BIOINFORMATICS

Vol. 00 no. 00 2005
Pages 1–7

Building Fragment Assembly String Graphs

Gene Myers [a*]

[a]Dept. of Computer Science, University of California, Berkeley, CA, USA

ABSTRACT

We present a concept and formalism, the *string graph*, that represents all that is inferable about a DNA sequence from a collection of shotgun sequencing reads collected from it. We give time and space efficient algorithms for constructing a string graph given the collection of overlaps between the reads and in particular, present a novel linear expected time algorithm for transitive reduction in this context. The result demonstrates that the decomposition of reads into k-mers employed in the de Bruijn graph approach of Pevzner *et al.* is not essential and in fact creates both efficiency problems and unecessary conceptual complexities. The current paper is the first in a series and presents the basic algorithm and preliminary results that demonstrate the efficiency and scalability of the method. The result is a step toward a next-generation whole genome shotgun assembler that will easily scale to mammalian genomes.

Contact: gene@eecs.berkeley.edu

1 INTRODUCTION

Paired-end whole genome shotgun sequencing has become the prevailing paradigm for the first phase of sequencing an organism's genome (WeM97; ACH00; VAM01) and routinely delivers 95-99% of the euchromatic sequence in large scaffolds of ordered and oriented contigs. The experiments required to finish the remaining few percent are an order of magnitude more expensive than the shotgun sequencing. For this reason, all but the most important reference genomes will likely never be finished. Thus, improved algorithms and software for whole genome shotgun sequencing will have a large impact on genomic science. For example, an assembler that takes a 97% reconstruction and improves it to 99% is reducing the amount of unresolved sequence by a factor of three (from 3% to 1%) and typically improving contig sizes by a factor of 10 or more (by resolving 90% of the gaps) according to the Poisson statistical theory of sampling (LaW88).

There are currently a number of assemblers capable of whole-genome assembly that all perform comparably by employing variations on the basic paradigm of first finding and assembling unique stretches of DNA with high reliability (MyS00; ApC02; MuN03; JBG03; HWA03). Recurrent strategies include finding mutually reinforcing read pairs and examining the relationship of a read to all others in order to assess its repetitiveness and to correct errors. The central objective for better assembly is to effectively resolve repetitive sequences.

Consider perfect data and a genome that has several perfectly repetitive elements as shown in Figure 1. Imagine that the genome is a piece of thread and meld or collapse all the like repetitive elements as illustrated in Figure 1. We call the resulting graph a *string graph* and it effectively represents everything that can be inferred from the read data. If one can identify arcs corresponding to unique sequence, then a simple flow analysis reveals how many copies of each repeat are collapsed together and if one replicates each of those edges according to their copy count, then the expanded graph has an Eulerian tour and one of those tours corresponds to the original sequence. We show how to build such a graph in this paper.

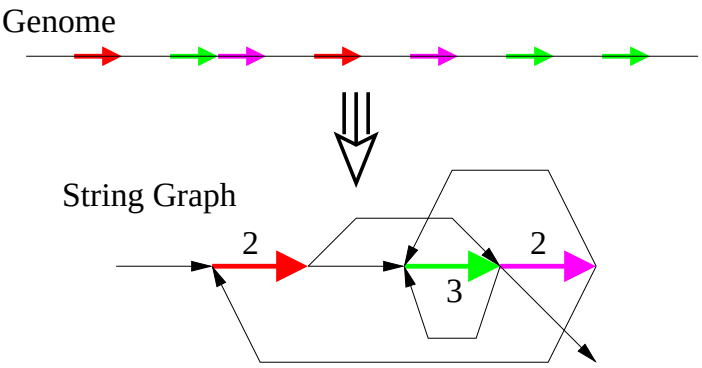

Fig. 1. A genome and its string graph. The colored segments of the same color represent identical repetitive sequences. The numbers in the string graph give the number of copies of each repeat inferable by counting entry and exits into the collapsed segment.

In 1992 this author and Waterman-Idury presented two new ideas for fragment assembly in back to back talks at a DIMACs workshop on bioinformatics that were later published in the same volume of a journal (Mye95; IdW95). Myers' approach was based on finding maximal interval subgraphs in the graph of all read overlaps and was subsequently developed into the "unitig" concept of the Celera assembler. Waterman and Idury presented an approach based on building the de Bruijn graph of all k-mers from the reads and then finding paths in this graph supported by the reads. This approach was subsequently extended by Pevzner and members of his research group, giving rise to the Euler assembler (PTW01).

While the idea of a string graph is explicit in the Euler algorithms, we show in this paper that a string graph directly follows from the unitig algorithm as well. What this amounts to is a demonstration that the idea of k-mers is unnecessary, that one can work directly from the reads, giving rise to a much more space efficient algorithm – one that can scale to a mammalian genome on current hardware. Our approach requires a transitive reduction step and we give a novel linear expected time algorithm for this in our context. We also show how to treat contained reads in an efficient way and how to efficiently solve large parts of the minimum cost network flow problem that is the last step by making some simple observations. This work

*to whom correspondence should be addressed

is a first report on a new line of algorithm development, so we conclude with some preliminary empirical results and a brief discussion of future work.

2 BUILDING A STRING GRAPH

Consider a genome sequence S of length G and a shotgun data set of n reads, $f_1, f_2, \ldots f_n$ randomly sampled from either the forward or reverse strand of the genome. Let $f.len$ be the length of read f and let $f[1]f[2]f[3]\ldots f[f.len]$ be its DNA sequence over the alphabet $\Sigma = \{A, C, G, T\}$. The average over-sampling of the genome is $c = N/G$ where $N = \Sigma_f f.len$ is the total amount of sequence in the data set. We assume that the reads have been preprocessed so that each is truly a sample of the genome (i.e. contains no vector sequence or other contaminant) and that the mean error rate of the read's sequence is less than ϵ (e.g. 2.5%) under an exponentially distributed arrival rate model.

