BIOINFORMATICS

Vol. 21 Suppl. 2 2005, pages ii79–ii85
doi:10.1093/bioinformatics/bti1114

Genes and Genomes

The fragment assembly string graph

Eugene W. Myers
Department of Computer Science, University of California, Berkeley, CA, USA

ABSTRACT
We present a concept and formalism, the string graph, which 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 described earlier is not essential, and exposes its close connection to the unitig approach we developed at Celera. This paper is a preliminary piece giving the basic algorithm and results that demonstrate the efficiency and scalability of the method. These ideas are being used to build a next-generation whole genome assembler called BOA (Berkeley Open 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 (Weber and Myers, 1997; Adams *et al*., 2000; Venter *et al*., 2001) 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, only the most important reference genomes will likely ever to be finished. Thus, improved algorithms and software for whole genome shotgun sequencing can 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 (Lander and Waterman, 1988).

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 (Myers *et al*., 2000; Aparicio *et al*., 2002; Mullikin and Ning, 2003; Jaffe *et al*., 2003; Huang *et al*., 2003). 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 perfect repeats whose lengths are longer than any read 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

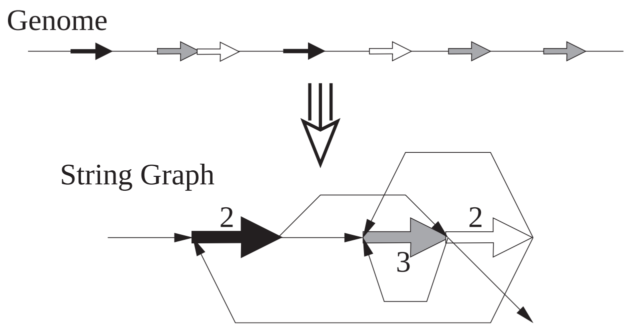

Fig. 1. A genome and its string graph. The thick arrows of the same shade represent identical repetitive sequences. The numbers in the string graph give the number of copies of each repeat inferable by counting entry and exits into the collapsed segment.

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.

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 which were later published in the same volume of a journal (Myers, 1995; Idury and Waterman, 1995). 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 *et al*. (2001) of his research group giving rise to the Euler assembler.

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, obviating the need for the complex read-based path splitting of Euler and 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 rigorously formulated, minimum cost network flow problem that is our last step with a series of simplifying reductions. Finally, the problem of orientation, i.e. that reads can be from either the forward or reverse strand, is addressed and accounted for in our string graph formulation. This work 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 $\Sigma = \{\mathrm{A, C, G, T}\}$. 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 dataset. 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 a string graph is to compute 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, e.g. when $\epsilon = 2.5\%$ we choose $\tau = 50$. The 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 (Rasmussen *et al.*, 2005) so that all desired overlaps are guaranteed to be found. On the order of $O(cN)$ overlaps generally results comprise 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 for these $o.f.beg$ ($o.f.end$) or $o.g.beg$ ($o.g.end$) it is extreme, but not both.

Given the set of all overlaps 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 and the concatentation 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 their overlaps gives a significant practical reduction in the size and memory requirements for the problem.

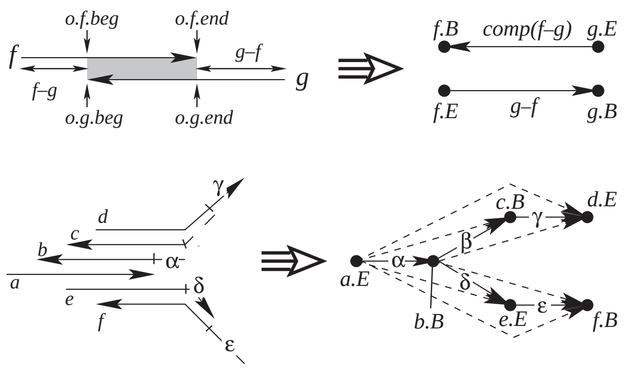

Fig. 2. String graph construction. At upper left is an overlap with its defining intervals and the left and right overhanging strings annotated. At the upper right are the two edges that should be placed in the string graph for the two overhangs of the overlap. At the lower left are some reads that overlap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

