Hapsembler: An Assembler for Highly Polymorphic Genomes

Nilgun Donmez and Michael Brudno

Department of Computer Science, The Donnelly Centre and Banting & Best Department of Medical Research, University of Toronto
{nild,brudno}@cs.toronto.edu

Abstract. As whole genome sequencing has become a routine biological experiment, algorithms for assembly of whole genome shotgun data has become a topic of extensive research, with a plethora of off-the-shelf methods that can reconstruct the genomes of many organisms. Simultaneously, several recently sequenced genomes exhibit very high polymorphism rates. For these organisms genome assembly remains a challenge as most assemblers are unable to handle highly divergent haplotypes in a single individual. In this paper we describe Hapsembler, an assembler for highly polymorphic genomes, which makes use of paired reads. Our experiments show that Hapsembler produces accurate and contiguous assemblies of highly polymorphic genomes, while performing on par with the leading tools on haploid genomes. Hapsembler is available for download at http://compbio.cs.toronto.edu/hapsembler.

Keywords: Genome assembly, polymorphism.

1 Introduction

In the last decade the sequencing and assembly of genomes have become routine biological experiments. Almost all genome projects today use the whole-genome shotgun sequencing approach: many copies of the genome are randomly sheared, and each part is sequenced to generate a read. Based on the overlaps between these reads an assembly algorithm then reconstructs contigs (contiguous regions) of the original, sequenced genome. Genome assembly has been the topic of extensive research, with a plethora of off-the-shelf methods that can reconstruct the genomes of many species. Simultaneously, de novo assembly of some organisms remains a challenge. Assembly tools currently available are optimized for the assembly of large mammalian genomes [1], [8], or smaller, bacterial genomes from High Throughput Sequencing (HTS) data [14], [3], [2]. In both of these settings the difficulty of genome assembly is due to the relative sizes of the reads and the genome: short reads cannot span longer repeats, and make the assembly of the genome difficult. While many of these assembly tools also consider the possibility of polymorphisms in the sequenced individuals, the low frequency of SNPs and other variants in these genomes makes addressing these polymorphisms a relatively tractable problem.

Simultaneously, there are now several known organisms with extremely high polymorphism rates [11],[13]: for example the C. savignyi genome has a SNP rate of 5% (50-fold higher than human). Because this variability is also present between the sequenced individual's paternal and maternal chromosomes, assembling these genomes is very difficult with current methods. Previous efforts to assemble these highly polymorphic genomes have used one of two approaches: either they allow for promiscuous overlaps, to connect the reads from different haplotypes[20], or they enforce strict overlapping requirements, thus separating the haploid genomes (haplomes) [10]. The first approach has the drawback that the spurious overlaps make it difficult to reconstruct long segments of the genome. The second approach will unnecessarily subdivide the genome, as many real overlaps will be removed. Furthermore, higher coverage will be needed, as the effective total size of the genome is doubled. In the C. savignyi genome assembly the two haplomes were aligned to each other to better reconstruct the genome [12], however only 60% of the genome could be assembled automatically, and the rest was finished via manual analysis of the contigs [10].

In this paper we present Hapsembler, a haplotype-aware genome assembly algorithm. Hapsembler combines the classical overlap-layout-consensus approach for genome assembly with a haplotype-aware, Bayesian approach for error correction and a novel structure we term the matepair graph, that helps to reconstruct the genome via a thorough analysis of the paired sequencing reads.

2 Methods

Hapsembler consists of three main modules: the alignment module, the error correction module, and the graph module. The alignment module is used to compute pairwise sequence alignments between reads that obey certain criteria. This procedure employs an efficient kmer hashing technique to detect overlaps longer than a user defined length. This initial set of overlaps is used to correct sequencing errors by a probabilistic error correction procedure based on Naive Bayes [15]. The corrected reads are then passed through another overlapping stage to compute the final set of overlaps.