The first step in constructing our string graph is to compute the set of all 2ϵ overlaps of length τ or more between the reads. For a given ϵ, τ should be chosen so that the probability of a 2ϵ match between two random strings of length τ is exceedingly low, for example, when ϵ = 2.5% we choose $\tau = 50$. This overlap computation is the most time consuming step and amounts to a large sequence comparison between the concatenation of all the reads against itself. Numerous heuristic and filtration algorithms have been developed that offer good performance – roughly $O(N^2/M)$ expected time where M is the available memory of the machine. In particular, we use a recently introduced filter based on q-grams (JSM05) so that all desired overlaps are guaranteed to be found. On the order of $O(cN)$ overlaps generally result comprising both true overlap relationships and those induced by repetitive sequences in the genome.

For an overlap o between reads f and g the matching substrings are specified by giving the two intervals, $[o.f.beg, o.f.end]$ and $[o.g.beg, o.g.end]$, delimiting them. We index the positions between characters starting at 0 so that $f[a, b] = f[a + 1]f[a + 2]\ldots f[b]$. Moreover, if $a > b$ then $f[a, b] = comp(f[b, a])$ where $comp(f)$ is the Watson-Crick complement of f. An interval endpoint is termed *extreme* if it is either 0 or the length of the relevant read. Observe that every overlap has at least two extreme endpoints, and that $|o.f.end - o.f.beg| \approx |o.g.end - o.g.beg|$. An overlap is a *containment* if both ends of a read are extreme and the read in question is said to be *contained*. Otherwise an overlap is *proper* and it follows that $o.f.beg$ ($o.f.end$) or $o.g.beg$ ($o.g.end$) is extreme but not both.

Given the set of all overlaps as specified above we can now give a preliminary construction of a string graph. The goal is a string-labeled graph in which the original genome sequence corresponds to some tour of the graph and where the graph has as few extraneous edges and alternate tours as possible. For example, a graph consisting of a single vertex and four self-loops with each DNA letter label is always a string graph of a genome, but not a particularly informative one.

The basic observation is that every read must be spelled in the graph and the concatenation of every overlapping pair of reads must be spelled in the graph. It follows immediately that every contained read can be removed from the problem because there will be a tour spelling the sequence of the reads containing it. Typically 40% of the reads in an assembly data set are contained, and so 64% of the overlaps involve a contained read. Removing these reads and

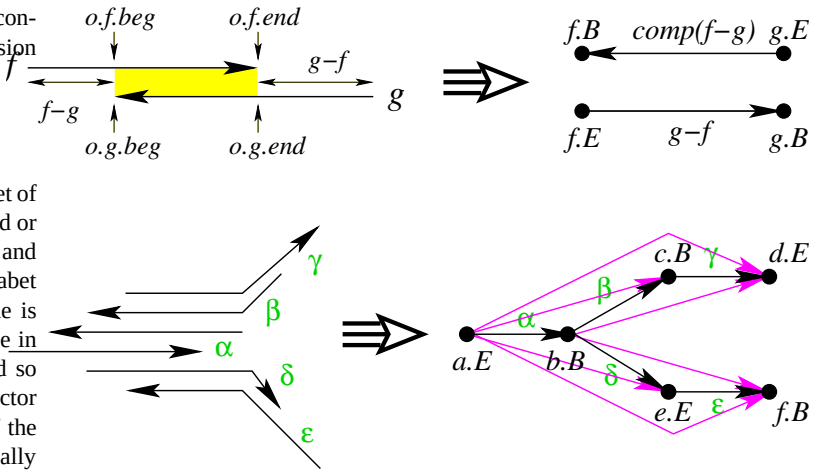

Fig. 2. String Graph Construction. At upper left is an overlap with its defining intervals and overhanging strings annotated. At the upper right are the two edges that should be placed in the string graph for the given overlap. At the lower left are some reads that overlap and then branch in two directions. At the lower right is the resulting portion of the string graph for the overhangs on the right side of the arrangement of reads. The edges that are transitively reduced are colored magenta.

their overlaps gives a significant practical reduction in the size and memory requirements for the problem.

Hence forward consider the set of non-contained reads and their overlaps. For each such read f there will be two vertices, $f.B$ and $f.E$, one for each end of the read, in the string graph. Figure 2 illustrates the construction of edges and their labels. The intuition is that one adds a directed edge labeled with the non-matched or overhanging sequence at each end of the proper overlap between two reads. More formally assume without loss of generality that in the encoding of overlap o, $o.f.beg < o.f.end$. Then exactly the following two edges are added for each overlap:

if $o.f.beg > 0$ *then*
  if $o.g.beg < o.g.end$ *then*
    Add edge from $g.B$ to $f.B$ labeled $f[o.f.beg, 0]$
    Add edge from $f.E$ to $g.E$ labeled $g[o.g.end, g.len]$
  else
    Add edge from $g.E$ to $f.B$ labeled $f[o.f.beg, 0]$
    Add edge from $f.E$ to $g.B$ labeled $g[o.g.end, 0]$
else
  if $o.g.beg < o.g.end$ *then*
    Add edge from $f.B$ to $g.B$ labeled $g[o.g.beg, 0]$
    Add edge from $g.E$ to $f.E$ labeled $f[o.f.end, f.len]$
  else
    Add edge from $f.B$ to $g.E$ labeled $g[g.len, o.g.beg]$
    Add edge from $g.B$ to $f.E$ labeled $f[o.f.end, f.len]$

For a vertex v in the string graph, let $v.read$ be the read corresponding to the vertex and let $v.type$ be B or E depending on which end of the read corresponds to the vertex. It follows by a simple induction that for a path $p = v_1 \rightarrow v_2 \rightarrow v_3 \ldots v_n$ every read $v_i.read$, iff $v_i.type = E$, or its complement, iff $v_i.type = B$ is a prefix or substring of the string spelled along the edges of the path

p. That is, any path in the graph represents a consistent assembly of the reads, or their reverse complement, whose ends occur along it. Therefore the original source sequence and its complementary strand can be spelled in the graph just constructed. Moreover, every sequence of two edges $v_1 \rightarrow v_2 \rightarrow v_3$ is a corroborated by a read, namely $v_3.read$. We call this property *read coherence*.

