 2
N50: 557
Mean contig length: 401.00
Minimum contig length: 245
Maximum contig length: 557

synthetic.large.noerrors.fasta
Number of contigs: 22
N50: 1125
Mean contig length: 212.86
Minimum contig length: 50
Maximum contig length: 1560

real.small.error.fasta
Number of contigs: 1
N50: 1238
Mean contig length: 1238.00
Minimum contig length: 1238
Maximum contig length: 1238

real.large.error.fasta
Number of contigs: 268
N50: 279
Mean contig length: 212.03
Minimum contig length: 100
Maximum contig length: 6700

Assembly Comparison

Metrics for the comparative analysis were produced by running each assembler on dataset 6 (a large, but not crazy large dataset - 19.fastq).

Our OLC assembler (wasn't able to process entire 100,000 reads)
Number of contigs: 3
N50: 3222
Mean contig length: 2876.33
Maximum contig length: 3659

Cap3
Number of contigs: 6836
N50: 526
Mean contig length: 369.82
Maximum contig length: 5228

Our de Bruijn

Average (mean) contig size: 79.42
N-50:100
Total contigs generated: 40,286
Largest contig size: 1,974

Discussion

The first major caveat we must point out in our comparison is that our assembler wasn't able to process the full reads list of the 100,000 reads file. We were able to generate a scoring matrix for each pairwise comparison but failed to fully traverse it using our travelling salesman approach due to memory limitations (we exceeded 250 gb memory allocation). We performed a low memory rework of our implementation that ended up running fascinatingly slow. As such we were only able to travel 183 nodes in about 10 hours.In any case, we assembled these first 183 reads to produce our results.

We compared our results with that of CAP3. We used the default settings and found CAP3 ran very quickly (12 minutes) and with significantly less memory than ours. We also thought it would be enlightening to compare our OLC results and those of Cap3 with our previously implemented de Bruijn Graph assembler.

Comparing our assembler to Cap3 we see that ours wasn't able to produce as quality of an assembly. A major consideration is the time and memory efficiency of the assembler and in both respects Cap3 is superior to our assembler.

Comparing just our overlap graph with our de Bruijn graph shows that the overlap approach produces much fewer, and larger contigs. We believe this is because of how dynamic the overlap scoring algorithm is and how well it handles sequencing errors. It is worth mentioning that while our de Bruijn graph produced far worse assemblies it ran much quicker (20 seconds). The de Bruijn graph approach to genome assembly has many limitations especially when it comes to long stretches of homopolymeric regions. However, we believe that running a preliminary assembly using a de Bruijn graph coupled with a more sophisticated and computationally expensive overlap consensus could be a potentially powerful approach to read assembly.

BLAST Results

CAP3 Results

The longest CAP3 contig was 5315 base pairs long, and mapped extremely well to Chromosome 19, Chromosome 15, and Chromosome 1. In Chr19, we see if mapping to a pseudogene of the OR4G3P gene, which is an olfactory receptor gene, part of the largest gene family in the human genome. That means than a somewhat imperfect copy of this gene's functioning version is stored at this location in chromosome 19. The match in Chr15 maps to OR4G6P, a very similar olfactory receptor pseudogene, while the third match is in Chr1, and is also an olfactory receptor pseudogene, OR4G4P.

These matches were all extremely exact- less than 20 mismatches/indels out of over 5000 potential matches! It may be interesting to speculate on the function of these pseudogenes' 'real' counterparts, considering the relative similarity of these genes (but considering olfactory function and sensitivity, this shouldn't be too surprising- but the extent of the similarity is nevertheless notable, especially considering the increasingly critical role we view smell to play in emotional health!).

Our Own Results

Our process resulted in a contig 3559 base pairs in length. In contrast to the CAP3 analysis, which found a contig that fit neatly into three places in the human genome, only one result (on the expected Chr19) had greater than 50% Query coverage. Our contig was also associated with a different gene- ABHD17A. This codes for an anhydrolase containing protein, the gene itself it predicted by BestRefSeq and Gnomon. It is our opinion that this could be a point in favor of our analysis, as we were able to find a contig matching a unique portion of Chr19.

Improvements

Assembly

Our assembler could benefit from a number of improvements. First, the speed and efficiency of our code and algorithm have a lot of room for improvement. Attempting our algorithm on a full human chromosome proved to be an impossible task. For this assembler to be at all viable we would need to rework many slower portions of the code as well as bring fundamental improvements to the overlap consensus algorithm mainly in the form of avoiding costly pairwise comparisons between irrelevent reads.

Another major improvement would be to implement a trimming algorithm before performing the alignment. For example, the Fastq format provides important quality metrics that should be taken into account by any serious assembler, however ours does not make use of this data. A quick preprocessing of reads based on their quality metrics would allow our assembler to prune/trim low quality reads and thus improve our downstream assembly.

Finally, our assembler could benefit from improvements in usability. We have many chunks of the assembly broken down into separate scripts but don't have a coherent pipeline to connect them all.

Overcoming Memory Limitations

For the smaller datasets, we were able to fit the entire matrix in memory. However, for the larger dataset with 100,000 reads, storing the matrix in memory proved to be more difficult. To overcome this, we tried two different approaches, outlined in brief below.

The first approach involved associating each read with a number (corresponding to its line number in the reads file) in order to compress the keys. We implemented this method using Python and used a large amount of memory on the supercomputer. For a reason unknown at the time of this writing, this method repeatedly failed after several hours on the supercomputer.

Our second approach was written in Java, and used an on disk key-value store to store the matrix. This enabled “relatively” fast access to individual entries in the matrix. The biggest limitation with this method is the amount of time it took to build the key-value store from the matrix file. Because we began this solution after several other attempts failed, we were not able to run this method for an extended period, and were able to produce only a limited number of contigs.
Conclusion: Although calculating the entire 100,000 by 100,000 matrix proved to be tractable, using this matrix to produce contigs proved to be rather intractable.

Project Suggestions

1. Perhaps a preliminary assignment where we detail our approach and talk it through with a TA. The TA can then tell us of potential things to consider when implementing our approach.
2. Perhaps consider removing the synthetic datasets and include only the very small dataset (for testing purposes) and then the large human reads with errors. In our experience, we made sure our program worked on the very small dataset then went straight to work on the bigger more realistic datasets. The medium size synthetic datasets were somewhat of an afterthought and didn't really contribute to our assembly report (except that we ran them because it’s required).