Metagenomic Assembly using Graph Algorithm and Applications

Sergey Koren[†]
sergek@cs.umd.edu

June 20, 2008

Abstract

Metagenomic assembly, the simultaneous assembly of a collection of organisms in an environment, has had many recent successes (gut data (macaque, human, japan human, etc)) and is the focus of many ongoing studies (HMP). Current assembly techniques have been tuned for single-organism assembly and are confounded by metagenomic data. As metagenomics becomes more popular and common, assembly software must be modified to exploit the characteristics of this novel data. We propose to explore and apply novel algorithms to maximize the benefits and utility of metagenomic datasets.

1 Introduction

In a traditional whole-genome shotgun sequencing project, assembly software reconstructs a genome from multiple short strings - collected by varying sequencers - called reads. The reads represent overlapping portions of the genome. Multiple sequencing techniques have been developed to generate the reads including the classic Sanger sequencing as well as next-generation platforms that generate short reads such as Solexa and SOLiD currently generate reads of 35bp [6],[11]. The pyrosequencing platform developed by 454 Life Sciences generates medium length reads of as long as 250 bp [14].

[†]Department of Computer Science, University of Maryland, College Park, MD

Many of these technologies can also generate paired-ends reads of varying lengths which, in addition to the reads, have a constraint on their distance and orientation.

Most whole-genome assemblers were developed in the age of Sanger sequencing for single-organism whole-genome sequencing projects. Recently, several assemblers have been modified to work with the new shorter read technologies [7], [22]. Several have recently been modified to handle mixed sets of reads [8], [15], [12], [19]. Metagenomic data presents challenges similar to those presented by novel sequencing technologies.

Current assemblers, developed for single-organism assembly, have been successfully used to assemble metagenomic data sets [20], [21]. However, these assemblies are not well specified, require manual intervention and undocumented settings, and are error prone. Therefore, they are not reproducible between different centers and not suitable for high-throughput settings which require many consistent runs. Therefore, we seek to develop new techniques for metagenomic assembly.

We will make the software available to the bioinformatics community by integrating with AMOS [2] and the Celera Assembler [3], two widely-used open-source platforms for genome assembly and analysis.

2 Background

Assembly software generally has several modules for successive phases. The phases are: overlap detection; ungapped sequence alignments called unitigs; assembling unitigs into scaffolds, gaped sequence alignments, using paired-end reads; and consensus determination.

Many of the assembly phases may be viewed as a series of graph reductions. The first graph is a read-ends graph, where each read consists of two nodes in the graph, representing the read ends. Each pair of read ends from a single read is connected by an undirected edge while directed edges connect overlapping read ends. A valid unitig is a path through the graph that crosses an undirected edge after every directed edge. A divergent path in the graph represents a contradiction in the data and stops the unitig under construction.

After the unitigs are constructed, they may be viewed as the nodes in a new graph. Using the set of paired reads in a unitig, we can bundle consistent pairs together into a weighted edge. Since each of the read pairs provides a

distance and an orientation, each edge between two unitigs has an orientation and a mean distance. The resulting graph is bi-directed [10]. That is, there are four possible edges between any two nodes:

- Both arrows pointing inwards, away from the nodes. This is generally known as an innie edge.
- Both arrows pointing outward, towards the nodes. This is generally known as an outie edge.
- Left arrow pointing inward, right arrow pointing outward. This is generally known as a normal edge.
- Right arrow pointing inward, left arrow pointing outward. This is generally known as an anti-normal edge.

Scaffolding starts by assigning an arbitrary orientation to a node and placing it at position 0. It them proceeds to orient nodes consistently with the orientations and positions implied by the edges. As a result of this operation the bidirected graph is transformed into a directed one. A scaffolding module must minimize the number of back edges, or contradictory edges, that are unsatisfied after it is finished. Each connected component in a graph becomes a scaffold. A final step is to project the graph form of a scaffold only a linear sequence. Several unitigs may occupy the same position and must be reconciled. This reconciliation sometimes leads to scaffolds being broken where unitigs cannot be linearized.