The graph still has more edges than necessary, in particular, if f overlaps g overlaps h in such a way that f overlaps h as well, then the string graph edge $f \rightarrow h$ is unnecessary as one can use the edges $f \rightarrow g \rightarrow h$ to spell the same sequence. That is one may remove all transitive edges from the graph above without impacting what can be spelled. Moreover this reduction typically decreases the number of edges in the graph by a factor of c.

Transitive reduction also leaves many vertices with in- and out-degree exactly one. That is, there are many chains with no branching possibilities. We call a vertex a *junction* vertex if it has in- or out-degree not equal to 1, and an *internal* vertex otherwise. Collapse all paths whose interior vertices are internal and that begin and end with a junction vertex, replacing them with a single *composite edge* whose label is the concatenation of the labels along the path. The resulting graph of junction vertices and composite edges is obviously still a read coherent string graph.

At this juncture observe that because no read contains the other, every composite edge $e = v_1 \rightarrow v_2 \rightarrow v_3 \cdots \rightarrow v_n$ has a complementary edge $comp(e) = comp(v_n) \rightarrow \ldots comp(v_3) \rightarrow comp(v_2) \rightarrow comp(v_1)$ where $comp(v)$ is the other end of the read for f. That is, if $v = f.B$ then $comp(v) = f.E$ and if $v = f.E$ then $comp(v) = f.B$. This property implies that we may instead think of the endpoint pairs as a single vertex with bidirected edges corresponding to an edge and its complement, where an arrowhead is directed into a vertex if the edge to its $.E$ vertex is the head of the relevant one of the two complementary edges, and directed our of the vertex otherwise. This gives us a framework identical to the one introduced by this author and Kececioglu (KeJ95) where a tour through a vertex must involve one inward arrowhead and one outward arrowhead. Figure 4 gives an example of our construction for the genome *C. jejuni*.

As we will see in the preliminary results section, the string graph that results from the above algorithm has between two to three orders of magnitude fewer vertices and edges then the number of reads and overlaps and thus represents a significant reduction in the complexity of the problem. We also point out that the transitive-reduction/chain-collapsing steps are in essence a recapitulation of the maximal interval subgraph algorithm introduced by this author in 1992. Previously the string graph was implicit, with edges being modeled by vertices (unitigs) and vertices by edges (overlaps). Now it is explicit and the connection to the de Bruijn approach of Euler should be apparent.

3 A LINEAR TRANSITIVE REDUCTION ALGORITHM

General transitive reduction of a graph takes $O(\Sigma_{v \rightarrow w \in E} deg(w)) = O(ED)$ time where $deg(w)$ is the out degree of w, E is the number of edges, and D is the maximum out degree of a vertex. But in our context the graph models overlaps in which the length of the interval represented by each read is known as is the amount of overlap between two such intervals. We will leverage this to give an algorithm that takes $O(\Sigma_v\ tr.deg(v) deg(v))$ worst case time where $tr.deg(v)$ is the out degree of v *in the transitively reduced graph*. Assuming all input sequences are equally likely, $tr.deg(v) = O(1)$ on average and the algorithm thus takes $O(E)$ expected time. Of course, real genomes have non-random repetitive structures, but even in these cases the preponderance of the genome is unique sequence so that in practice we see very rapid, near linear behavior.

Consider the edges out of a vertex v: $v \rightarrow w_1$, $v \rightarrow w_2$, ..., $v \rightarrow w_n$. Let $len(v \rightarrow w)$ be the length of the string labeling the edge, which we also consider to be the length of the edge. In a preprocessing sorting step we order the adjacency lists of all vertices so that the edges out of each are in increasing order of their length. Suppose that $tr.deg(v)$ is one, i.e. that there is only one non-transitive or *irreducible* edge out of v. Then it must be the shortest edge $v \rightarrow w_1$ and every edge $w_1 \rightarrow w_2$, ..., $w_1 \rightarrow w_n$ must be in the graph. In general, suppose $tr.deg(v) = k$. Then there is at least one edge from one of the w at the head of one of the k irreducible edges out of v to each w that is not at the head of a irreducible edge. Therefore the following simple marking strategy, the pseudo-code for which is in Figure 3, correctly identifies the heads of the irreducible edges.

```
      constant FUZZ ← 10

1.    for v ∈ V do
2.     { mark[v] ← vacant
3.       for v → w ∈ E do
4.           reduce[v → w] ← false
       }

5.    for v ∈ V do
6.     { for v → w ∈ E do
7.           mark[v] ← inplay

8.       longest ← max_w len(v → w) + FUZZ

9.       for v → w ∈ E in order of length do
10.          if mark[w] = inplay then
11.              for w → x ∈ E in order of length and
12.                    len(w → x) + len(v → w) ≤ longest do
13.                 if mark[x] = inplay then
14.                     mark[x] ← eliminated

15.      for v → w ∈ E in order of length do
16.          for w → x ∈ E in order of length and
17.               (len(w → x) < FUZZ or
                    w → x is the smallest edge out of w) do
18.              if mark[x] = inplay then
19.                  mark[x] ← eliminated

20.      for v → w ∈ E do
21.       { if mark[w] = eliminated then
22.             reduce[v → w] ← true
23.          mark[w] ← vacant
         }
       }
```

Fig. 3. Transitive Reduction Algorithm.