Henceforward 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 $g.B \xrightarrow{f[o.f.beg,0]} f.B$ and $f.E \xrightarrow{g[o.g.end,g.len]} g.E$
  else
    Add $g.E \xrightarrow{f[o.f.beg,0]} f.B$ and $f.E \xrightarrow{g[o.g.end,0]} g.B$
else
  if $o.g.beg < o.g.end$ then
    Add $f.B \xrightarrow{g[o.g.beg,0]} g.B$ and $g.E \xrightarrow{f[o.f.end,f.len]} f.E$
  else
    Add $f.B \xrightarrow{g[g.len,o.g.beg]} g.E$ and $g.B \xrightarrow{f[o.f.end,f.len]} f.E$
```

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. Consider any path $p = v_1 \rightarrow v_2 \rightarrow v_3 \cdots v_n$ and the reads $v_i.read$ of each vertex oriented as given if $v.type = E$ or complemented if $v.type = B$. By induction, the layout of these n reads induced by the $n-1$ edges of p form a valid contig that up to the sequencing error rate models the sequence of v_1 (or its complement depending on $v.type$) followed by sequence spelled along p. Since every overlap between reads is modeled in the graph, it follows that the original source sequence S and its complement are spelled by some path in the graph. We say that the graph is read coherent to mean that any path in the graph models a valid assembly of the reads. Note that this is not true of a de Bruijn graph built from k-mers of the reads and most of the effort for such approaches is in restoring this 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 any other, every composite edge $e = v_1 \rightarrow v_2 \rightarrow v_3 \cdots \rightarrow v_n$ has a complementary edge $comp(e) = comp(v_n) \rightarrow \cdots comp(v_3) \rightarrow comp(v_2) \rightarrow comp(v_1)$, 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 can model endpoint pairs as a single, read vertex with bidirected edges between them corresponding to an edge and its complement. Bidirected edges have an arrowhead at each end that can independently be directed in or out of the vertex at each end of the edge. The 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 out of the vertex otherwise. This framework was first introduced by this author and Kececioglu (Kececioglu and Myers, 1995). The concepts of in-degree and out-degree of a vertex still make sense and a path is a subgraph, where every interior vertex has in-degree and out-degree exactly one. Figure 3 gives an example of our construction for the genome *Canpylobacter jejuni*, although due to scale we suggest the reader look at Figure 5 in order to see an example of bidirected edges.

The bidirected string graph typically has between two to three orders of magnitude fewer vertices and edges than the number of reads and overlaps giving a significant reduction in the complexity of the problem. Moreover 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 its generalization of the de Bruijn graph should be apparent.

3 A LINEAR TRANSITIVE REDUCTION ALGORITHM

General transitive reduction of a graph takes $O(\Sigma_{v \rightarrow w \in E} \text{deg}(w)) = O(ED)$ time, where $\text{deg}(w)$ 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 $O(\Sigma_v \text{ tr.deg}(v)\text{deg}(v))$ worst case time, where $\text{tr.deg}(v)$ is the out degree of v in the transitively reduced graph. Assuming all input sequences are equally likely, $\text{tr.deg}(v) = O(1)$ 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.

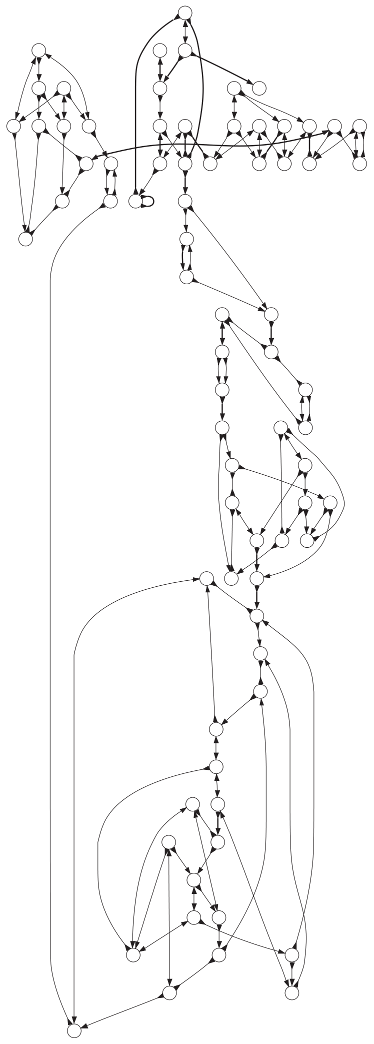

Fig. 3. The bidirected string graph of *C.jejuni* (prior to traversal analysis and compression).

Consider the edges out of a vertex $v : v \rightarrow w_1, v \rightarrow w_2, \ldots, v \rightarrow w_n$. Let $\text{len}(v \rightarrow w)$ 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 $\text{tr.deg}(v)$ 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 $w_1 \rightarrow w_2, \ldots, w_1 \rightarrow w_n$ must be in the graph. In general, suppose $\text{tr.deg}(v) = k$. 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 4, correctly identifies the heads of the irreducible edges.

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).