The graph module builds an overlap graph (a.k.a string graph [7]) using the overlaps reported by the alignment module. This graph is used to construct "path sets" representing possible tilings between paired reads. A mate pair graph is then constructed in which nodes represent mate pairs, and edges represent possible overlaps between mate pairs as constrained by the path sets. Finally, contigs are determined by finding maximal paths in the mate pair graph.

In the following subsections we give a detailed explanation of these steps.

2.1 Overlap Computation

To compute overlaps between the reads, we employ a kmer hashing technique similar to [9]. After masking frequent kmers, reads that share a sufficient number of kmers (see Appendix 1 for these and other parameters) are aligned using a modified Needleman-Wunsch algorithm. This modified algorithm uses the positions of matching kmers to estimate the overlapping portion of the reads. For instance, if a kmer is at index 60 in read R and at index 45 in read R', the start of the reads are assumed to be 15 base pairs apart. Consequently, only the corresponding diagonal and diagonals within a distance of d are computed in the dynamic programming matrix. The parameter d is dynamically set to $e * l$, where e is a user defined error tolerance rate and l is the length of the read. In general, different kmers that are shared between two reads may suggest different starting indices. To prevent redundant calls to the Needleman-Wunsch procedure, we bundle such indices based on their proximity. Overlaps greater than a minimum length l_{min} with identity $(1 - e)$ or more are reported.

2.2 Error Correction

Real sequence reads often have errors even after aggressive quality trimming. To correct these errors at an early stage, we employ a double pass overlap computation approach. In this approach, the initial set of overlaps are used to correct the reads. For each read, the set of all pairwise alignments initially found are used to decide, for each base of the read, whether the position is correct or is a sequencing error. Note that, in the case of diploid organisms, error correction is further complicated, as disagreements between reads may be due to either errors, or polymorphisms (SNPs or small indels). To avoid overcorrecting reads with SNPs, we employ a Naive Bayesian method at this stage.

Briefly, this method works by scoring each pairwise alignment using base quality values and using this score to derive the probability that two reads are sampled from the same haploid genome (haplome). The consensus sequence for the read is then computed by weighting each alignment's contribution with this probability.

Formally, let $C_x \in \{A, C, G, T\}$ denote the x^{th} base of a read R. Ambiguous bases (eg. N) are not used to correct other errors; they can, however, themselves be corrected. Suppose that R has pairwise alignments with the reads $\{S^1, S^2, ..., S^n\}$. Let F_i denote the base aligned with x in the pairwise alignment between R and S^i. Then the probability of C_x assuming a particular nucleotide is given by:

$$p(C_x|F_{i=1,2,...,n}) = p(C_x) \prod_{i=1}^{n} p(F_i|C_x) \tag{1}$$

Above $p(C_x)$ is the prior probability of C_x and it is equal to $1 - 10^{-q/10}$ if it has the same value as the nucleotide present in the read, where q is the Phred style [16] quality score associated with the base. Otherwise it is equal to $(10^{-q/10})/3$. $p(F_i|C_x)$ is given by the equation:

$$\begin{align} p(F_i|C_x) &= p(F_i|C_x, B_i)p(B_i) + p(F_i|C_x, \neg B_i)p(\neg B_i) \tag{2} \\ &= p(F_i|C_x, B_i)p(B_i) + p(F_i)p(\neg B_i) \tag{3} \end{align}$$

$p(F_i|C_x, B_i)$ is the conditional probability of F_i given C_x and B_i where B_i is a binary variable indicating whether the two reads belong to the same locus of the genome/haplome or not. When two reads belong to different loci, the bases are independent of each other hence $p(F_i|C_x, \neg B_i) = p(F_i)$ (i.e. we have conditional independence). Here, we abuse the notation slightly and use $p(B_i)$ to denote the posterior probability of S^i and R belonging to the same locus. To estimate $p(B_i)$, we first have to compute the following probabilities:

$$p(B_{R,S^i}|R, S^i) = p(B) \prod_{j=1}^{k} p(R_j, S^i_j|B) \tag{4}$$

$$p(\neg B_{R,S^i}|R, S^i) = p(\neg B) \prod_{j=1}^{k} p(R_j, S^i_j|\neg B) \tag{5}$$