A complication in any assembly are genomic repeats. Repeats cause problem both in the overlap graph and in the scaffold graph. In overlap graphs, they can connect chimeric sections of the genome and lead to divergent paths. In scaffold graphs a repeat unitig introduces many false edges and connects disparate sections of the genome, such as different chromosomes, leading to graph tangles or cycles. The repeats prevent simplification of the scaffold graph. Therefore, assemblers try to identify the repetitive areas for special handling. For example, the Celera Assembler [16] uses coverage, based on the average number of bases between read start positions, to identify repeats. An increase in coverage is presumed to be an indicator of a repeat. These may also be identified using local graph features such as converging and diverging paths.

In metagenomic assembly, high coverage and local graph features do not necessarily correspond to repeats. Over-represented organisms may appear

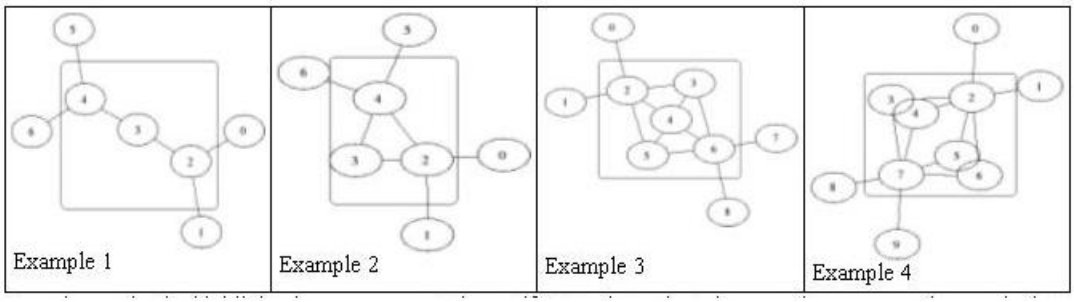

Figure 1: In each sample, the highlighted area represents the motif. Note that only 1-deep graphs are currently searched. In the first example, only 3-length motifs are found, the motif 2-3-4-5 would ge counted as 2-3-4 and 5-4-3. However the width of the subgraph is arbitrary. In the fourth exaple, the subgraph would be found if the middle section (nodes 3,4,5,6) contained only nodes 3,4 or if it contained more than four nodes.

repetitive due to their higher abundance relative to other organisms. Divergent paths, which are expected to be high in a metagenomic assembly due to intra-species polymorphism, are also not indicative of repeats. Therefore, linearizing scaffolds may break correct assemblies of an organism when two polymorphic unitig cannot be reconciled.

We propose to identify graph motifs that are common in metagenomic assembly and apply them as reductions to the graph before assembling. As an example, suppose a two organisms share common sequence, separated by a diverged region (see figure 1). This pattern appears as a bulge in the graph. A typical assembler will generate four separate scaffolds. If this pattern is recognized, the assembler can generate a single scaffold, detailing the polymorphism. We believe that recursively reducing the graph by identifying and collapsing these motifs will simplify the graph and identify conserved or divergent regions of interest.

We previously explored graph reduction to improve assembly. Given the overlaps for a set of reads R, we construct a multi-graph G with both directed and undirected edges. Each read r is represented by a pair of nodes $r_{5'}$ and $r_{3'}$, representing the two ends of a read, connected by an undirected edge. Directed edges represent dovetail overlaps, that is, those that span one end of each read. A dovetail path is an acyclic path in G that includes an undirected edge before and after every directed edge. In general, unitigging proceeds by following dovetail paths until there is a split, at which point a new unitig is

started. We aggressively simplified the graph by removing all directed edges except those that represent the mutually-best edge between any pair of read ends. A mutually-best edge is one that spans the most bases. We then record for each read r the number of nodes reachable from it, the score for a read is the sum of its nodes ($r_{5'}$ and $r_{3'}$). Unitigs are constructed by starting with the highest scoring read to build the longest unitigs first. Whenever a path ends, we proceed to the next highest-scoring read that is not already in a unitig. Finally, heuristics are applied to split unitigs that were incorrect. The resulting assemblies are better than other available assemblers[15].

