The Pennsylvania State University
The Graduate School
College of Engineering

EXTENDING SAFE AND COMPLETE CONTIG ASSEMBLY TO NON-IDEAL DATA

A Thesis in
Computer Science and Engineering
by
John E. Hutton

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2018

The thesis of John E. Hutton was reviewed and approved* by the following:

Paul Medvedev
Assistant Professor of Computer Science and Engineering
Thesis Advisor

Kamesh Madduri
Assistant Professor of Computer Science and Engineering

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering
Professor in Charge of Graduate Program

*Signatures are on file in the Graduate School.

Abstract

In bioinformatics, genome assembly is a problem that, as yet, has no direct answer. Due to the way we are able to read DNA, we cannot reconstruct an original genome with 100% accuracy. Instead, we must read shorter segments of nucleotides. From this set of reads we attempt to assemble nucleotide strings that we know will be in the original genome. These contigs can then be used to produce a candidate assembly of the original genome. This method implies that we would like to get the longest contigs possible to use in the assembly process. Tomescu and Medvedev [1] considered this for a circular genome and developed the omnitig algorithm, which provides all possible contigs for a circular genome. However, the work focuses on a perfect case (i.e. the entire genome is represented by error free reads with no coverage gaps). This thesis extends the omnitig algorithm to work on a linear genome. The modified algorithm is then tested along with omnitig and two other algorithms to see how they react to increasingly non-ideal data. We look at a linear genomes in cases when 100% coverage is not achieved and in cases when reads contain errors. We find that the modified algorithm outperforms the other algorithms to varying degrees in terms of contig length and genome coverage. Additionally, we see that the extent of the benefit from the new algorithm is somewhat dependent on the quality of the input data.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Foremost, I would like to express my gratitude to my advisor, Paul Medvedev. I could not have completed this work without his guidance, insight, and encouragement.

Besides my advisor, I would like to thank Kamesh Madduri for his time and input in review of my thesis.

I would also like to thank Alexandru Tomescu and Massimo Cairo for their work on the code base for the omnitig as well as the unitig and Y-to-V algorithms. This was an invaluable starting point for the implementation of the linear algorithms in this thesis.

Finally, I would like to acknowledge Gary Gray for posting to the internet the Latex template used in creating this document, saving me hours of work on the formatting.

Chapter 1
Introduction

1.1 Background

Sequencing of a genome is an important and difficult problem. The basic problem stems from the fact that we cannot simply read an entire strand of DNA. Instead, we must break the DNA into many short segments and read the short segments, which we refer to as "reads". For the most popular sequencing instruments in use today, services can readily be found that can return reads from 50 to 300 base pairs (for example Illumina sequencing [2]). Considering that even a simple genome can be several million base pairs, the ratio of the read length to genome length is very small. The challenge becomes how to reassemble the reads into the original genome. In order to get the best assembly possible, we need the longest possible strings of nucleotides that are represented by the reads. A typical way of producing these "contigs" is to create a de Bruijn graph from the reads and return walks from the graph that we know will be in the genome.

1.1.1 The de Bruijn Graph

The de Bruijn graph stems a from 1946 paper [3], wherein de Bruijn described a method of graph construction by overlapping substrings of an original, longer string. The terminology has changed somewhat now that the graph has been adopted into use in genomics, but the idea is the same. We can create the graph by selecting k-mers (substrings of length k) from the original string. A k-mer will have an overlap of length $k-1$ with a neighboring k-mer. If we make each k-mer a node in a graph, we would place a directed edge between nodes that overlap by $k-1$ characters. For example, in the string ABC, we would have 2-mers AB and BC. The graph would have node AB connected by a directed edge to node BC.

1.1.2 Basic Problem

At its core, the problem addressed by this thesis is a graph theory problem. How can we get the longest strings (contigs) from our graph that are representative of the original string (genome)? We have three basic requirements to make any contig retrieval algorithm successful.

1. The contigs should be found in the original string.
2. The contigs should be as long as possible.
3. The contigs should cover as much of the original string as possible.

The first item is the idea of safety that will be use throughout this thesis. We will define specific applications of safety more precisely later on. The last item is somewhat equivalent to finding as many strings as possible that fit the first criterion.

In addition to the graph theoretical problem, we will also look into what happens when we put the theory into practice. That is, how well does the theory work when the data are not well behaved.

1.2 Related Work

We will use three algorithms against which we will test alongside our new, modified algorithm.

1.2.1 Unitig

The unitig algorithm returns walks from the de Bruijn graph in which the all nodes except the first and last have both in-degree and out-degree greater than one (see Figure 1.1). It is easy to see that any edge covering walk must traverse the unitig.

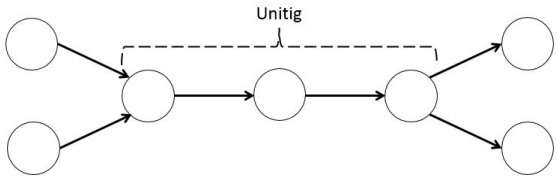


Figure 1.1: Unitig

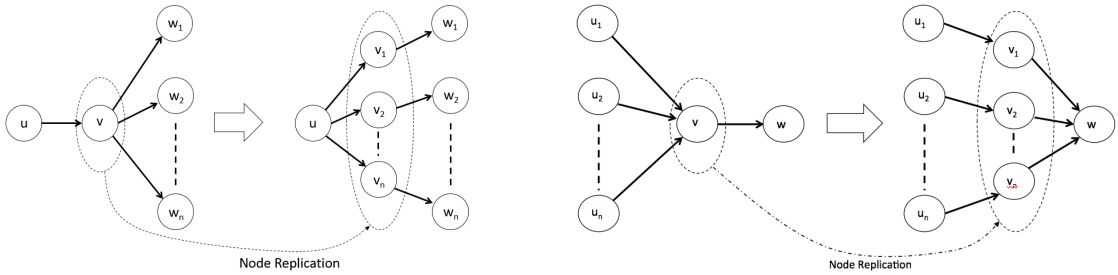
The unitig algorithm has the advantage that is easy to implement and fast. It is also easy to see that the results will be safe. However, the constraint imposed is such that we expect to miss vaild contigs in the graph.

1.2.2 Y-to-V

The Y-to-V algorithm attempts to improve upon unitig by modifying the graph. We look for nodes in the graph that have an in-degree of one and an out-degree of greater than one. For any node v with a single in-node u and n out-nodes $w_1 \dots w_n$, we will replicate v into n nodes $v_1 \dots v_n$. We remove v and the incident arcs from the graph and replace them with the replicated nodes and arcs $u \to v_i$ and $v_i \to w_i$ for $1 \leq i \leq n$.

Likewise we look for any node in the graph that has an in-degree of greater than one and an out degree of one. For any node v with n in-nodes $u_1 \dots u_n$ and one out node w, we replicate v into n nodes $v_1 \dots v_n$. We remove v and the incident arcs from the graph and replace them

with the replicated nodes and arcs $u_i \to v_i$ and $v_i \to w$ for $1 \leq i \leq n$. Figure 1.2 illustrates the Y-to-V transformation.



(a) One to Many (b) Many to One

Figure 1.2: Y-to-V Transformation

Once all candidate junctions have been split, we apply the unitig algorithm to the modified graph.

1.2.3 Omnitig

In [1], the authors investigated a circular genome. The algorithm developed not only returns safe contigs, but also returns all possible contigs for a given genome (a complete set). However, the algorithm is predicated on a perfect set of k-mers. That is, the de Bruijn graph is constructed from every k-mer in the original genome, and it is stipulated that there are no errors in the k-mers.

We can represent a walk in the graph as a sequence of nodes (v_i) and edges (e_i). The omnitig is formed by a walk $(v_0, e_0, v_1, e_1, ..., v_n, e_n, v_{n+1})$ in the graph such that for any $1 \leq i \leq j \leq n$ there exists no path of length greater than zero from v_j to v_i where the first edge is not e_j and the last edge is not e_{i-1}. Figure 1.3 illustrates this definition. We see that for the example walk labeled "Not Omnitig," the cycle in the graph allows a path from v_2 to v_1 that uses neither e_2 as the first edge nor e_0 as the last edge.