$$= p(\neg B) \prod_{j=1}^{k} p(R_j)p(S^i_j) \tag{6}$$

Above, k is the number of bases in the pairwise alignment between R and S^i. $p(R_j, S^i_j|B)$ denotes the conditional probability of the j^{th} position in the alignment. If the reads R and S^i belong to the same region, this means the disagreements between the two sequences should be due to sequencing errors alone. In other words, if the two bases differ, at least one of them must be an error. Let q denote the quality value of the j^{th} base in read R and q^i denote the quality value of the corresponding base in read S^i. If $R_j \neq S^i_j$:

$$p(R_j, S^i_j|B) = (1 - 10^{-q/10})((10^{-q^i/10})/3) \tag{7}$$

$$+(1 - 10^{-q^i/10})((10^{-q/10})/3) \tag{8}$$

$$+2((10^{-q/10})/3)((10^{-q^i/10})/3) \tag{9}$$

If $R_j = S^i_j$, on the other hand, either both reads are correct or they both have a sequencing error:

$$p(R_j, S^i_j|B) = (1 - 10^{-q/10})(1 - 10^{-q^i/10}) \tag{10}$$

$$+((10^{-q/10}))((10^{-q^i/10})/3) \tag{11}$$

The prior probability $p(B)$, of two reads belonging to the same location is set to a value near 1.0 since we only compare reads that have a sufficient overlap. The posterior probability $p(B_i)$ is then estimated as:

$$p(B_i) = \frac{1}{1 + exp(\log p(\neg B_{R,S^i}|R, S^i) - \log p(B_{R,S^i}|R, S^i))} \tag{12}$$

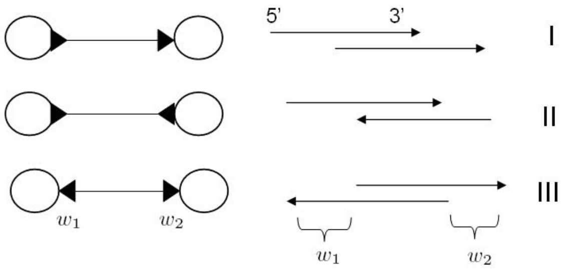

Fig. 1. *Possible edges in a bidirected overlap graph.* The directed lines to the right represent the reads where the arrowed end is the 3' end while the flat end is the 5' end. w_1 and w_2 are the lengths of the reads that are not covered by the overlap.

The consensus nucleotide of read R for position x is chosen to be the nucleotide that gives the highest probability $p(C_x|F_{i=1,2,...,n})$. In practice, we use the log probabilities to avoid multiplication. We also compute a look-up table in advance to avoid doing the same computations many times.

Note that the equations above do not account for indel (insertion/deletion) errors. Although indels could be handled similarly, there are no associated quality values with missing bases. Consequently, we handle indels separately. If a significant fraction of reads are calling for a deletion (at least 3 votes for deletion and at most 1 vote for no deletion; these parameters are conservative for the relatively low coverage levels used in this study, to avoid over-correction) the base is deleted. A similar rule is applied for insertion and the insertion base is selected using the same procedure as above where $\log p(C_x)$ is taken to be $\log(1/4)$. For the computation of $p(B_i)$, we use a default gap quality value. After all the reads are corrected as above, we do another pass of the overlapping phase, now with the corrected reads.

2.3 Building the Overlap Graph

Once the overlaps between the reads are finalized we build a bidirected overlap graph where the nodes are reads and the edges represent overlaps between the reads. Formally, a bidirected graph G is a graph with node and edge sets V and E, where each edge can acquire either of the two types of arrows at each vertex; *in-arrow* and *out-arrow* [6], [19]. As a result, there are 3 possible types of edges in a bidirected graph; *in-out*, *out-out* and *in-in* (Figure 1). A walk in a bidirected graph must obey the following rule: If we come to a node using an in-arrow we must leave the node using an out-arrow. Similarly, if we come to a node using an out-arrow we must leave using an in-arrow. In the former case, the node is said to be *in-visited* and in the latter case it is said to be *out-visited.*

The Figure 1 omits non-proper overlaps that are caused by reads that are entirely contained by other reads. These reads are excluded when building the overlap graph.