Initially mark every vertex in the graph as *vacant* and record that every edge need not be reduced (lines 1-4). Then for each vertex apply the following marking strategy (line 5). First, mark every vertex reachable from v as *inplay* (lines 6-7). Then for each vertex w_i on v's adjacency list in order of edge length do the following (line 9). If w_i has been marked *eliminated* during the processing of an earlier w_j then nothing need be done (line 10). Otherwise, $v \rightarrow w_i$ is a irreducible edge and we traverse edges out of w_i marking as *eliminated* any vertex we encounter at the head of such an edge that is marked *inplay*, indicating that it is adjacent to v (lines 11, 13-14). We further take advantage of the fact that w_i's edges are ordered to stop processing edges once an edge is too long to eliminate edges out of v (lines 8, line 12). One concludes the processing of v by examining every vertex on its adjacency list, marking as needing reduction any edge whose head has been marked *eliminated* and then restoring the vertex marks to *vacant* (lines 20-23). Confirming the time complexity of the algorithm stated above is left as an exercise.

Thus far we have been implicitly assuming that read overlap relationships are completely consistent with each other as one might see if the data were perfect. But read overlaps are approximate matches which can have two consequences. First endpoint positions can shift a bit and one needs to allow for this in any logic that uses distances, i.e. the use of $FUZZ$ in line 8. Secondly and more importantly, approximate equality is not an equivalence relation because transitivity can fail to hold depending on the distribution of errors in the reads. For example, it is not infrequent that $w_1 \rightarrow w_n$ is not found because even though w_n has a larger overlap with w_1 than with v, both overlaps are thin and coincidentally w_1 has just a couple more errors in the relevant interval than v does and so is pushed above the ϵ error rate for overlaps. Notice in this case that the read most likely to be found overlapping with w_n is w_{n-1}, its nearest predecessor. So we make the algorithm very stable with respect to approximate matching by adding lines 15-19, that for each w_i checks if its immediate successor and additional successors within $FUZZ$ (10) base pairs of it eliminate vertices on v's adjacency list. In expectation, the neighborhood is $O(1)$ in size so the addition adds only a total of $O(E)$ expected time to the algorithm.

4 ESTIMATING GENOME SIZE AND IDENTIFYING UNIQUE SEGMENTS

Given a read coherent string graph we now wish to label every edge with an integer specifying the number of times one should traverse the edge in reconstructing the underlying genome. Our first step towards this end is to determine with very high probability those edges that should be traversed exactly once, i.e., that correspond to unique stretches of DNA. Consider a composite edge between two junction vertices v and w, and suppose it is of length Δ and there are k internal vertices that comprise the chain of the composite edge. This path models an overlapping sequence of $k+2$ reads that assemble together consistently. Suppose for the moment we know the size G of the genome so that we know the average spacing G/n expected between reads. As we did for the Celera assembler we can then determine the log-odds ratio, or A-statistic, of the path representing a unique sequence versus an overcollapsed repeat.

A quick derivation of the A-statistic is as follows. The probability that the path is single copy is $\binom{n}{k} (\frac{\Delta}{G})^k (\frac{G-\Delta}{G})^{n-k}$ which is approximately $((\frac{\Delta n}{G})^k/k!)\, e^{-\Delta n/G}$ in the limit as $G \rightarrow \infty$. By the same approximation the probability that the path should be traversed twice is $((\frac{2\Delta n}{G})^k/k!)\, e^{-2\Delta n/G}$. The natural log of the ratio of these two probabilities, or A-statistic is $A(\Delta, k) = \Delta(\frac{n}{G}) - k \ln 2$.

Unfortunately, we do not know the size of the genome G. An inspection of the typical string graph reveals that most of the total length of all the edges is concentrated in a few rather large ones which are almost all likely to be single copy. So an effective bootstrap procedure is to compute the average arrival rate over all edges over a certain length, say 10Kbp, and then compute the A-statistic for every edge using this estimate. One then considers every edge with an A-statistic over a threshold, say 17 (1-in-24million chance of being wrong), to be single copy. One can then re-estimate the average arrival rate over this subset of edges and then reapply the statistic until convergence or a fixed number of times. We find one turn suffices to yield an estimate of G that is typically accurate to within 1% of the sampled genome's true size.

Using the A-statistic, we have, with very high probability, labeled every single copy edge of the string graph. Further observe that every composite edge that has an interior vertex must be traversed at least once if the read(s) corresponding to the interior vertex(ices) are to take part in the reconstruction of the genome. Since every read was presumably sampled from the genome, we know that every such edge must be used once. All other edges, those that do not have interior vertices, may be ignored if desired.

In summary, we have estimated the size of the genome and now have a string graph graph in which every edge has a lower and possibly upper bound on the number of times it must be traversed in a reconstruction. Specifically there are three cases: (a) the edge must be traversed $= 1$ times, (b) the edge must be traversed ≥ 1 times, or (c) the edge must be traversed ≥ 0 times.

5 MAPPING CONTAINED READS

There is actually a rather serious flaw in the procedure of the previous section. Recall that contained reads were removed from the problem, and the graph only built from the remaining reads, all of which properly overlap. This means that the density of read start points is underestimated as typically 40% of the reads are contained. In and of itself this would not be a problem save that the probability of a read being contained by reads from a unique segment is significantly less than the probability of a read being contained by reads from a repeat segment, the probability increasing the higher the copy number. This has the effect of making repeat segments look less repetitive than they are with respect to an A-statistic computed over just the non-contained fragments.