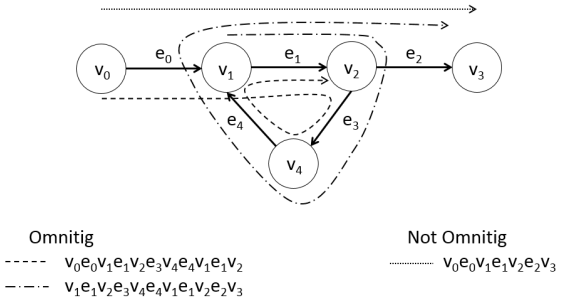


Figure 1.3: Omnitig

Acosta, Makinen, and Tomescu [4] futher investigated the case of when you may have several circular genomes in a set of reads. This lends some insight into the incomplete coverage case but is still predicated on complete coverage and circular genomes.

The omnitig algorithm is the starting point for the modified algorithms we will be investigating. The modified algorithms are an extension of the omnitig algorithm. We will refer to the new algorithms throughout this thesis as 'modified'.

1.3 Contribution

This thesis investigates increasingly difficult cases from the perfect, circular case. We will look at a linear genome with a perfect set of k-mers (i.e. every k-mer in the genome is represented in the set). We will then look at what happens if the read set suffers from incomplete coverage of the genome. Finally, we will investigate what happens if there are errors in the read set. Additionally, we will show that, for the linear case, a safe walk in a de Bruijn graph may not produce a string not found in the original genome.

Chapter 2
Methods

2.1 Basic Approach

We make the observation that a circular genome with incomplete coverage will look like a linear genome (or a collection linear genomes, depending on how bad the coverage is). Figure 2.1 illustrates this. The 'C' at 11 o'clock is not covered by any read, so the set of reads from the circular genome on the left could have come from the linear genome on the right.

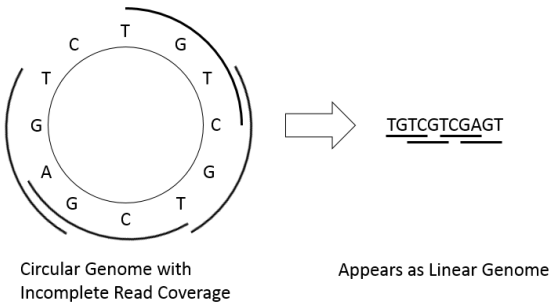


Figure 2.1: Incomplete Coverage

Since we know from [1, 5] how to find all safe contigs in a strongly connected graph, we will attempt to convert a connected graph into a strongly connected graph so we can apply the Omnitig algorithm. From the contigs obtained in this manner, we will then extract the contigs that apply to the linear genome.

2.2 Basic Linear Algorithm

2.2.1 Definitions

Let G be a directed graph. A walk is *edge-covering* if it visits every edge of the graph at least once. A *circular genomic reconstruction (CGR)* is an edge covering circular walk in G. Let s and t be two designated vertices in G, not necessarily distinct. A *linear genomic reconstruction*

(LGR) is an edge covering $s \to t$ walk in G. A walk w in G is *lin-safe* (respectively, *cir-safe*) iff for any linear (respectively, circular) genome reconstruction g, w is a subwalk of g. A lin-safe walk is *maximal* if it is not a sub-walk of another, longer walk which is lin-safe. A set W of walks is *lin-complete* (respectively, *cir-complete*) if for all w that are not a substring of a string in W, w is not lin-safe (respectively, cir-safe). The circular versions of these definitions correspond to the notions used in [1], while the linear versions are what we introduce in this thesis.

When we consider lin-safe walks, we will initially assume that the graph admits at least one LGR. Furthermore, we also assume that the vertices s and t are source and sink vertices, respectively. That is, s has no in-edges and t has no out-edges. This is without loss of generality, since an artificial source or sink vertex can always be added to the graph. Combined with the fact that there exists an LGR, we also know that s has exactly one out-edge and t has one in-edge. We use G^{core} to denote the graph obtained from G by removing s, t, and the two incident edges. We say that G is a *basic* graph if G^{core} is strongly connected.

2.2.2 Basic Graphs

We first present an Algorithm 1 to find lin-safe walks in basic graphs. We define G^{cir} as the circularization of G, that is, G plus an edge e_c from t to s. First, we prove that Algorithm 1 is safe (i.e. all the walks it returns are lin-safe in G).

Lemma 1. *Let G be a basic graph and let w be a cir-safe walk in G^{cir}. If w does not contain e_c, w is lin-safe in G.*

Proof. Assume for the sake of contradiction that there exists an LGR α of G that does not contain w. We can add the edge e_c to α to create a CGR β of G^{cir}. Since w does not contain e_c and is not a subwalk of α, it is not a subwalk of β. This contradicts the fact that w is cir-safe for G^{cir}. □

Theorem 1. *The set of walks returned by Algorithm 1 is lin-safe.*

Proof. There are three cases by which a walk is added to the output. The first case is when G_1 contains only one vertex. In this case, g is simply a path of two edges. The only LGR is that path, and, hence, the walk (e_0, e_1) is lin-safe. The second case is when w does not contain e_c. By Lemma 1, w is lin-safe. In the third case, w contains e_c. Both w_p and w_s are subwalks of w, and any subwalk of a cir-safe walk is also cir-safe. Hence, w_p and w_s are cir-safe walks of G^{cir}, and, by Lemma 1, they must be lin-safe in G. □

2.2.3 Extended Basic Graphs

We will define an *extended basic* graph as a graph that admits at least one LGR, but contains multiple strongly connected components (SCCs) in G^{core}. In this section, we give an algorithm for extended basic graphs. We start by describing the structure of G in terms of its SCCs.

Algorithm 1: Find lin-safe strings of a basic graph.

input : A basic graph G
output: A set of all maximal lin-safe walks in G

```
basicLinSafe(G)
    Let W ← ∅;
    if G^core is one node then
        Let e_0 be the edge leaving s;
        Let e_1 be the edge entering t;
        Add (e_0, e_1) to W;
    else
        foreach maximal cir-safe walk w in G^cir do
            if w does not contain e_c then
                Add w to W;
            else
                // w must contain e_c only once
                Let w_p be the prefix of w up to but not including e_c;
                Let w_s be the suffix of w beginning from but not including e_c;
                Add w_p and w_s to W;
    Remove any member of W that is a sub-walk of another member of W;
    return W;
```

Lemma 2. *Let G be a digraph with a designated source vertex s and sink vertex t that admits at least one LGR. Let n be the number of SCCs in G^{core}. Then, there exist vertices $s_1, \ldots, s_n, t_1, \ldots, t_n$ such that*

1. *s_i and t_i belong to the same SCC of G^{core}, for all $i \in [n]$.*
2. *There is a bijection between the vertices $s_1, \ldots, s_n$ and the SCCs of G^{core}.*
3. *The set of edges that span different SCCs of G^{core} is given exactly by $\{e_i = t_i \to s_{i+1} : 1 \leq i < n\}$.*

Proof. Let G_i be an SCC in G^{core}. Assume that G_i has more than one edge exiting the SCC. Let e_a and e_b be two exiting edges from G_i. Let α be an LGR in G. Let α cross e_a. α has now left G_i, but α still needs to cross e_b. Therefore, α must re-enter G_i. However, this implies that G_i must be part of a larger SCC and contradicts that it is its own SCC within G^{core}.

We can perform a topological sorting of the graph formed by the SCCs in G and label them $(G_i,\ 0 \leq i \leq n+1)$ in order. For any edge that connects two SCCs, G_i and G_{i+1}, we can then label the edge as e_i, the source vertex as t_i, and the destination vertex as s_{i+1}. □

This lemma implies a structure to all the LGRs of G. Let us define $e_0 = s \to s_1$ and $e_n = t_n \to t$ and name the SCCs of G^{core} as $G_1, \ldots, G_n$, according to the bijection of the Lemma. Then every LGR α of G must traverse each edge e_i, for $0 \leq i \leq n$, exactly once, in order of increasing i. Furthermore, the subwalk contained in between e_i and e_{i+1}, not inclusive, is an LGR of G_{i+1}.

Based on this structure, we can present our algorithm for extended basic graphs (Algorithm 2). The algorithm builds the set of lin-safe walks of G by first computing the lin-safe walks for all the SCCs, and then gluing together those that extend to the connector edges (the e_i, for $0 \leq i \leq n$). The glue function takes two walks w_p and w_s such that the last edge of w_p is the same as the first edge of w_s and returns the walk obtained by taking w_p and following it with w_s, with the shared edge included only once.