We have also explored graph reductions in the consensus step. A multi-alignment can be represented as a graph G. Vertices represent non-overlapping sequence segments, edges connect vertices and represent ungapped alignments between reads while gaps are implicit. For a set S of n reads, G is n-partite. Every position in a read r is represented in exactly one vertex. Since some edges may be contradictory, only a subset of the edges E can be satisfied in a given alignment. A proper subset $E' \subset E$, called a trace, can be found efficiently using heuristics. The consensus tool built on this concept generates superior multi-alignments, especially in the case of high-error and short reads[18].

The alignment graph approach allowed the consensus tool to create better layouts, especially in the presence of high-error reads[18].

Assembly graphs are large and complex. Optimally finding all motifs is NP-complete. We will explore and apply approximation algorithms. We will also attempt to parallelize the algorithms where possible, using randomized algorithm and graph partitioning approaches. Since repeat unitigs will still pose a problem for assembly, we will explore novel methods for identifying repeats using global graph structure.

Metagenomic assembly also requires novel visualization techniques. Biologists need to view how unitigs are connected to see the paths representing organisms as well as areas of polymorphism between them. Current visualization is static [17] and the complex graph structure as well as poor node layout makes it difficult to view. We believe that by simplifying the graphs, they will be easy enough for human interpretation. We will explore how to lay out the graphs in a logical order while minimizing the number of back-edges, or cycles, and path crossings. This could be done using the minimum feedback vertex set algorithm, which is again NP-complete. We will evaluate graph visualization techniques, such as those presented in [13], and explore novel algorithms. We will finally allow custom visualization by enabling zoom and

allowing the user to dynamically expanded sections of the graph that have previously been collapsed.

3 Methods

Scaffolding consists of three operations: orientation, positioning, and simplification.

We must first convert the bi-directed graph into a directed graph, orienting the unitigs. While converting, we would like to minimize the number of back-edges, or cycles, created in the graph. That is, we would like a uni-directional ordering of the nodes of the graph. This problem may be formulated as a the minimum feedback arc set. which is NP-complete[9].

In addition to assigning an edge direction, we want to assign a position. There may be multiple edges assigning contradictory positions to a unitig. We want to maximize the number of satisfied edges by placing unitigs as close to the specified position as possible. This can be formulated as a least squares minimization of $\sum abs(v.distance - ((u.distance + e.distance) \forall e(u,v) \in E)$. That is, we desire to minimize the sum of the distances between the assigned unitig position and the positions specified by the edges incident on that unitig.

The prototype implementation uses greedy approaches. The orientation and distances are computed using the first edge incident on a vertex. Contradictory edges are ignored. The greedy algorithm is based on a breadth-first search traversal and has a complexity of $O(|E| + |V|)$. The details are in algorithm 1.

Once we have oriented the unitigs and positioned them, we search for common motifs to simplify the graph. To make the problem simpler, only one-level deep motifs are analyzed (see figure 1. The prototype is based on algorithm 2. Each iteration of algorithm 2 has a worst-case runtime of $O(|V| \times (\Delta(G)^2 + 3\Delta(G)))$ where $\Delta(G)$ is the maximum degree of G. This has a worst-case runtime of $O(|V| \times (|E|^2 + 3|E|))$. However, in a unitig graph it is likely that $\Delta(G) << |E|$. Every level of depth multiplies the runtime by a factor of $\Delta(G)$. Therefore, it is necessary to explore efficient approximation and parallel algorithms to run on larger motifs or motifs that are more than one-level deep.

Algorithm 1 Order and Orient Unitigs Given a Bi-Directed Multi-Graph

Given a bi-directed, multi-graph $G(V, E)$
Each vertex $v) \in V$ has a position and orientation.
Each edge $e(u, v) \in E$ has an orientation and a distance.
Queue q initialized to empty.
for all $v \in V$ **do**
  v.position $\leftarrow$ uninitialized, v.orientation $\leftarrow$ uninitialized
end for
while $\exists v \in V$ such that v is uninitialized **do**
  $q.clear()$
  $q.push(v)$
  $v.position \leftarrow 0$, $v.orientation \leftarrow$ FORWARD
  while $q.isEmpty() ==$ FALSE **do**
    $v = q.pop()$
    for all $u \in V$ such that $\exists e(u, v) \in E$ **do**
      $q.push(u)$
      $position \leftarrow v.position + e.distance$
      $orientation \leftarrow getOrientation(v.orientation, e.orientation)$
      if $u.position ==$ uninitialized **then**
        $u.position \leftarrow position$
        $u.orientation \leftarrow orientation$
      else
        $position \leftarrow \frac{position+u.position}{2}$
        $dist \leftarrow position - u.position$
        if $dist \leq$ LIMIT AND $u.orientation == orientation$ **then**
          $u.position \leftarrow position$
        end if
      end if
    end for
  end while
end while