To rectify this bias and also get a better true estimate of the genome size, we map every contained read endpoint to the composite edge or edges in which it would lie if it had been part of the original graph. Note carefully that all we need to do is accumulate the count with each edge in order to compute a more accurate A-statistic, the location of the end point in the composite edge's chain is irrelevant. Also note that we state that a contained read endpoint can map to several edges. The reason for this becomes apparent as we sketch the mapping procedure below.

Consider the end of a contained read f. The treatment for the position of the start of a read is symmetric and will not be given for brevity. We first find the containing overlap $o = O_E(f)$ for which the length of the overhang, $D_E(f) = H_E(f, o)$, off that end of f

to the relevant end of the containing read, $V_E(f)$ is the smallest. Formally,

$$\begin{aligned}
H_E(f,o) &= \begin{cases} o.g.len - o.g.end & \text{if } o.g.beg < o.g.end \\ o.g.end & \text{otherwise} \end{cases} \\
O_E(f) &= o \text{ s.t. } H_E(f,o) \text{ is smallest over all } o \text{ for which } o.g \\
&\qquad \text{contains } f \text{ and } o.g \text{ is not itself contained} \\
V_E(f) &= \begin{cases} O_E(f).g.E & \text{if } O_E(f).g.beg < O_E(f).g.end \\ O_E(f).g.B & \text{otherwise} \end{cases} \\
D_E(f) &= H_E(f, O_E(f))
\end{aligned}$$

The computation of O, V, D_E for each contained read can clearly be accomplished in time linear in the number of overlaps. The endpoint of contained read f belongs $D_E(f)$ base pairs upstream of the vertex $V_E(f)$ in the string graph. By upstream we mean in the direction opposite to those for which the edges through $V_E(f)$ are directed. Algorithmically, we engage in a reaching computation that moves $D_E(f)$ base pairs along the reverse edges of the graph from $V_E(f)$. While in most cases the graph does not branch during the search in some cases it may in which case we find all edges at which the endpoint of f could lie. Formally, we compute $Map(V_E(f), D_E(f))$ as follows:

$$Map(v,d) = \bigcup_{w \to v} \begin{array}{l} \text{if } len(w \to v) < d \text{ then} \\ \quad Map(w, d - len(w \to v)) \\ \text{else} \\ \quad \{w \to v\} \end{array}$$

If a contained read endpoint maps to a unique edge then the endpoint is counted toward that edge's A-statistic. When an endpoint maps to multiple locations we give each location a fractional count of $1/|Map(V_E(f), D_E(f))|$. As an estimator of the mean population this is sensible as the reads in such an ambiguous situation are equally likely to be in one of the possible locations. Note that the A-statistic as formulated above easily accommodates fractional edge counts. Finally, if the number of possible locations exceeds some threshold, say 100, the reaching computation is terminated and the endpoint, which would contribute less than 1/100th of a count to any edge, is ignored. This guarantees that the mapping phase takes a maximum of $O(n)$ time.

In summary, containment endpoints are first mapped as above and then with these revised edge endpoint counts, the estimation of genome size and edge traversal bounds described in the prior section takes place.

6 MINIMUM COST NETWORK FLOW AND SIMPLIFICATIONS

The last task is to decide on the traversal count, $t(e)$ for each edge in the string graph, given upper and lower bounds $[l(e), u(e)]$ on the edges from the previous phases of the construction. We begin by formulating the problem as a minimum cost network flow problem where in we find the minimum traversal counts that satisfy the edge bounds and preserve equality of in- and out-counts (flows). That is, if we think of the traversal counts as integral flows, then if net inflow to a node equals net outflow, there is a generalized Eulerian tour that traverses each edge $t(e)$ times. By minimizing the flow subject to the bound constraints we are appealing to parsimony.

A reconstruction takes the form of a number of contiguous sequences, the breaks between sequences being due to a failure to sample some regions by chance. Each of these is a distinct tour and typically these begin at a vertex with zero in degree and end at a vertex with zero out degree. However, a contig could begin or end at a vertex with non-zero in/out-degree. So to correctly model the flow problem, we must add ϵ-labeled meta-edges $s \to v$ and $v \to s$ into and out of every junction vertex from a special source vertex s. There are no bounds on these edges and we now seek a cyclic tour where in we understand there to be a contig break whenever s is traversed. We appeal to parsimony and seek a minimum integral flow satisfying the edge bounds. Formally, a minimum cost network flow problem (AMO93) is usually formulated as follows

> *Input:* For each edge e, an upper bound $c(e) \geq 0$ on flow, and a cost per unit flow $v(e)$. For each vertex v, a supply (positive) or demand (demand) $b(v)$ for flow.
>
> *Output:* A flow $x(e)$ on each edge such that $\Sigma_e v(e)x(e)$ is minimal subject to $x(e) \in [0, c(e)]$ and $\Sigma_{u \to v} x(u \to v) + b(v) = \Sigma_{v \to w} x(v \to w)$.

In these terms our problem is as follows where we have used a known transformation to convert the lower bounds into 0's by capturing them in the supply/demand values:

$$\begin{aligned}
c(e) &= u(e) - l(e) \\
b(v) &= \Sigma_{u \to v} l(u \to v) - \Sigma_{v \to w} l(v \to u) \\
v(e) &= 1 \\
t(e) &= l(e) + x(e)
\end{aligned}$$

The particularly simple structure of our edge bounds – $= 1, \geq 1$, or ≥ 0 – leads to a particularly simple flow problem. Specifically, all $(= 1)$-edges have $c(e) = 0$ and all other edges have $c(e) = \infty$. In compensation the supply/demand values $b(v)$ take on some integral value.