Algorithm 2: Find lin-safe strings of an extended basic graph

input : An extended basic graph G
output: A set of all maximal lin-safe walks in G

```
1  Find the strongly connected components in G^core and label the connector edges e_i and the
   SCCs G_i according to Lemma 2;
2  Let n ← the number of SCCs in G^core;
3  Let W ← ∅;
4  for i ← 1 to n do
5  |  Let G_i^ext be the basic graph created from G_i plus the edges e_{i−1} and e_i;
6  |  Add to W all the maximal lin-safe walks in G_i^ext;
7  for i ← 1 to n do
8  |  foreach walk w_p ∈ W that ends with e_i do
9  |  |  foreach walk w_s ∈ W that begins with e_i do
10 |  |  |  Remove w_p and w_s from set W;
11 |  |  |  Add glue(w_p, w_s) to W;
12 return W;
```

Next, we prove that Algorithm 2 is safe.

Theorem 2. *The walks returned by Algorithm 2 are safe.*

Proof. By Lemma 2, any LGR of G must contain a subwalk which is a LGR of G_i^{ext}, for all i. Therefore, a lin-safe walk of G_i^{ext} must be a lin-safe walk of G. This implies that the walks in set W after the iteration of the first for loop are safe.

Next, consider two lin-safe walks w_p and w_s where e_i is the last edge of w_p and the first edge of w_s. This is the case prior to the glue step in the second for loop. By Lemma 2, every LGR of G must contain e_i exactly once. Hence, since an LGR must contain both w_p and w_s, they must coincide on e_i and hence the glued walk must also be a subwalk of the LGR. □

Next, we prove that Algorithm 2 is complete.

Theorem 3. *Every safe walk in G is output by Algorithm 2.*

Proof. Let w be a safe walk in G. According to Lemma 2, w can be expressed as a series of walks $w_b, \ldots, w_e$, where (1) each w_i is contained in G_i^{ext}, and (2) w can be obtained by gluing w_b to w_{b+1} and so on until w_e. Note that we cannot really have a safe walk that needs more than one glue unless there is a trivial G_i in the walk.

First, we show that w_i is lin-safe for G_i^{ext}.

A walk is safe iff all subwalks are also safe. Therefore, since w_i is a subwalk of w, it must be safe, or it would contradict the safety of w.

Second, we show that w is obtained in the second for loop.

If w is wholly contained in some G_i^{ext}, it would be added to W in line 6. Otherwise, there exists an e_i such that $e_i \in w$, and we can write w as $w'e_iw''$. In this case, $w'e_i = w_i$, and $e_iw'' = w_{i+1}$.

Observe that any single edge is safe. If we build an edge covering walk by continually adding a single edge, we must either complete our edge covering walk with a safe walk, or we must, at some point, add a single edge that makes the walk unsafe. We can view the walk as a series of overlapping edge pairs. Every consecutive pair along a safe walk must be safe. When the edge is added that makes the walk unsafe, the last edge pair is unsafe. If we never add an unsafe pair, the walk remains safe.

Let p_ap_b be an edge pair that is a subwalk of w. If $p_a \neq e_i$ and $p_b \neq e_i$, the p_ap_b must be in either w_i or w_{i+1} and is therefore safe. If $p_a = e_i$, then $p_ap_b \in w_{i+1}$, and p_ap_b is safe. If $p_b = e_i$, then $p_ap_b \in w_i$, and p_ap_b is safe. Since all consecutive edge pairs in w are safe, w must be safe.

□

2.2.4 Simplified Algorithm

As a practical matter, we can simplify the algorithm from Algorithm 2. Instead of treating each SCC in turn and gluing the results together, we could apply a single circularizing edge to the extended basic graph. We add e_c from t to s. This creates a single SCC from which we can find all cir-safe walks. We then need to exclude the walks containing e_c as we have done above. The logic in Lemma 1 can be applied here to tell us that all cir-safe walks will be lin-safe, so we would not have introduced erroneous walks. The simplification is shown in Algorithm 3.

Algorithm 3: Find lin-safe strings of an extended basic graph (simplified)

input : An extended basic graph G
output: A set of all maximal lin-safe walks in G

```
1 Identify the source node s and the sink node t;
2 Let e_c be a new edge from t to s;
3 Let G^cir ← G ∪ e_c;
4 Let W ← the set of all cir-safe walks in G^cir;
5 foreach walk w ∈ W do
6     if e_c ∈ w then
7         Remove w from set W;
8 Remove all walks from W that are a subset of a longer walk in W;
9 return W;
```

2.3 General Graphs

To this point, we have only considered graphs with a single source node and a single sink node. Now let us consider a graph G like the extended basic graph above except that, instead of a single source node s and a single sink node t, we will have a source SCC S (i.e. S has no in-edges) and a sink SCC T (i.e. T has no out-edges). In this arrangement, S and T may contain more than one vertex. In this case, we can use a similar approach to the extended basic graph, adding e_c from T to S. However, we have to make a decision about which nodes within S and T will be connected with the circularizing edge. Likewise, we may have a graph with multiple source and/or sink SCCs, and we will have to decide how to circularize these graphs.

2.3.1 Non-trivial Source Sink SCCs

For any graph, we will identify a source SCC as an SCC with no inbound edges. Likewise, we will identify a sink SCC as an SCC with no outbound edges. Consider a graph G that, has a single source SCC and a single sink SCC. Additionally, let us require, as above, that G admit an LGR. For such a graph with n internal SCCs, we will label the SCCs in topological order as G_0 to G_{n+1}. The definition of an LGR can remain the same as above. Note that the designated vertex s could be any vertex in G_0, and the designated vertex t could be any vertex in G_{n+1}.

As with the internal strongly connected components, G_0 will have a exit node t_0, and G_{n+1} will have a entry node s_{n+1} as described in Lemma 2. We can circularize the graph by adding a pseudo edge from s_{n+1} to t_0. If we remove all nodes except t_0 from the source SCC and all nodes except s_{n+1} from the sink SCC, we have an extended basic graph. From Lemma 1 we see that all walks in this graph that are cir-safe in the circularized graph are also lin-safe in the original graph.

We observe that an LGR in G could begin anywhere in G_0 and end anywhere in G_{n+1}. However, any LGR can be extended to begin with t_0 and end with s_{n+1}. If we stipulate this, we can create G^{cir} by adding the circularizing edge from s_{n+1} to t_0. Applying logic of Lemma 1 and see that all walks in that are cir-safe in G^{cir} are also lin-safe in G.

If the LGR were to have ended at some other node $a \neq s_{n+1}$ in G_{n+1}, there would still have to be a path from $a \rightsquigarrow s_{n+1}$ in the LGR, or it would not be edge covering. Therefore, extending the LGR at most simply adds redundancy to some edges being crossed, and does not add or remove any path from what must be traversed. Likewise for the source, the LGR could begin at any node $b \neq t_0$ within G_0. However, there must still be a possible LGR with a path from t_0 to b. This path would then continue back to t_0 so it can exit the SCC.

2.3.2 Multiple Source and Sink SCCs

At this point, we will enter the realm of heuristics. When constructing a graph from reads, we may have incomplete coverage or errors. This can lead to the case where, after topologically sorting the graph and identifying the source and sink SCCs, there may be more than one source and/or sink SCC (for example, Figure 2.2). Note that when this happens, the graph will not

admit an LGR. That is, no single walk can cover all edges. Instead, the best we can do is provide a set of walks that is edge covering in G. We will define a *Collective Linear Genomic Reconstruction (CLGR)* as a set of edge covering s to t walks for any node s in a source SCC and any node t in a sink SCC. We will say that a walk w is *collectively lin-safe (clin-safe)* if, for any CLGR C, w is a subwalk of some walk in C.

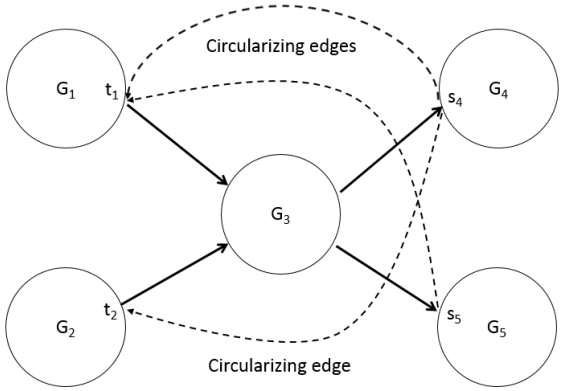


Figure 2.2: Multiple Sources/Sinks

We will continue with an extension of the method above. Our overall goal is to circularize the graph. One easy way (conceptually) to circularize the graph is to connect each sink SCC to each source SCC. This can quickly create a proliferation of circularizing edges. In practice, this was not a problem on the data sets used in Chapter 3. However, when we introduce error, the number of edges became a problem both in execution time and in memory, sometimes consuming over 80GB to store the edges.