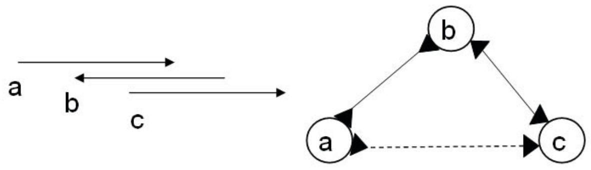

Fig. 2. *Transitive edge reduction.* Since we can reach the node **c** from **a** via **b**, we do not need the edge between **a** and **c**. Note that during this procedure, we check whether the two paths have the same overall length within a permissible difference f, where f is defined as the total number of indels that are present in the pairwise alignments associated with the overlaps. Otherwise, the edge is not deleted.

Many edges in the initial overlap graph will be redundant since they can be inferred by other edges. These edges are removed from the graph using an operation called *transitive edge reduction* [7] as illustrated in Figure 2.

2.4 Finding Paths between Mate Pairs

At this stage, we have a simplified overlap graph in which most nodes have 1 incoming and 1 outgoing edge. However, some nodes will have degrees of 3 or more due to repeats or polymorphic regions. In this section, we will show how to use the mate pairs to resolve such nodes to generate longer paths.

For each mate pair in our dataset, we assume there is a given approximate insert size mean μ_l and standard deviation σ_l. Ideally, we would like to find a single path between each pair with a length in the range $\mu_l \pm (k\sigma_l)$, where k is a real number controlling the largest deviation from the mean we are willing to allow. In general, there may be an exponential number of such paths in the graph. However, we can identify the subgraph containing all paths between two nodes *shorter than* a given length in polynomial time.

This idea can be summarized as follows. Let nodes a and a' be a mate pair. First, we perform Dijkstra's [4] shortest path finding algorithm starting from a (and leaving the node using an out-arrow). While Dijkstra's algorithm is originally invented for directed or undirected graphs, the generalization to bidirected graphs is straightforward. The only difference is that instead of a single distance from the source, we have to keep track of the shortest in-distance and out-distance separately for each node. We also modify the algorithm so that only the nodes that are within a distance of $\mu^* = \mu_l + (k\sigma_l)$ are enqueued. During this search, if we do not encounter a' it means there is no path in the graph between a and a' less than the given length. This situation can arise for several reasons: (1) the insert size deviation might be higher than the expected for that mate pair, (2) there might be a region with no coverage between the mates, or (3) we might be missing overlaps (due to sequencing errors, short overlaps, etc). If we encounter a' during the search, we do another pass of Dijkstra's, this time starting from a'. During this second pass, we enqueue a node if and only if the sum of its shortest distance from a, its current distance from a' and the read's

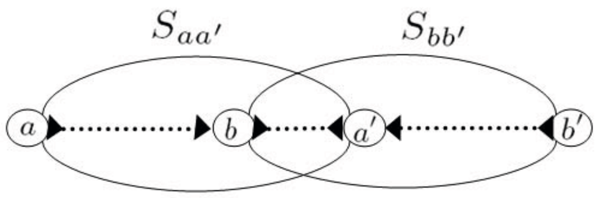

Fig. 3. *Overlapping mate pairs.* By comparing the previously computed sets of $S_{aa'}$ and $S_{bb'}$, we can determine whether the two mate pair nodes should be connected or not.

length is less than μ^*. Furthermore, we put such nodes into a set which we shall call $S_{aa'}$ together with their shortest in/out-distances from a and a'.