```
     constant $FUZZ \leftarrow 10$
1.   for $v \in V$ do
2.     { mark[v] $\leftarrow$ vacant
3.       for $v \rightarrow w \in E$ do
4.           reduce[$v \rightarrow w$] $\leftarrow$ false
       }

5.   for $v \in V$ do
6.     { for $v \rightarrow w \in E$ do
7.           mark[w] $\leftarrow$ inplay
8.       longest $\leftarrow max_w len(v \rightarrow w) + FUZZ$
9.       for $v \rightarrow w \in E$ in order of length do
10.          if mark[w] = inplay then
11.              for $w \rightarrow x \in E$ in order of length and
12.                      $len(w \rightarrow x) + len(v \rightarrow w) \leq longest$ do
13.                  if mark[x] = inplay then
14.                      mark[x] $\leftarrow$ eliminated

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

20.      for $v \rightarrow w \in E$ do
21.        { if mark[w] = eliminated then
22.              reduce[$v \rightarrow w$] $\leftarrow$ true
23.          mark[w] $\leftarrow$ vacant
           }
       }
```

Fig. 4. Transitive reduction algorithm.

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 an 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. Second 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 of 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 those edges that with very high probability should be traversed exactly once, i.e. those which 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

$$\binom{n}{k}\left(\frac{\Delta}{G}\right)^k\left(\frac{G-\Delta}{G}\right)^{n-k},$$

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

Unfortunately, we do not know the size of the genome G. An inspection of the typical string graph reveals that most of the total lengths 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 10 kb, 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-24 million 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 that just one iteration 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 identify every edge of the string graph that is with extremely high probability a single copy edge, and label it an (=1)-edge. Of the remaining edges, observe that those that are composite and have 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 conclude that every such edge must be used at least once and we label it a ($\geq$1-edge. All other edges, those that do not have interior vertices, do not have to be in a solution path and are labelled ($\geq$0)-edges.

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 an upper bound on the number of times it must be traversed in a reconstruction. Specifically, there are three cases: (1) the edge must be traversed = 1 time, (2) the edge must be traversed $\geq$1 time or (3) the edge must be traversed $\geq$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 is 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. If the underestimation were uniform accross all edges then there would be no problem, but unfortunately 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,

$$H_E(f,o) = \begin{cases} o.g.len - o.g.end & \text{if } o.g.beg < o.g.end \\ o.g.end & \text{otherwise} \end{cases}$$

$$O_E(f) = o \text{ s.t. } H_E(f,o) \text{ is smallest over all } o \text{ for which } o.g \text{ contains } f \text{ and } o.g \text{ is not itself contained}$$

$$V_E(f) = \begin{cases} O_E(f).g.E & \text{if } O_E(f).g.beg < O_E(f).g.end \\ O_E(f).g.B & \text{otherwise} \end{cases}$$

$$D_E(f) = H_E(f, O_E(f)).$$

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:

```
if $\text{len}(w \to v) < d$ then
    $\text{Map}(v,d) = \bigcup_{w \to v} \text{Map}(w, d - \text{len}(w \to v))$
else    $\{w \to v\}$
```

If a contained read's 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 $1/|\text{Map}(V_E(f), D_E(f))|$. 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 $<$1/100-th 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 wherein 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. The use of network flow is suggested in (Pevzner *et al.*, 2001), but without elaboration.

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 wherein we understand there is 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 (Ahuja *et al.*, 1993) 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):

$$c(e) = u(e) - l(e)$$

$$b(v) = \Sigma_{u\to v}l(u \to v) - \Sigma_{v\to w}l(v \to u)$$

$$v(e) = 1$$

$$t(e) = l(e) + x(e)$$

The particularly simple structure of our edge bounds, =1, ≥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 integral values as per the formula immediately above.

Before invoking an existing algorithm (Ahuja *et al.*, 1993) 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 $v \to w$ and $b(v) > 0$, then add $b(v)$ to $x(v \to w)$, add or subtract $b(v)$ to $b(w)$ depending on the direction of the arrowhead at w (i.e. in(+), out(−)), and set $b(v)$ to 0.
4. If a vertex v has a single in-edge $u \to v$ and $b(v) < 0$, then add $b(v)$ to $x(u \to v)$, add or subtract $b(v)$ to $b(u)$ depending on the direction of the arrowhead at u (i.e. in(−), out(+)), and set $b(v)$ to 0.

We call the edgess that remain, after the simplifications above, non-trivial edges and the vertices that have non-zero supply/demand unsatisfied vertices. While these transformations are all quite simple, on assembly string graphs they are very effective in reducing the size of the problem, in particular, the edge removals turn the graph into a collection of small connected components with few unsatisfied vertices, each of which is more efficiently solvable with the full, standard algorithms. Basically, one needs to push flow between unsatisfied vertices in a minimal cost way. Flow-pushing algorithms generally perform much better than their worst case complexity in such scenarios. Indeed, in some cases, we find components that have no unsatisfied vertices, in which case the solution is trivial. For an example of the final string graph see Figure 5.

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 highly 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.

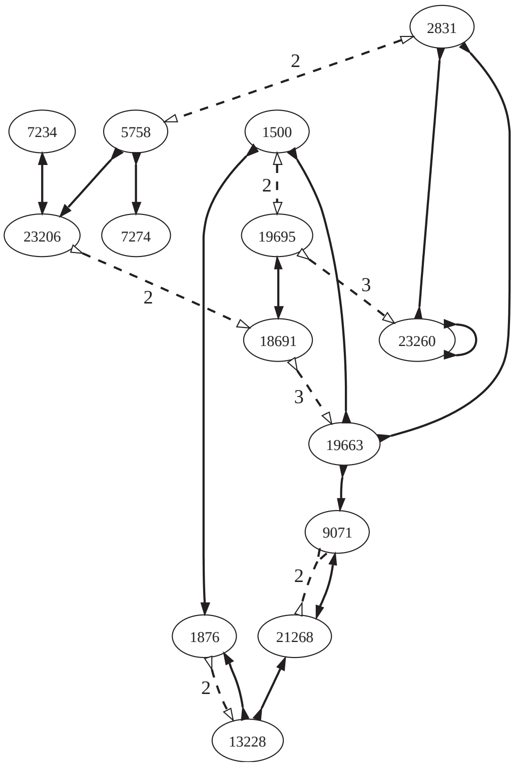

Fig. 5. The final *C.jejuni* string graph with traversal counts. The dashed edges are those that need to be traversed more than once and their traversal counts label them.

We consider simulated shotgun datasets of three target genomes: a 500 kb synthetic genome with 10 copies of a 300 bp repeat at 2% variation ('synthetic alus'), the 1.64 Mb sequence of the bacteria *C.jejuni*, and the first 13.9 Mb of the euchromatic sequence of arm 3R of *Drosophila melanogaster*. For each genome we synthetically sampled a 10× dataset of reads of length chosen uniformly between 550 and 850 bp. Each read has errors introduced with probably 0.008 at its 5′ end linearly ramping to .025 at its 3′ end. We used the *celsim* simulator (Myers, 1999).

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 realized here is conservative compared to what we observe for real data. The fourth

Table 1. Computational results on three simulated shotgun datasets

	Synthetic alus	*C.jejuni*	*D.melanogaster* 3R
Genome size (Mbp)	0.500	1.641	13.919
Reads	7000	23 900	202 200