Algorithm 2 Identify and Simplify Motifs in Unitig Graph

Given a directed, multi-graph $G(V, E)$
Each vertex $v) \in V$ has a position and orientation.
Each edge $e(u, v) \in E$ has an orientation and a distance.
Set M initialized to empty.
while a motif is found **do**
  for all $v \in V$ **do**
    for all $u \in V$ such that $\exists e(v, u) \in E$ **do**
      $sink \leftarrow UNINITIALIZED$
      $numNeighbors \leftarrow 0$
      for all $w \in V$ such that $\exists e(u, w) \in E$ **do**
        if $sink \neq w$ AND $sink \neq$v **then**
          $sink \leftarrow w$
          $numNeighbors = numNeighbors + 1$
        end if
      end for
      if $numNeighbors == 1$ **then**
        $M.add(v)$, $M.add(u)$, $M.add(sink)$
      end if
    end for
    for all $u \in V$ such that $\exists e(v, u) \in E$ **do**
      if $u \notin M$ **then**
        $M.clear()$
      end if
    end for
    for all $u \in V$ such that $\exists e(u, sink) \in E$ **do**
      if $u \notin M$ **then**
        $M.clear()$
      end if
    end for
    for all $m \in$M — $m \neq v$ AND $m \neq sink$ **do**
      for all $u \in V$ such that $\exists e(m, u) \in E$ or $\exists e(u, m) \in E$ **do**
        if $u \notin M$ **then**
          $M.clear()$
        end if
      end for
    end for
  end for
end while

4 Implementation

The prototype software implementation, named Bambus 2, is available as part of the AMOS package [2]. The AMOS package consists mostly of C++ programs with some shell and perl script utilities. The Bundler 2 pipeline is written in C++.

AMOS also includes a binary file format, known as a bank, that contains all the information about an assembly. This includes read sequences, contig layout and multi-alignment, and mate pair information. The bank allows random-access read/write operations. Each element of a bank is represented with AMOS as a C++ class. AMOS also includes a large set of utilities for converting Celera Assembler [3] output, among others, to AMOS bank format. We also include the SeqAn library for graph processing to simply graph storage and graph algorithms [5].

By integrating with AMOS, Bambus 2 can exploit the power of an existing random-access binary database while maintaining interoperability with other open-source tools. The Bambus 2 implementation consists of three executables working in a pipeline:

- CLK - This executable processes an AMOS bank, identifies all the mate pair information present, and creates contig link messages.
- Bundler - This executable processes the contig link messages generated by clk and bundles consistent ones together into contig edge messages.
- OrientContigs - This executable processes the contig edge messages output by Bundler and assigns and order and orientation to each of the contigs in the AMOS bank. It outputs both an AGP-formatted file [1] and a Graphvis-formatted graph [4].

Below we describe the implementation details of each step in the Bambus 2 pipeline.

4.1 Contig Link Builder - CLK

The purpose of the CLK module is to converts mate pairs connecting reads to mate pairs connecting contigs. The module starts by building a mapping of reads to contigs and reads to libraries. It then reads all the mate pairs present in a bank. For any mate pair that connects two contigs, the module

adjusts the size based on the position of the mated reads in the contigs. It also adjusts the orientation based on the orientation of the reads within the contig. The standard deviation remains unchanged. Finally, the module outputs all the contig link messages that were generated into a bank.

4.2 Contig Link Bundler - Bundler

The Bundler module serves to bundle together consistent links between contigs in contig edges. A contig edges differs from contig links since it combines several consistent contig links into one weighted edge. Currently, the weight is equal to the number of consistent contig links connecting the contigs. This can be expanded to include other sources of connection information such as overlaps or to give higher confidence to some edges versus others. Multiple consistent edges can exist between a pair of contigs. The Bundler will output different orientation edges between contigs but it will not output inconsistent-length edges.