Before invoking an existing algorithm (AMO93) for the flow problem, whose worst case complexity is $O(EV)$, there are a number of simplifications that can take place. (1) For any $(= 1)$-edge, implying $c(e) = 0$, set $x(e) = 0$ and remove the edge. (2) If a vertex has $b(v) = 0$ and either no in-edges or no-out edges then set $x(e) = 0$ for any edge e adjacent to the vertex and then remove the vertex and its adjacent edges. (3) If a vertex v has a single out-edge $v \to w$, then set $x(v \to w)$ to $b(v)$, reset $b(w)$ to $b(w) + b(v)$, and merge v and w removing the edge. (4) If a vertex v has a single in-edge $u \to v$, then set $x(u \to v)$ to $-b(v)$ and reset $b(u)$ to $b(u) - b(v)$ and merge v and u. While these transformations are all quite simple, it turns out that on practical problems they are very effective in reducing the size of the problem, in particular, the edge removals and fusions turn the graph into a large collection of small connected components each of which is more efficiently solvable with the full, standard algorithms. For an example, see Figure 5 We call the edges that remain after the simplifications above *nontrivial edges* and the vertices that have non-zero supply demand *unsatisfied vertices*. Basically, one needs to push flow between unsatisfied vertices in a minimal way. Flow pushing algorithms generally perform much better than their worst case complexity in such scenarios. For an example of the final string graph see Figure 6.

7 PRELIMINARY RESULTS

This paper is a preliminary algorithms piece. We are still in the process of producing a total solution that takes into account all the subtleties of real data, which does not satisfy key assumptions made at the outset of the paper. Specifically, vector sequence generally contaminates some percentage of the reads, the sample genome DNA is not completely isogenic, and the error rate across a purported high-quality interval determined using Phred scores is not always particularly accurate. We are working on a variety of levels including preprocessing methods and extensions of the basic approach presented here to address these realities. Our more modest goal here is to show that this approach is *very* time and memory efficient and under the stated assumptions produces the desired string graph. The method will scale on current architectures to problems of the scale of the human genome, something not possible with the de Bruijn graph approach.

We consider simulated shotgun data sets of three target genomes: a 500Kbp synthetic genome with ten copies of a 300bp repeat at 2% variation ("Synthetic Alus"), the 1.64Mbp sequence of the bacteria *C. jejuni*, and the first 13.9Mbp of the euchromatic sequence of arm 3R of *D. melanogaster*. For each genome we synthetically sampled a 10X data set of reads of length chosen uniformly between 550bp and 850bp. Each read has errors introduced with probably .008 at its 5' end linearly ramping to .025 at its 3' end. We used the *celsim* simulator (Mye99).

In Table 1 we present a number of empirically measured parameters for these three genomes of increasing size. The first row gives the genome size. The next grouping gives the number of reads in the input data set and the number of overlaps computed between those reads. The third grouping gives the number of reads that are contained by at least one other read and the number of (relevant) overlaps that are between non-contained reads. We note that when read lengths are normally distributed, as opposed to the uniform distribution of our simulation, the percentage of contained reads is even higher. That is, the savings from eliminating contained reads at this point is conservative compared to what we observe for real data. The fourth grouping gives the number of irreducible edges not removed from the initial string graph, the number of junction nodes, and the number of composite edges that result when chains are collapsed. Note that the graph is generally small compared to the number of reads and overlaps input. For example, for 3R we go from 202,000 reads to 756 junction vertices, and from 4 million overlaps to 1,200 composite edges. The fifth grouping gives the genome size estimated after inserting contained read endpoints and the number of edges that are with very high confidence deemed to be single copy DNA, i.e. $(= 1)$-edges. The sixth grouping characterizes the results of the simplifications we apply before invoking general min-cost network flow algorithms. By non-trivial edges we mean those that do not get eliminated by the simplifications, and we give the number of connected components containing those edges. For example, in the case of 3R, the min-cost network flow algorithms are applied to 33 components containing a total of 538 non-trivial edges, for an average of 16-17 edges per component. Also note that the number of unsatisfied vertices for which $b(v) \neq 0$ is small. Finally, we report the total computation time and space used. One sees a clearly linear increase in resources and very efficient times. In particular the amount of memory is slightly more than the size of the target genome in Mb.

Table 1. Computational Results on Three Simulated Shotgun Datasets.

	Synthetic Alus	*C. jejuni*	*D. mel.* 3R
Genome Size(Mbp)	.500	1.641	13.919
Reads	7000	23,900	202,200
Overlaps	127K	462K	3,997K
Contained Reads	41.4%	42.6%	43.1%
Relevant Overlaps	44K (41%)	150K (43%)	1,268K (43%)
Irreducible Edges	8,310	27,500	231,096
Junction Vertices	33	75	756
Composite Edges	89	113	1294
Size Estimate (Mbp)	.499	1.626	13.765
$(= 1)$-edges	11	20	179
Nontrivial Edges	2	51	538
No. of Components	1	4	33
Unsatisfied Vertices	0	6	109
Time (sec)	3.7	13.1	113.6
Space (Mb)	.53	1.81	15.28

Table 2. Containment Mapping for *D. mel.* 3R Dataset.

$\|Map\|$	Containment Endpoints
1	173,399
2	642
3	173
4	70
5	18
6	6
7	5
8	22
9	3

In Table 2 we illustrate the amount of ambiguity that occurs in mapping containment endpoints by giving a histogram of the size of $|Map|$ for the contained reads in the *D. mel.* 3R dataset. The main thing to observe is that most endpoints map to a unique location with an exponentially vanishing but somewhat irregular tail of multiple location endpoints. In effect the mapping is linear in expected time and very rapid.

We illustrate the shape and size of the string graph at various steps in the process with the *C. jejuni* data set as the automated graph drawing program produces rather unaesthetic results for the larger 3R dataset. In Figure 4 we show the graph after transitive reduction and chain collapsing. In Figure 5 we show the 4 subproblems that must be solved with min-cost network flow, and in Figure 6 we show the final string graph. Note carefully that it is exactly what we desired to compute at the outset. The red edges are those that are to be traversed multiple times. There are 72 possible tours of the final string graph. Seven PCR reactions would resolve the true tour, or in a project with paired end reads, the correct tour would readily be apparent.