As an alternative. we can choose an entry node a_1 from the set of sink SCCs. We will connect this node to every node b that is a source SCC exit node. Likewise, we will choose an source SCC exit node b_1. We will connect every node $a \neq a_1$ that is a sink SCC entry nodes to b_1. The result will be a transformed graph G^T that is a single strongly connected component. From here, we can find the cir-safe walks in the circularized graph G^T. We must then break each contig that crosses any of the circularizing edges.

In addition to multiple sources/sinks, incomplete coverage can also lead to a graph that consists of multiple connected components. When this happens, we simply need to treat every connected component in turn.

The generalized algorithm for any given CC is shown in Algorithm 4. This version uses the second, more efficient, method of circularizing the graph. It should be noted, however, that the more edge efficient method can yield different results for graphs with many source and sink SCCs.

Algorithm 4: Find collectively-lin-safe strings of a graph G.

input : A directed graph G
output: A set of maximal collectively-lin-safe walks in G

Find the Strongly Connected Components in G;
Let S be the set of all source SCCs in G;
Let T be the set of all sink SCCs in G;

`// Create a circularized graph` G^T
Let $G^T \leftarrow G$;
Let $E \leftarrow \emptyset$;
Select an entry node a_1 from an SCC $\in T$;
foreach *exit node* b *in* S **do**
    Add to G^T a circularizing edge e_c from a_1 to b;
    Add e_c to E;
Select an exit node b_1 from an SCC $\in S$;
foreach *enty node* $a \neq a_1$ *in* T **do**
    Add to G^{cir} a circularizing edge e_c from a to b_1;
    Add e_c to E;
Let $C \leftarrow$ the set of all maximal cir-safe walks in G^T;
foreach *walk* $w \in C$ **do**
    if $w \in E$ *of the form* $w_p e_c w_s$ **then**
        Remove w from C;
        Add w_p and w_s to C;
Remove any member of C that is a sub-walk of another member of C;
return C;

Chapter 3
Ideal Data

3.1 Introduction

In this chapter, we will look at ideal data. That is, the data will be guaranteed to be error free, and we impose a constraint that reads are only along the forward strand. In the next chapter we will investigate non-ideal data.

For all test scenarios, four algorithms are run:

1. Unitig
2. Y-to-V
3. Omnitig (NM)
4. Modified

The results from these tests are compared within the following metrics:

1. Number of contigs found that are in the original genome
2. Number of contigs found that are not the original genome
3. Average length of the contigs
4. E-size of the set of contigs
5. Coverage

For E-size, we use the notion from [6]. The E-size is designed to answer what the expected size would be of a contig containing a randomly chosen base in the genome. The E-size formula is

$$E\text{-}size = \sum_{contig} \frac{{L_{contig}}^2}{L_{genome}}$$

where L_{contig} is contig length, and L_{genome} is genome length. Since, for our testing, we know the actual genome length, we are not required to use an estimate as in [6].

For coverage, we will use the percent of bases in the reference genome that are covered exactly by at least one contig. For this we require that the contig is a substring of the reference (i.e. there is an exact match for the contig within the reference).

Testing is approached in incremental stages. Note that all runs are made as a linear genome. This is somewhat undefined in the omnitig algorithm's case, as the omnitig algorithm is predicated on a circular genomic reconstruction. Therefore, we may expect some abnormal behavior for the unmodified omnitig algorithm. The genomes considered are:

1. Toy genome of length 15
2. Fragment genome of length 110
3. *E.coli* complete genome

Tests are first run on the genome in sequence. By this, we mean that the de Bruijn graph is created by taking all k-mers from the given genome. This is equivalent to taking reads on the forward strand with perfect coverage (i.e. for every base pair in the genome, there exists a read that starts at that base pair). For the complete *E.coli* genome, we additionally run tests with decreasing read coverage (40x, 20x, 10x, and 5x).

3.2 Toy Genome

First, a very small toy genome is run (TGTCGTCGAGTCGAA). This toy genome is small enough that the de Bruijn graph can be hand built for comparison. For this test, we use a k-mer size of 2 and build the graph from all k-mers in the genome (see Figure 3.1). Table 3.1 shows the contigs returned by each algorithm. As expected, all algorithms return contigs that are found in the genome. That is, no erroneous contigs were returned. We can see that all unitigs and Y-to-V contigs are substrings of the modified contigs. Additionally, we notice that the omnitig algorithm misses the contig that begins at the source node. We should note that the omnitig algorithm is predicated upon a graph that is a single strongly connected component, so it is not altogether surprising that it would make some errors when this is not the case. However, we will see that this failure to detect a trivial source will have some other ramifications in later results. All algorithms except omnitig give 100% coverage.

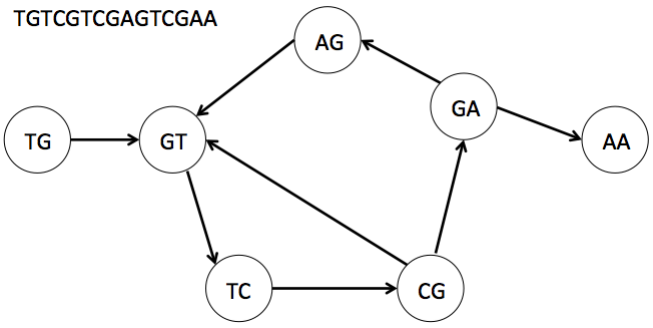


Figure 3.1: Toy Genome de Bruijn Graph (k = 2)

#	Contig	Unitig	Y-to-V	Omnitig	Modified
1	TGT	x	o		o
2	TGTC		x		o
3	TGTCG				x
4	CGT	x	o	o	o
5	TCGTC		x	o	o
6	GTCGTCGA			x	x
7	CGA	x	o	o	o
8	GAA	x	o	o	o
9	TCGAA		x	o	o
10	GTCG	x		o	o
11	GTCGAA			x	x
12	GAGT	x	o	o	o
13	TCGAGTC		x	o	o
14	GTCGAGTCG			x	x

Table 3.1: Toy Genome Contigs (k = 2)
x ≡ contig is found by the algorithm. o ≡ contig is found as a substring of a longer contig.

3.3 Fragment Genome

Next, a slightly larger, but still tiny, fragment genome is used. The fragment genome is the first 109 base pairs of *E.coli* with a 'T' pre-pended. The reason for the pre-pended 'T' will become apparent later in Section 3.5.

In this genome, a k-mer size of 5 produces the graph in Figure 3.2. The contigs are given in Table 3.2. We see again that contigs found by unitig and Y-to-V are substrings of the modified algorithm. We also note that omnitig again misses the contig that begins at the source node (contig 1). However, more troubling is contig 14. This contig is reported only by the omnitig algorithm. Although the contig is in the original genome, it is not a safe walk in the graph. We see in Figure 3.2 that the contig is formed by a walk through the nodes $13 \to 4 \to 16 \to 3 \to 14 \to 13$. However, we can create an LGR that does not use the walk $4 \to 16 \to 3$. This walk would have been safe if the source node did not exist, so again omnitig's inability to see the single node source causes issues.

From Table 3.3 we see that unitig and Y-to-V return similar results with respect to average length and E-size. Omnitig and modified both outperform the other two, and modified slightly outperforms omnitig. We see that the average contig length for modified goes down from omnitig, but this is largely due to the short contig (1) that is not found by omnitig. The E-size shows a significant increase for the modified algorithm.

We can also see that contigs 17 and 18, reported by unitig, Y-to-V, and omnitig, were combined by modified to produce a longer contig (#19).