Overlaps	127 K	462 K	3997 K
Contained reads (%)	41.4	42.6	43.1
Relevant overlaps	44 K (41%)	150 K (43%)	1268 K (43%)
Irreducible edges	8310	27 500	231 096
Junction vertices	33	75	756
Composite edges	89	113	1294
Size estimate (Mbp)	0.499	1.626	13.765
(=1)-edges	11	20	179
Non-trivial edges	2	51	538
No. of components	1	4	33
Unsatisfied vertices	0	6	109
Time (s)	3.7	13.1	113.6
Space (Mb)	0.53	1.81	15.28

Table 2. Containment mapping for *D.melanogaster* 3R dataset

|Map|	Containment endpoints
1	173 399
2	642
3	173
4	70
5	18
6	6
7	5
8	22
9	3

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 1200 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.

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.melanogaster* 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.

The shape and size of the string graph for *C.jejuni* is shown in Figure 3, after transitive reduction and collapsing, and the final solution after flow analysis in Figure 5. 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 probably be readily apparent.

8 FUTURE WORK

We are developing an open-source pipeline called BOA (Berkeley Open Assembler) with a very compact code base and clean, data-defined interfaces. Our primary efforts are on (1) developing a 'scrubber' that removes vector, chimers and low quality segments of reads, (2) sequence error correction, (3) using mate-pairs to further resolve the solution path through the string graph and (4) addressing the issue of polymorphism with a more sophisticated network flow approach. Additional modules are contemplated and could be incorporated by third parties.

ACKNOWLEDGEMENTS

The author wishes to thank members of his group, Kevin Lacker, Stuart Davidson and Michael Brudno for many helpful discussions and Michael in particular for suggesting the use of min-cost network flow for determining traversal counts.

Conflict of Interest: none declared.

REFERENCES

Adams,M.D. *et al*. (2000) The genome sequence of *Drosophila melanogaster*. *Science*, **24**, 2185–2195.

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

Aparicio,S. *et al*. (2002) Whole-genome shotgun assembly and analysis of the genome of *Fugu rubripes*. *Science*, **23**, 1301–1310.

Huang,X. *et al*. (2003) PCAP: A whole-genome assembly program. *Genome Res.*, **13**, 2164–2170.

Idury,R.M. and Waterman,W.S. (1995) A new algorithm for DNA sequence assembly. *J. Comp. Bio.*, **2**, 291–306.

Jaffe,D.B. *et al*. (2003) Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Res.*, **13**, 91–96.

Kececioglu,J. and Myers,E.W. (1995) Combinatorial algorithms for DNA sequence assembly. *Algorithmica*, **13**, 7–51.

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

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

Myers,E.W. (1995) Toward simplifying and accurately formulating fragment assembly. *J. Comp. Bio.*, **2**, 275–290.

Myers,E.W. (1999) A dataset generator for whole genome shotgun sequencing. In *proceedings of the Conference on Intelligent Systems for Molecular Biology*, Heidelberg, Germany, pp. 202–210.

Myers,E.W. *et al*. (2000) A whole-genome assembly of *Drosophila*. *Science*, **24**, 2196–2204.

Pevzner,P.A. *et al*. (2001) An Eulerian path approach to DNA fragment assembly. *Proc. Natl Acad. Sci. USA*, **14**, 9748–9753.

Rasmussen,K., Stoye,J. and Myers,E.W. (2005) Efficient q-gram filters for finding all e-matches over a given length. In *proceedings of the 9th Conference on Computational Molecular Biology*, Boston, MA, pp. 189–203.

Venter,J.C. *et al*. (2001) The sequence of the human genome. *Science*, **16**, 1304–1351.

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