4.3 Contig Edge Processor - OrientContigs

OrientContigs is the scaffolding module of Bambus 2. It reads in the contigs and contig edges and assigns each contig a position and an orientation (either forward or reverse). Connected components in the contig graph become scaffolds where the first contig in the output is adjusted to start at position 0. The scaffolds are not linearized.

The module includes options to ignore contig edges below a specified weight, to ignore contigs marked as repetitive, and to perform graph simplification. If graph simplification is enabled, the module recursively searches for known metagenomic motifs and collapses them into a single contig. We will update the implementation to instead split the motif out into a separate scaffold that is referenced by the original. Finally, when no more motifs are found OrientContigs outputs the results in both AGP and Graphviz formats.

4.4 Repeat Detection

We have also implemented several novel repeat detection method as a prototype.:

- Component-Joining Repeats - The first method is designed for a general scenario. It is based on the observation that repeat connect compose

separate sections of a genome. Therefore, the node they represent will be connected to many nodes in the graph. In order to detect them, we calculated all-pairs-shortest paths and for each node, v, we calculate the number of times it appears on a shortest path: P_v. A node is declared repeat if $P_v < \bar{x} - c \times \sigma$ or $P_v > \bar{x} + c \times \sigma$ where c is a constant.

- Connected-Component Repeats - The second method is designed for a metagenomic setting. Traditional assemblers use statistics calculated on the entire data set to identify repetitive contigs. We find the strongly-connected components of the graph. For each strongly connected component S, for each node $v \in S$, we compute the astat value as in [16]. An abundant organism that is less likely to appear repetitive in our approach as the connected component will not include nodes from other organisms in the astat calculation.

4.5 Performance

In order to evaluate scalability, we ran several benchmarks through the Bambus 2 pipeline. We recorded run times for on a 2GHz machine with 2GB of memory running Cygwin. The results are presented in table 1. Note that the pipeline does not depend on the number of input reads. The clk scales with the number of mates input. The Bundler and OrientContigs modules scale with the number of contigs and their edges. In the worst case, this can be equal to the number of mates if each mate-end is a contig. However, this is very unlikely in any real assembly. Therefore, the algorithms used in Bambus 2 have lower performance requirements than those used in other assembly steps, such as overlapping, which scales with the number of reads.

We also evaluated the performance impact of our repeat detection. The results are in table 2. Note that only the times for OrientContigs is presented as it is the only component that includes repeat-detection code. Notice the '-' in the lower-right corner. The algorithm did not complete on the machine due to memory limitations. The implementation of all pairs shortest paths has efficiency $O(|V|^2 \log |V| + |V||E|)$ based on Johnson's algorithm[9] but runs out of memory in the SeqAn implementation. This highlights the need for exploring parallel computation techniques to allow algorithms to run on commodity grids.

We expect new sequencing platforms will exponentially increase the number of mate pairs available for assembly. To take advantage these new tech-

Organism	Number of Mates	Number of Contigs	Module	Runtime
B. suis (1330)	16,850	178	CLK	2s
			Bundler	2s
			OrientContigs	3s
Acid Mine	85,839	24,869	CLK	45s
			Bundler	92s
			OrientContigs	100s

Table 1: The runtimes for each of the steps of the Bambus 2 pipeline without repeat detection. Run times were measured on Cygwin running on a 2GHz x86 machine with 2GB of RAM. Runtimes are presented in seconds unless otherwise indicated.

Organism	Number of Mates	Number of Contigs	Module	Runtime
B. suis (1330)			OrientContigs	6s
Acid Mine			OrientContigs	-

Table 2: The runtimes for each of the steps of the Bambus 2 pipeline with repeat detection. Run times were measured on Cygwin running on a 2GHz x86 machine with 2GB of RAM. Runtimes are presented in seconds unless otherwise indicated.

nologies, we will improve Bambus 2 performance by using approximation algorithms and distributed computing where possible.

5 Preliminary Results

NOT DONE. Show results for bacteria with repeats marked. Explain how repeats were marked appropriately. Mention problems with Acid Mine.