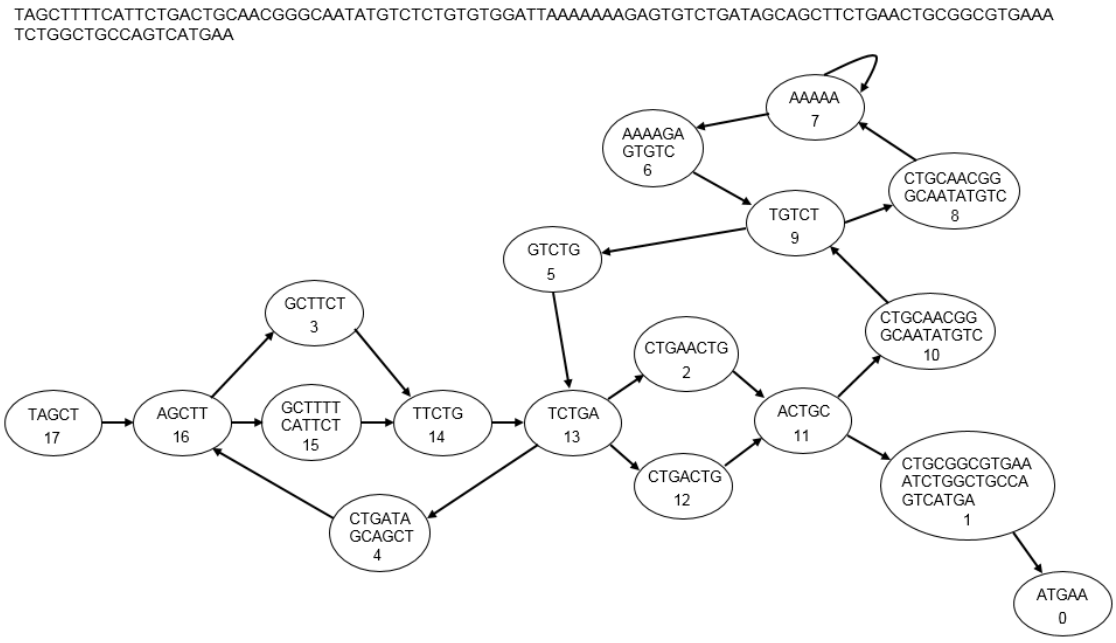


Figure 3.2: Fragment Genome de Bruijn Graph (k = 5)

3.4 *E.coli* Genome

Recognizing that *E.coli* is a circular genome, we will treat it as a linear genome for testing purposes. The accession number for the reference genome is NC_000913 [8]. For these tests, a k-mer size of 31 is used, which is consistent with [1]. Each coverage level was tested three times with a different simulated read set. Table 3.4 shows the average results from the runs at different coverages. Note that the first entry in the table is for perfect coverage, by which we mean that all k-mers in the genome are used (the method used for the toy and fragment genomes), as opposed to extracting k-mers from reads. A more comprehensive list of *E.coli* run results is given in the Appendix.

We note that, for these results, we did not use the edge efficient form of the general algorithm. The edge efficient algorithm yielded over 100 erroneous contigs for the 5x coverage case. The reason for this bears further investigation, but the indication is that the less edge efficient method is superior if we can get it to complete. For comparison, we also provided results using the gluing method as in Algorithm 2. That is, each SCC is treated individually, applying the glue method within a CC versus treating each as in Algorithm 4. This is shown as 'Modified g' in the table. For the high coverage cases, all versions of the modified algorithm perform identically. This is as expected, or it would indicate a problem with the theory.

We expect that, as the coverage diminishes, we will be unable to build a de Bruijn graph that represents the entire genome. Table 3.5 shows the components of the graph for each coverage case. We can see that at 10x coverage, the components of the graph begin to increase. This indicates that we have significant gaps in coverage, and in the best case, any contigs from the

#	Contig	Unitig	Y-to-V	Omnitig	Modified
1	TAGCTT	x	x		x
2	AAAAAA	x	x	o	o
3	AAAAAGAGTGTCT	x	x	o	o
4	AAAAAAGAGTGTCT			x	x
5	TTCTGA	x	o	o	o
6	AGCTTTTCATTCTG	x	o	o	o
7	AGCTTTTCATTCTGA		x	x	x
8	TGTCTGA	x	x	o	o
9	AAAAAGAGTGTCTGA			x	x
10	TCTGACTGC	x	x	x	x
11	AGCTTCTG	x	o	o	o
12	AGCTTCTGA		x	o	x
13	TCTGATAGCAGCTT	x	x	o	x
14	TCTGATAGCAGCTTCTGA			x	
15	TCTGAACTGC	x	x	x	x
16	ACTGCGGCGTGAAATCTGGCTGCCAGTCATGAA	x	x	x	x
17	TGTCTCTGTGTGGATTAAAAA	x	x	x	o
18	ACTGCAACGGGCAATATGTCT	x	x	x	o
19	ACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAA				x

Table 3.2: Fragment Genome Contigs (k = 5)
x ≡ contig is found by the algorithm. o ≡ contig is found as a substring of a longer contig.

Algorithm	# Contigs	Avg Len	E-size
Unitig	13	12.9	26.7
Y-to-V	12	13.7	26.8
Omnitig	9	17.3	28.4
Modified	10	16.2	32.7

Table 3.3: Contig Statistics from Fragment Genome (k = 5)

data look like several non-overlapping fragments. When coverage drops to 5x, the number of components explodes. At this point, we would expect more difficulty in extracting information from the data.

Consistent with the observation of graph components, we see that results from the 40x and 20x coverages are reasonably similar. An interesting result is that the omnitig algorithm finds a few contigs that are not in the original genome. We see that the average contig length is much larger than for unitig. Average length is comparable to omnitig, though we find a few more contigs and have a slightly better E-size. More importantly, the modified algorithm does not return any erroneous contigs.

For 10x coverage, we see that omnitig returns several erroneous contigs. We also note that modified maintains a good E-size, while other algorithms begin to suffer a noticeable decrease.

When coverage drops to 5x, all algorithms suffer a large performance hit. All algorithms but unitig report erroneous contigs. We will examine the erroneous contigs in the next section. We

Coverage Depth	Algorithm	# Good Contigs	# Bad Contigs	Reference % Covered	Average Length	E-size
Perfect	Unitig	1749	0	100	2645	32830
	Y-to-V	1007	0	100	4671	33301
	Omnitig	985	0	99.88	4771	33796
	Modified	987	0	100	4817	34910
	Modified g	987	0	100	4817	34910
40x	Unitig	1740	0	100	2645	32830
	Y-to-V	1007	0	100	4671	33301
	Omnitig	983	2	99.88	4818	34904
	Modified	987	0	100	4817	34910
	Modified g	987	0	100	4817	34910
20x	Unitig	1740	0	100	2585	32830
	Y-to-V	1007	0	100	4671	33301
	Omnitig	984	0.7	99.88	4788	34165
	Modified	987	0	100	4817	34910
	Modified g	987	0	100	4817	34910
10x	Unitig	1774	0	99.99	2608	29652
	Y-to-V	1029	0	99.71	4527	29690
	Omnitig	967	16.7	82.78	4330	26459
	Modified	1010	0	99.02	4661	31056
	Modified g	1027	0	99.99	4847	33091
5x	Unitig	2790	0	99.38	1651	6100
	Y-to-V	1212	5.3	38.78	1538	2452
	Omnitig	962	5	23.02	1153	1388
	Modified	1176	5.3	37.73	1548	2401
	Modified g	2510	5.3	99.32	2048	6801

Table 3.4: Contig Statistics from *E.coli* Genome (k = 31)

Coverage	# SCC	# CC	# Src SCC	# Sink SCC
Perfect	5	1	1	1
40x	5	1	1	1
20x	5	1	1	1
10x	118	3	26	26
5x	4386	878	1077	1080

Table 3.5: Graph Components from *E.coli* Genome (k = 31)

also see that the unitig goes from worst to best in both average length and E-size. An interesting result is that the gluing method greatly outperforms the non-gluing method of the modified algorithm. Although we would expect some difference due to the difference in the heuristic, it is surprising to see such a large difference. This result bears further investigation.

Although it is not the point of this thesis to examine efficiency, we note that for perfect coverage, all algorithms complete in under one second. From Table 3.5, we see that, for perfect coverage, the graph consists of only one connected component and very few strongly connected components. When the coverage drops to 20x, the time for the modified algorithm stays under one second, but it is roughly three times the perfect coverage case. When including output time for results, the total time for the 5x coverage case is around five seconds.

#	Contig	Unitig	Y-to-V	Omnitig	Modified
1	AAAAAA	x	x	o	o
2	AAAAAGAGTGTCT	x	x	o	o
3	AAAAAAGAGTGTCT			x	x
4	TTCTGA	x	o	o	o
5	AGCTTCTG	x	o	o	o
6	TCTGATAGCAGCTTCTGA		x	x	x
7	TGTCTGA	x	x	o	o
8	AAAAAGAGTGTCTGA			x	x
9	TCTGACTGC	x	x	x	x
10	TCTGAACTGC	x	x	x	x
11	TGTCTCTGTGTGGATTAAAAA	x	x	x	o
12	ACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAA				x
13	ACTGCAACGGGCAATATGTCT	x	x	x	o
14	TCTGATAGCAGCTT	x	o	o	o
15	AGCTTTTCATTCTG	x	o	o	o
16	TCTGATAGCAGCTTTTCATTCTGA		x	x	x
17	TGTCTGATAGCAGCTT				x
18	ACTGCGGCGTGAAATCTGGCTGCCAGTCATGAA	x	x	x	x

Table 3.6: Problematic Fragment Genome Contigs (k = 5)
x ≡ contig is found by the algorithm. o ≡ contig is found as a substring of a longer contig.