REFERENCES

[WeM97]J.L. Weber and E.W. Myers. Human whole-genome shotgun sequencing. *Genome Res.* 7, 4 (1997), 401-409.

[ACH00]M.D. Adams, S.E. Celniker, R.A. Holt RA, *et al.* The genome sequence of Drosophila melanogaster. *Science* 24, 287 (2000), 2185-2195.

[VAM01]J.C. Venter, M.D. Adams, E.W. Myers, *et al.* The sequence of the human genome. *Science* 16, 291 (2001), 1304-51.

[LaW88]E.S. Lander and M.S. Waterman. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics* 2 (1988), 231-239.

[MyS00]E.W. Myers, G.G. Sutton, *et al.* A whole-genome assembly of Drosophila. *Science* 24, 287 (2000), 2196-204.

[ApC02]S. Aparicio, J. Chapman J, *et al.* Whole-genome shotgun assembly and analysis of the genome of Fugu rubripes. *Science* 23, 297 (2002), 1301-1310.

[MuN03]J.C. Mullikin and Z. Ning. The phusion assembler. *Genome Res.* 13 (2003), 81-90.

[JBG03]Jaffe DB, Butler J, Gnerre S, Mauceli E, Lindblad-Toh K, Mesirov JP, Zody MC, Lander ES. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Res.* 13, 1 (2003), 91-96.

[HWA03]X. Huang, J. Wang, S. Aluru, S.P. Yang, and L. Hillier. PCAP: A Whole-Genome Assembly Program. *Genome Res.* 13 (2003), 2164-2170.

[Mye95]E.W. Myers. Toward simplifying and accurately formulating fragment assembly. *J. Comput. Biol.* 2, 2 (1995), 275-290.

[IdW95]R.M. Idury, W.S. Waterman. A new algorithm for DNA sequence assembly. *J. Comput. Biol.* 2, 2 (1995), 291-306.

[PTW01]P.A. Pevzner, H. Tang, and M.S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA* 14, 98 (2001), 9748-9753.

[JSM05]K. Rasmussen, J. Stoye, and E.W. Myers. Efficient q-Gram Filters for Finding All e-Matches Over a Given Length. *9th Conf. on Computational Molecular Biology* (Boston, MA, 2005), to appear.

[KeJ95]J. Kececioglu and E. Myers. Combinatorial Algorithms for DNA Sequence Assembly *Algorithmica* 13, 1-2 (1995), 7-51.

[AMO93]R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliff, N.J. (1993).

[Mye99]E.W. Myers. A Dataset Generator for Whole Genome Shotgun Sequencing. *Conf. on Intelligent Systems for Molecular Biology* (Heidelberg, Germany, 1999), 202-210.