After this second pass, we end up with a set of nodes, $S_{aa'}$, that are guaranteed to lie on at least one path that has length less than μ^* between a and a'. It is easy to show that this set is also exhaustive; that is all vertices v that satisfy $indist(a, v) + outdist(a', v) + length(v) < \mu^*$ or $outdist(a, v) + indist(a', v) + length(v) < \mu^*$, where $in/outdist(x, y)$ denotes the in/out distance of node y from node x, are included in $S_{aa'}$.

Note that we find this set of nodes in polynomial time even though there might be an exponential number of paths between a and a'.

2.5 Building the Mate Pair Graph

The process described above is repeated for each mate pair, yielding a set of sets $\bar{S}$. We then use these sets to build the mate pair graph.

Consider two mate pair sets $S_{aa'}$ and $S_{bb'}$. To decide if these mate pairs have paths that overlap with each other, we first check whether each of the following conditions hold:

$$a \in S_{bb'} \tag{13}$$
$$a' \in S_{bb'} \tag{14}$$
$$b \in S_{aa'} \tag{15}$$
$$b' \in S_{aa'} \tag{16}$$

Whenever there is less than two positive answers to the checks, an overlap of paths is not possible. Otherwise, we check if the length of the paths and orientations are compatible. For example, if we find that $a' \in S_{bb'}$ and $b \in S_{aa'}$, we check whether the following inequalities hold (Figure 3):

$$indist(a, b) + outdist(a', b) + length(b) < \mu^* \tag{17}$$
$$outdist(b, a') + indist(b', a') + length(a') < \mu^* \tag{18}$$

where μ^* is defined as above. Recall that we store these distances together with the nodes, hence these checks are done in constant time. However, this algorithm may become problematic in terms of memory for large insert sizes since we store a set proportional to the size of the insert for each mate pair (for fixed coverage and read length). In practice, we use a slightly different version of this algorithm which can be implemented to give linear space complexity independent of the insert size. In this version, we perform two extra Dijkstra's starting from each end of a mate pair, this time in opposite directions (i.e. leaving the node using an in-arrow). As before, we only enqueue nodes that are within the distance cutoff. This gives us two additional sets $S_{\hat{a}}$ and $S_{\hat{a'}}$. Then for each node b in $S_{aa'}$ we check if its mate pair b' is in $S_{\hat{a}}$ or $S_{\hat{a'}}$ depending on the orientation of the node. If the path lengths are compatible with the insert sizes (see above) then we put an edge between aa' and bb' in the mate pair graph. Since we can immediately determine which edges aa' should be incident to, we do not have to store the set of aa' after we process aa'. As a result, this alternative algorithm takes only linear space in the number of nodes.

2.6 Processing the Mate Pair Graph

The mate pair graph is structurally similar to the overlap graph and can undergo similar simplifying procedures, namely nested mate pair removal and transitive edge reduction.

When a mate pair lies entirely within an other mate pair, it creates an unnecessary bubble or dead end in the mate pair graph . Such nested mate pairs are detected and marked while building the mate pair graph. For a mate pair aa', this is done by checking the sets $S_{\hat{a}}$ and $S_{\hat{a'}}$ to see if there is any mate pair bb' such that $b \in S_{\hat{a}}$, $b' \in S_{\hat{a'}}$ and:

$$outdist(a, b) + dist(a, a') + outdist(a', b') < \mu^* \tag{19}$$

where $dist(a, a')$ is the shortest distance between the pair aa' as computed during Dijkstra's algorithm. If there is any such mate pair, aa' is removed from the mate pair graph.

Finally, we perform transitive edge reduction on the mate pair graph. When creating the mate pair graph, three values are stored with each edge. For example, for the mate pair edge between aa' and bb' of Figure 3, we would store the distances $indist(a, b)$, $outdist(b, a')$ and $indist(a', b')$. Given three mate pair nodes where each node is connected to the other two, we decide whether one of the edges can be inferred from the other two using these distances.