3.5 Problematic Graphs

If we return to the *E.coli* fragment genome without the pre-pended 'T' and a k-mer size of 5, we will get the results in Table 3.6. We see that Y-to-V, omnitig, and modified all return a contig (#16) not found in the original genome. If we look at the de Bruijn graph (Figure 3.3), we see that the erroneous contig is created by a walk through node 16. Node 16 is the k-mer that begins the genome. Because this k-mer also appears later in the genome, the resulting graph has an input node into node 16 (i.e. node 16 is not a trivial source to the graph). The erroneous contig has a suffix (from the start k-mer, shown in the first dashed box) and a prefix (to the start k-mer, shown in the second dashed box) that are in the genome, but the entire contig does not appear in the genome. This problem can occur whenever the k-mer that begins the genome appears again later in the genome.

If we could demand that all LGRs start at node 16, the problem would go away, because only the suffix of the erroneous contig would remain safe. Therefore, if we were to apply a pseudo edge into node 16, the problem disappears. However, we cannot know a priori that node 16 is where we should begin. Note that we would also lose contig 6 in this case, as it would be also broken into its prefix and suffix.

One solution to the safety problem would be to examine each node in s. Any node that has a single input edge and multiple output edges is a candidate for the problem (see Figure 3.4). If we apply a pseudo edge to each of these candidates, we make any traversal of the node unsafe, and eliminate the problem. However, this comes at the expense of eliminating the valid contigs

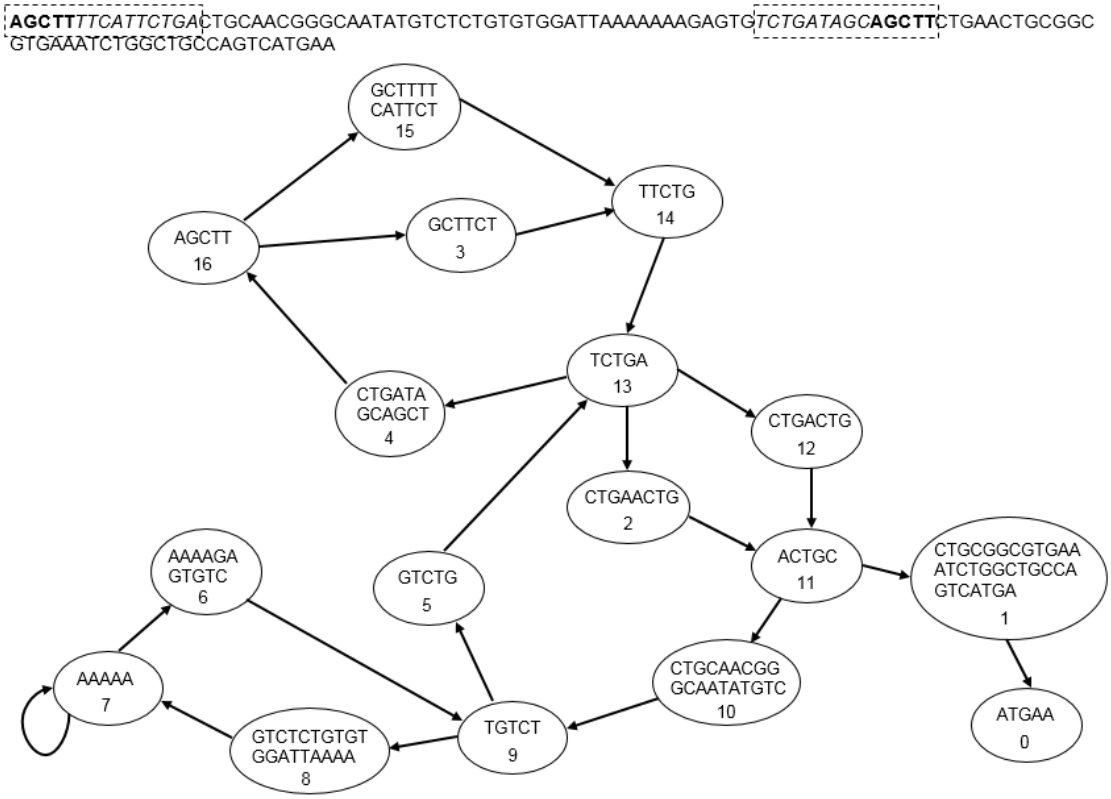


Figure 3.3: Problematic Genome de Bruijn Graph (k = 5)

that traverse the node. It also increases the proliferation of edges, which inmpacts run time.

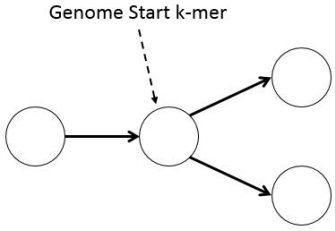


Figure 3.4: Potentially Problematic Configuration

Conversely, we could accept the chance of the problem occurring. A longer k-mer size can reduce the chance of the starting k-mer repeating later. In the case of the *E.coli* reference used in this thesis, a k-mer size of 11 will exhibit the problem, but a k-mer size of 12 will not. In the case of our problematic fragment genome, if we use a k-mer size of 6, we get the graph shown in Figure 3.5. The k-mers are shown in Table 3.7. We see that the erroneous contig disappears. This is because the starting k-mer "AGCTTT" does not occur again in the genome. This causes the k-mer to be in a source node for the graph rather that merely reside in a source component, where there would be an input edge to the node containing the k-mer.

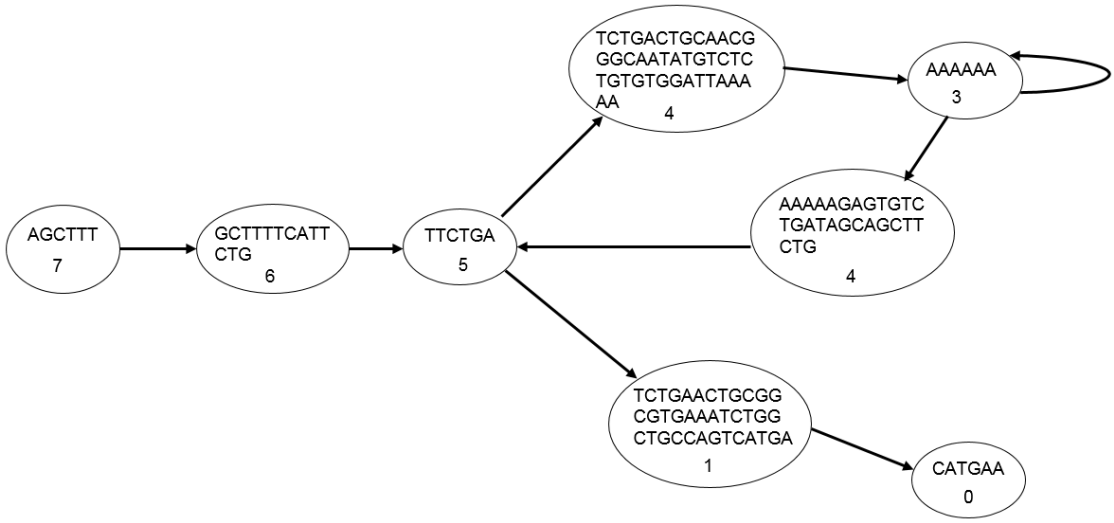


Figure 3.5: Problematic Genome de Bruijn Graph (k = 6)

#	**Contig**	**Unitig**	**Y-to-V**	**Omnitig**	**Modified**
1	AAAAAAA	x	x	o	o
2	AAAAAAGAGTGTCTGATAGCAGCTTCTGA	x	x	o	o
3	AAAAAAAGAGTGTCTGATAGCAGCTTCTGA			x	x
4	AGCTTTTCATTCTGA	x	x		o
5	TTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAA	x	x	x	o
6	AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGAT TAAAAAA				x
7	TTCTGAACTGCGGCGTGAAATCTGGCTGCCAGTCATGAA	x	x	o	o
8	AAAAAAGAGTGTCTGATAGCAGCTTCTGAACTGCGGCGTGAAAT CTGGCTGCCAGTCATGAA			x	x

Table 3.7: Problematic Fragment Genome Contigs (k = 6)
x ≡ contig is found by the algorithm. o ≡ contig is found as a substring of a longer contig.

Note that for all algorithms the contig is a safe walk in the graph. However, the graph itself can admit walks that are not safe with regard to the genome.