6 Proposed Work

NOT DONE. Describe that we can do least squares, better orientation, better motif finding.

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible.

First and foremost I'd like to thank my adviser, Professor Mihai Pop for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past four years. He has always made himself available for help and advice and there has never been an occasion when I have knocked on his door and he has not given me time. It has been a pleasure to work with and learn from such an extraordinary individual.

It is impossible to remember all, and I apologize to those I have inadvertently left out.

References

[1] AGP File Specification (v. 1.1).

[2] *AMOS Project Page. http://amos.sourceforge.net/.*

[3] *Celera Assembler Page. http://sourceforge.net/projects/wgs-assembler/.*

[4] Graphviz - Graph Visualization Software.

[5] Andreas, D., David, W., Tobias, R., and Knut, R. SeqAn An efficient, generic C++ library for sequence analysis. *BMC Bioinformatics 9*.

[6] Bentley, D. Whole-genome re-sequencing. *Current Opinion in Genetics & Development 16*, 6 (2006), 545–552.

[7] Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I., Belmonte, M., Lander, E., Nusbaum, C., and Jaffe, D. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Research 18*, 5 (2008), 810.

[8] Chaisson, M., and Pevzner, P. Short read fragment assembly of bacterial genomes. *Genome Research 18*, 2 (2008), 324.

[9] Cormen, T. *Introduction to Algorithms*. MIT Press, 2001.

[10] Edmonds, J., and Johnson, E. Matching: A Well-Solved Class of Integer Linear Programs. *LECTURE NOTES IN COMPUTER SCIENCE* (2002), 27–30.

[11] Fu, V., e. a. SOLiD System Sequencing and 2 Base Encoding.

[12] Goldberg, S., Johnson, J., Busam, D., Feldblyum, T., Ferriera, S., Friedman, R., Halpern, A., Khouri, H., Kravitz, S., Lauro, F., et al. A sanger/pyrosequencing hybrid approach for the generation of high-quality draft assemblies of marine microbial genomes. *Proceedings of the National Academy of Sciences 103*, 30 (2006), 11240.

[13] Herman, I., Melançon, G., and Marshall, M. Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS* (2000), 24–43.

[14] Margulies, M., Egholm, M., Altman, W., Attiya, S., Bader, J., Bemben, L., Berka, J., Braverman, M., Chen, Y., Chen, Z., et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature 437* (2005), 376–380.

[15] Miller, J. R., Delcher, A. L., Koren, S., Venter, E., Walenz, B. P., Brownley, A., Johnson, J., Li, K., Mobarry, C., and Sutton, G. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics Advance Access originally published on October 24, 2008. This version published October 28.* (2008).

[16] Myers, E., Sutton, G., Delcher, A., Dew, I., Fasulo, D., Flanigan, M., Kravitz, S., Mobarry, C., Reinert, K., Remington, K., et al. A Whole-Genome Assembly of Drosophila. *Science 287*, 5461 (2000), 2196.

[17] Pop, M., Kosack, D., and Salzberg, S. Hierarchical Scaffolding With Bambus, 2004.

[18] Rausch, T., Koren, S., Denisov, G., Weese, D., Emde, K. A., Döring, A., and Reinert, K. A consistency-based consensus algorithm for de novo and reference-guided sequence assembly of short reads. *Bioinformatics in review.* (2009).

[19] Roche. Genome sequencer flx data analysis software manual. *Roche Applied Science, Mannheim Germany* (2007).

[20] Rusch, D., Halpern, A., Sutton, G., Heidelberg, K., Williamson, S., Yooseph, S., Wu, D., Eisen, J., Hoffman, J., Remington, K., et al. The sorcerer ii global ocean sampling expedition: Northwest atlantic through eastern tropical pacific. *PLoS Biol 5*, 3 (2007), e77.

[21] Venter, J., Remington, K., Heidelberg, J., Halpern, A., Rusch, D., Eisen, J., Wu, D., Paulsen, I., Nelson, K., Nelson, W., et al. Environmental genome shotgun sequencing of the sargasso sea. *Science 304*, 5667 (2004), 66–74.

[22] Zerbino, D., and Birney, E. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research 18*, 5 (2008), 821.