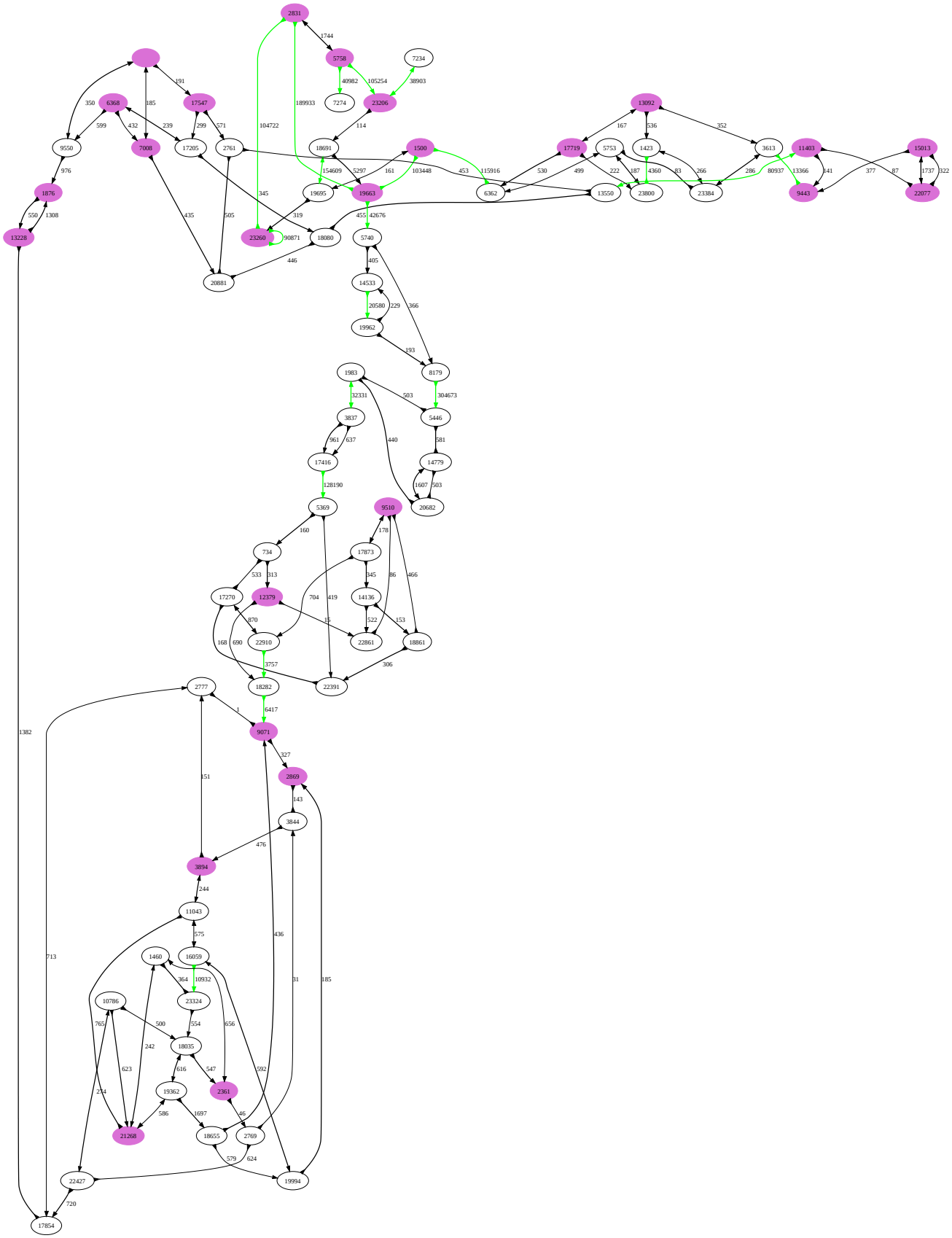

Fig. 4. *C. jejuni* String Graph Prior To Traversal Analysis.

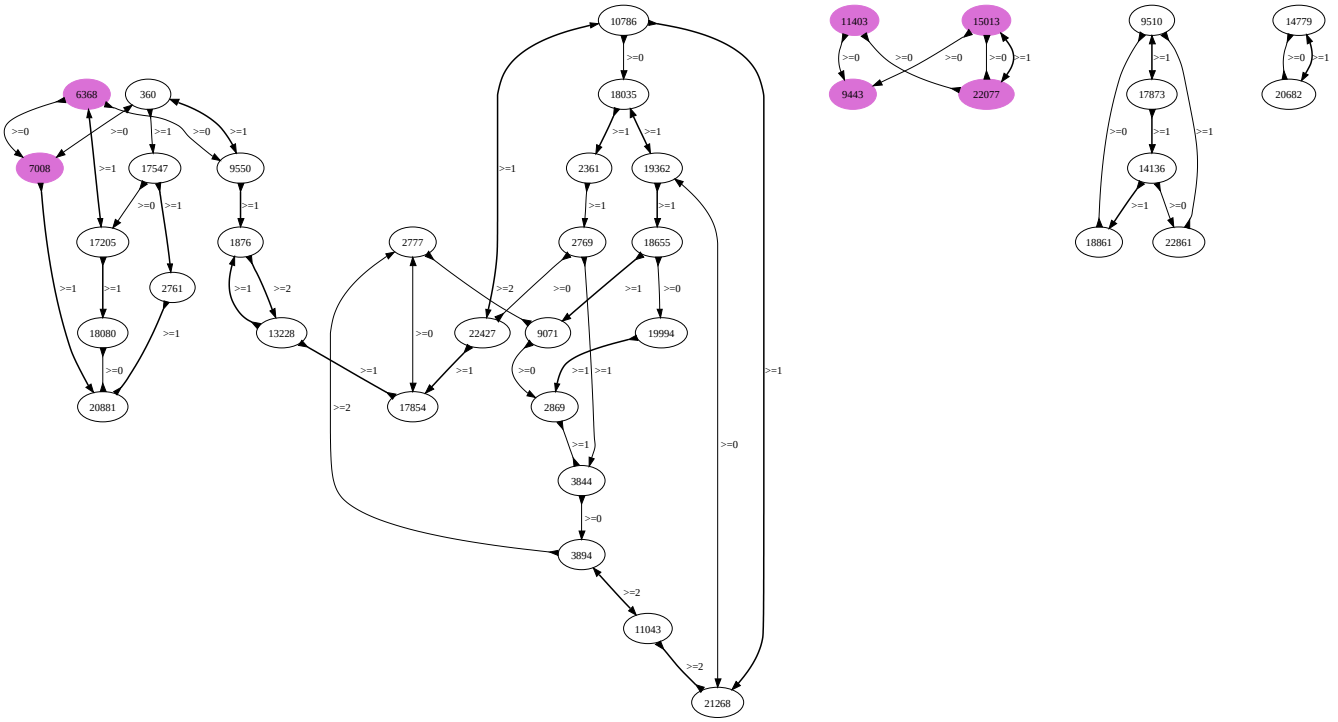

Fig. 5. *C. Jejuni* String Graph Components Requiring Min Cost Network Flow for *C. jejuni*. We show the subgraph of non-trivial edges with unsatisfied vertices drawn as solid purple vertices.

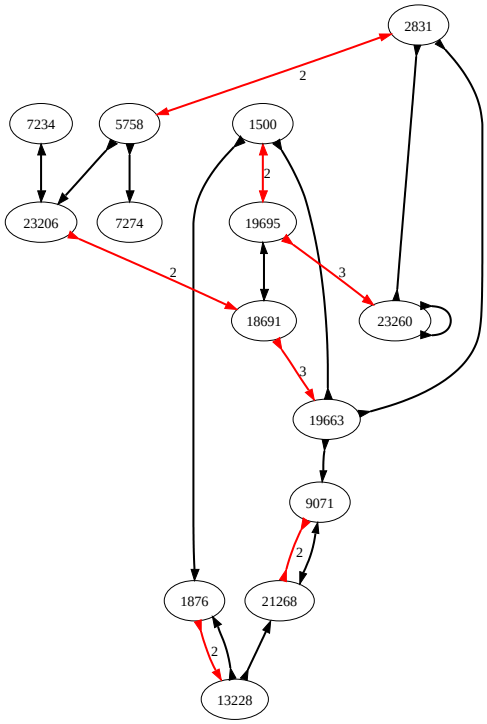

Fig. 6. The Final *C. jejuni* String Graph with Traversal Counts. The red edges are those that need to be traversed more than once.