If we treat the genome as a random sting of nucleotides, we would expect the probability of a match for a k-mer of length k at any given position to be

$$\frac{1}{4^k}$$

Therefore, the probability of not finding a match over an entire genome of length L would be

$$(1 - \frac{1}{4^k})^{L-k}$$

For *E.coli* with $k = 11$, this gives a probability of 0.67. For $k = 12$, this gives 0.24. For $k = 21$, the probability drops to $1.1 \cdot 10^{-6}$, and for a $k = 31$, the probability drops to $1.0 \cdot 10^{-12}$. For something the length of human chromosome 10 with $k = 21$, the probability of a collision is $3.0 \cdot 10^{-5}$. We can see that a longer genome increases the probability of a repeat, but a larger k-mer size has a greater affect in decreasing the probability.

In reality, we know that a genome is not a random string. For an investigation of probabilities with regard to actual genomes, we refer the reader to [9], where the authors investigate matching a fingerprint, which is a substring of a genome, to a set of clones.

Chapter 4
Non-Ideal Data

Up to now, we have considered data in an idealized case. Even when we looked at poor coverage, we still guaranteed error free data as well as data only on the forward strand. Now we will look at more realistic cases. We will begin with simulated data, using wgsim [10] to generate the data. Initially error free data will be simulated. Then we will introduce errors. Ultimately, we will look at two read sets from the Short Read Archive [11] [12].

Note that in these results, the modified algorithm refers to the gluing method (i.e. treating each SCC individually). This is due to the method of Algorithm 4 having trouble completing in all cases. Additionally, in cases where both versions did complete the results were very similar, unlike for the 5x coverage in the previous chapter. The gluing method typically takes much longer to run, but has a better chance of completing without crashing due to memory problems. Note that all tests were run on a system with 128GB of RAM.

4.1 Data Generation

4.1.1 Wgsim Data

The wgsim data was created to follow the idealized data presented in the previous chapter. Both the non-errored data set has 250 base pair reads at a coverage rate of 40x. Wgsim produces a paired read set, and we will look at the forward read file. This has the effect of halving our coverage, but we note from the previous results that 20x coverage is enough to produce a good graph. The error free data set will be called "wgsim clean" in the results below.

The errored data set was created using the default error rates from wgsim. In this case, enough reads (250 bp each) were created to produce 400x coverage. These reads were then culled by looking for 50-mers with an abundance ≥ 10. The surviving 50-mers were fed into each algorithm for processing. The errored data will be called "wgsim default" in the results below.

Read Set	# SCC	# CC	# Nodes	# Edges
wgsim clean	9	1	6261	7778
wgsim default	31719	757	99131	116932
SRR4048497	635638	181449	635638	455421
SRR4048496	768025	175222	768025	593926

Table 4.1: Graph Components from *E.coli* Genome (k = 31)

4.1.2 Short Read Archive Data

The final tests are run on data from the Short Read Archive. The two data sets SRR4048496 [11] and SRR4048497 [12]. These both contain single end reads of length 36. SRR4048496 contains 8,556,356 reads, and SRR4048497 contains 3,117,170 reads. Data from SRR4048497 was error corrected with musket [13]. Data from SRR4048496 was corrected by culling all reads with an abundance of only 1.

4.2 Results

4.2.1 Graphs

Table 4.1 shows the characteristics of the de Bruijn graphs created from the input data. The graphs from the wgsim clean data look much like those from the ideal data. Note that the number of strongly connected components should be expected to be about twice as for the ideal. Since the read may come from either the forward or the reverse strand, there will be basically two graphs –one from each direction of read. We might also expect twice as many connected components, but this doesn't seem to be the case. This indicates there may be a crossover between the forward and reverse graphs. That is, at least one k-mer is common between the forward and reverse reads.

When we introduce errors, however, the number of graph components increases dramatically. This tells us that, unsurprisingly, we should not expect to retrieve results as good as in the ideal case. Note that in the two SRA data sets, each SCC is a single node. This may be due in part to over or under correction of the data.

We also see that, for the SRA data, there are fewer edges than nodes, also indicating poor input data quality. Therefore, we should expect relatively short contigs.

4.2.2 Contig Statistics

We have added a metric that we did not consider in the ideal case. For the ideal data, we looked at percent of reference covered, requiring exact matches. For data that are not error free, this measure may not be as informative, since a contig that has a single base error would be excluded from consideration. Therefore, we will also look at the alignment percentage. That is, we will see what percent of the contigs align to the reference. For the alignment percentage, we have

Read Set	Algorithm	# Good Contigs	# Bad Contigs	Ref % Covered	Ref % Aligned	Average Length	E-size
wgsim clean	Unitig	4181	0	100	98.82	2210	54772
	Y-to-V	2384	0	100	99.63	3931	55291
	Omnitig	2345	0	99.99	99.60	3992	56480
	Modified	2348	0	100	99.62	4038	58472
wgsim default	Unitig	31001	35140	54.10	99.93	181	4877
	Y-to-V	14886	35232	44.36	99.98	705	27920
	Omnitig	10351	31198	44.51	99.98	846	28122
	Modified	14998	50168	44.54	99.98	646	28012
SRR4048497	Unitig	22223	218731	13.21	17.76	35	65
	Y-to-V	2487	45176	2.55	2.63	39	17
	Omnitig	1765	37339	1.79	2.21	40	15
	Modified	24097	244123	13.21	17.81	36	73
SRR4048496	Unitig	28162	306998	16.34	16.95	35	88
	Y-to-V	2149	119929	1.46	1.63	38	39
	Omnitig	1717	99337	1.28	1.49	38	33
	Modified	29308	360727	16.34	16.97	35	105

Table 4.2: Contig Statistics From Non-ideal Data

used the idea from ContigValidator [7]. The alignment percentage can be considered as a looser measure of the percentage of contigs that match exactly to the reference.

Table 4.2 shows the contigs returned by each algorithm. In this case we present the good contig count includes both forward and reverse strand matches. The reference percent covered is figured by laying exactly matching contigs over the reference genome. In this case, a contig may be overlaid more than once. The reference percent aligned is similar, but uses the alignment start and length from a bwa mem [14] alignment to determine the reference nucleotides matched to a contig. In this case, a contig can only be used once, which could make the coverage percentage come out below the exact match percentage.

We note that the SRA data contains many nucleotides coded as 'N'. This has several side effects. One, it can cause two potentially equal kmers to be unequal. Two, it can lead to contigs also containing 'N's. When figuring the contig matches, we remove the leading and trailing 'N's from the contig. Internal 'N's were left in place, which will decrease the number exact matches, but will preserve length and, hopefully, leave longer contigs that still align.

Somewhat surprisingly, the omnitig algorithm performed the best on the wgsim default data set with respect to average length and E-size. This may be due to some peculiarity with that data set. Further investigation into this is warranted. We see for the other data sets, that omnitig does not outperform the others. For exact match coverage on wgsim default, we see that unitig does best. However, when considering aligned coverage, it does worst. This is likely do to contigs that are very short compared to the other contigs. Shorter contigs have a better chance at an exact match. In the extreme case, we could return four contigs of length 1 (one for each nucleotide) and guarantee 100% exact match coverage every time.

We also see that all algorithms perform poorly for the SRA data. This is unsurprising, considering how disjoint the graph is. We see that unitig seems to outperform Y-to-V and omnitig.

Read Set	Unitig		Y-to-V		Omnitig		Modified	
	Time	Mem	Time	Mem	Time	Mem	Time	Mem
wgsim clean	1s	40	1s	45	1s	41	1s	40
wgsim clean	1s	62	1s	105	32s	80	19m	11900
SRR4048497	1s	227	2s	234	1s	226	1h 12m	626
SRR4048496	1s	276	3s	299	1s	274	2h 44m	549

Table 4.3: Runtime and Memory Usage (MB)

We also see that the modified algorithm yields equal or better genome coverage compared to all other algorithms. The modified algorithm gives an average length similar to unitig, but returns more contigs. This results in a somewhat larger e-size. On the downside, the algorithms all return more erroneous contigs than exact matches for the SRA data. This may be due in part to the large number of 'N's in the reads.

It should also be noted that in the case of the SRA data, we have reads from DNA that will not exactly match the reference. That is, the sample will undoubtedly be a variant. This will cause some good contigs to be reported as erroneous, so the results might be slightly better than is immediately apparent from the statistics. The percent aligned metric aligned may mitigate this, but the effect appears small.

4.3 Efficiency

While it is not the intent of this thesis to investigate algorithm efficiency, presentation of the results would be somewhat deficient without some mention. It is not the purpose of this section to conduct a detailed analysis.

With the ideal data in the last chapter, all algorithms completed in, at worst, about five seconds. However, when we start to introduce errors, the situation becomes much worse. For the data represented in Table 4.2, run times and memory usage can be seen in Table 4.3. The time shown is wall clock, as opposed to CPU time. Memory is the maximum allocation during a run (in megabytes). Investigation into efficiency is warranted.

Chapter 5
Conclusion and Future Work

For the case of error free data with good coverage, all algorithms perform well. Y-to-V, omnitig, and modified all provide significantly longer contigs than unitig for error free data. When coverage drops, we see a significant drop off in the performance of Y-to-V and omnitig, where both contig length and coverage with respect to the reference suffer. Unitig and the modified algorithm both also see a reduction in average contig length, but are much more robust to poor coverage. For data containing errors, all algorithms suffer enormously.

The modified algorithm performs well with regard to length of contigs, e-size, and coverage of the genome. In most cases, the modified algorithm matched or beat the performance of the other algorithms. However, algorithm run time can be extreme.

For genome segments that begin with a k-mer that is later repeated in the segment, only unitig will not report an erroneous k-mer. In this case, the contig is in error even though the walk in the graph is safe. For good coverage, the probability of this problem is small. However, the more disjoint segments that we have, the higher the chance of the problematic configuration.

The efficiency of the modified algorithm in its current state is poor. Future work could examine ways to increase efficiency. There is currently significant time used in coordinating source and sink SCCs, and there are undoubtedly more efficient ways of handling a proliferation of source and sink SCCs.

The results indicate that even what may seem a subtle change in how the graph is circularized may have a noticeable impact on the results. This indicates an opportunity for investigation of the heuristics applied to graph circularization.

Additionally, more investigation is warranted into the case of data containing errors. All algorithms appear very sensitive to errors. The poor state of the graphs indicates that better error correction could lead to better contig creation.

Bibliography

[1] Tomescu, A. I. and P. Medvedev (2016) "Safe and complete contig assembly via omnitigs," in *International Conference on Research in Computational Molecular Biology*, Springer, pp. 152–163.

[2] University of Wisconsin (2017), "DNA Sequencing Facility: Illumina Sequencing," . URL `https://www.biotech.wisc.edu/services/dnaseq/services/Illumina`

[3] de Bruijn, N. G. (1946) "A Combinatorial Problem," in *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, vol. 49, Koninklijke Nederlandse Akademie van Wetenschappen, pp. 758–764.

[4] Acosta, N. O., V. Makinen, and A. I. Tomescu (2018) "A Safe and Complete Algortihm for Metagenomic Assembly," *Algorithms for Molecular Biology*, **unk**, p. unk.

[5] Cairo, M., P. Medvedev, N. Obscura Acosta, R. Rizzi, and A. I. Tomescu (2017) "Optimal Omnitig Listing for Safe and Complete Contig Assembly," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 78, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, pp. 29:1–29:12.

[6] Steven L. Salzberg and, A. M. P., A. Zimin, D. Puiu, T. Magoc, S. Koren, T. J. Treangen, M. C. Schatz, A. L. Delcher, M. Roberts, G. Marcais, M. Pop, and J. A. Yorke (2012) "GAGE: A critical evaluation of genome assemblies and assembly algorithms," *Genome Research*, **22**, pp. 557–567.

[7] Pahadia, M. and P. Medvedev (2017), "ContigValidator," . URL `https://github.com/mayankpahadia1993/ContigValidator`

[8] National Center for Biotechnology Information, U.S. National Library of Medicine (2016), "Escherichia coli str. K-12 substr. MG1655, complete genome," . URL `https://www.ncbi.nlm.nih.gov/nuccore/556503834/`

[9] Lander, E. S. and M. S. Waterman (1988) "Genomic Mapping by Fingerprinting Random Clones: A Mathematical Analysis," *Genomics*, **2**, pp. 231–239.

[10] Li, H. (2011), "Reads simulator," . URL `https://www.ncbi.nlm.nih.gov/nuccore/556503834/`

[11] National Center for Biotechnology Information, U.S. National Library of Medicine (2012), "SRX119991: GSM875732: E. coli TSS 1; Escherichia coli K-12; RNA-Seq," . URL `https://github.com/lh3/wgsim`

[12] ——— (2012), "SRX119992: GSM875733: E. coli TSS 2; Escherichia coli K-12; RNA-Seq," .
URL https://www.ncbi.nlm.nih.gov/sra/SRX119992[accn]

[13] Liu, Y., J. Schroder, and B. Schmidt (2012) "Musket: a multistage k-mer spectrum-based error corrector for Illumina sequence data." *Bioinformatics*.
URL https://academic.oup.com/bioinformatics/article/29/3/308/257257

[14] Li, H. (2013) "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM," *ArXiv e-prints*, 1303.3997.

Appendix
Test Data

1 E-coli Test Data

Coverage	Run	Algorithm	# Good	# Bad	Avg Len	E-size	# SCC	# CC
Perfect		Unitig	1749	0	2645	32830	5	1
		Y-to-V	1007	0	4671	33301	5	1
		Omnitig	985	0	4771	33796	5	1
		Modified	987	0	4817	34910	5	1
		Modified g	987	0	4817	34910	5	1
40	a	Unitig	1740	0	2645	32830	5	1
		Y-to-V	1007	0	4671	33301	5	1
		Omnitig	983	2	4812	34904	5	1
		Modified	987	0	4817	34910	5	1
		Modified g	987	0	4817	34910	5	1
	b	Unitig	1740	0	2645	32830	5	1
		Y-to-V	1007	0	4671	33301	5	1
		Omnitig	983	2	4821	34904	5	1
		Modified	987	0	4817	34910	5	1
		Modified g	987	0	4817	34910	5	1
	c	Unitig	1740	0	2645	32830	5	1
		Y-to-V	1007	0	4671	33301	5	1
		Omnitig	983	2	4821	34904	5	1
		Modified	987	0	4817	34910	5	1
		Modified g	987	0	4817	34910	5	1
20	a	Unitig	1740	0	2465	32830	5	1
		Y-to-V	1007	0	4671	33301	5	1
		Omnitig	983	2	4821	34904	5	1
		Modified	987	0	4817	34910	5	1
		Modified g	987	0	4817	34910	5	1

	b	Unitig	1740	0	2645	32830	5	1
		Y-to-V	1007	0	4671	33301	5	1
		Omnitig	985	0	4771	33796	5	1
		Modified	987	0	4817	34910	5	1
		Modified g	987	0	4817	34910	5	1
	c	Unitig	1740	0	2645	32830	5	1
		Y-to-V	1007	0	4671	33301	5	1
		Omnitig	985	0	4771	33796	5	1
		Modified	987	0	4817	34910	5	1
		Modified g	987	0	4817	34910	5	1
10	a	Unitig	1771	0	2612	30406	113	5
		Y-to-V	1026	0	4534	30507	113	5
		Omnitig	985	0	4332	27717	113	5
		Modified	1008	0	4665	31881	113	5
		Modified g	1025	0	4860	33985	113	5
	b	Unitig	1773	0	2609	30030	121	2
		Y-to-V	1030	0	4554	30413	121	2
		Omnitig	955	32	4339	25608	121	2
		Modified	1011	0	4690	31711	121	2
		Modified g	1025	0	4940	34223	121	2
	c	Unitig	1777	0	2603	28520	121	2
		Y-to-V	1031	0	4494	28150	121	2
		Omnitig	962	18	4319	26052	121	2
		Modified	1012	0	4629	29576	121	2
		Modified g	1031	0	4740	31066	121	2
5	a	Unitig	2783	0	1654	6162	4387	874
		Y-to-V	1208	4	1592	2721	4387	874
		Omnitig	964	3	1242	1673	4387	874
		Modified	1172	4	1589	2633	4387	874
		Modified g	2571	4	1954	6611	4387	874
	b	Unitig	2797	0	1647	6251	4418	869
		Y-to-V	1223	7	1481	2360	4418	869
		Omnitig	963	7	1067	1210	4418	869
		Modified	1183	7	1502	2316	4418	869
		Modified g	2489	7	2090	7183	4418	869
	c	Unitig	2789	0	1651	5887	4353	891
		Y-to-V	1206	5	1540	2274	4353	891
		Omnitig	959	5	1150	1280	4353	891
		Modified	1172	5	1555	2253	4353	891
		Modified g	2469	5	2048	6608	4353	891

Table A1: Contig Statistics from E-coli Genome (All Runs)