2.7 Polymorphism and Repeat Resolution with the Mate Pair Graph

In essence, polymorphism resolution is very similar to repeat resolution and Hapsembler is designed to exploit paired reads to handle both problems at once. Figure 4 illustrates how the graph module works on a toy example. In this

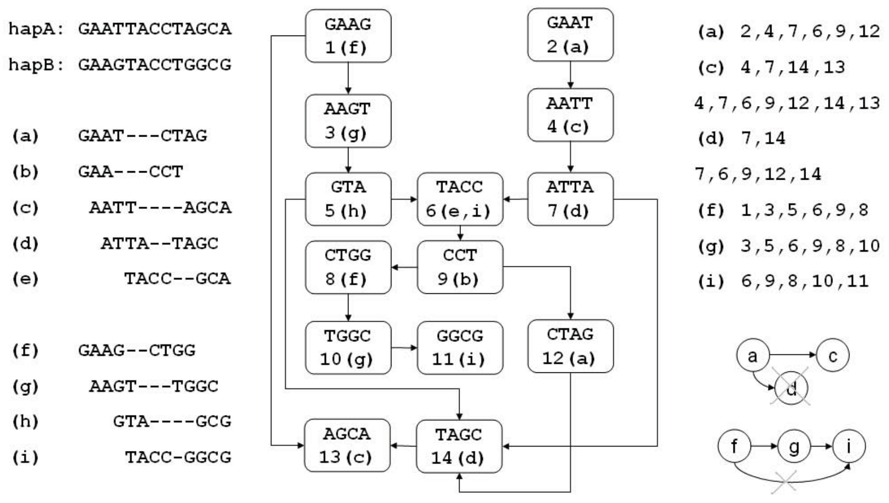

Fig. 4. *Polymorphism and repeat resolution.* **Left:** The diploid genome and mate pairs sampled from it. We do not know the exact distances between the pairs but we assume that we are given an upper bound (in this case 13bp). **Middle:** The overlap graph after removal of contained reads (i.e. GAA, GCA and GCG) and transitive edge reduction. The nodes are labelled with the mate pairs they belong to and arbitrarily numbered. The minimum overlap size is set to 2bp. **Right:** The paths between the mate pairs shorter than the given upper bound and the resulting mate pair graph. In practice, we do not need the exact paths and we only compute the set of nodes that lie on at least one path. The node d is removed since it is contained by node c. In addition, the edge between f and i is removed during transitive edge reduction. The resulting paths correspond to the two haplomes.

example, we have a diploid genome[1] which has several SNPs. After the overlap graph is built and simplified, we still have several ambiguous nodes (nodes with degree 3 or more). Some of these ambiguities are due to short repeats and some due to regions that are identical in both haplomes. Nevertheless, the mate pair graph built upon this overlap graph is less tangled. Indeed, the simplified mate pair graph has exactly two disjoint paths, each spelling the sequence of one haplome.

Although the toy example given in Figure 4 is completely resolved by our algorithm, in general the mate pair graph might still have ambiguous nodes. In particular, as with repeats, haplotype resolution is limited to the size of the inserts. After simplification, we report each uninterrupted path in the mate pair graph as a contig. If there are long chains in the simplified overlap graph that have not been visited during the mate pair graph traversal (or if all the reads are unpaired), these are also reported as contigs. The consensus sequence for each contig is generated using a greedy multiple sequence alignment.

[1] For simplicity, we assume the genome is single stranded.

3 Results

We analyzed the performance of Hapsembler in two categories: read correction and assembly. For each category, we performed experiments with simulated and real reads. Since Hapsembler is built for polymorphic data, we compiled a reference sequence using the C. savignyi reference assembly as described in [11]. This draft assembly is organized in 374 "hypercontigs", where each hypercontig is a pairwise alignment of two sequences, each representing a single haplotype. To use in our experiments, we picked the largest three hypercontigs, totalling roughly 33mbp (haploid size = 16.5mbp).

Since C. savignyi has a very complex genome with many long repeats and a very high polymorphism rate, we only used Sanger reads in our experiments with this genome. However, we also give results on a bacterial genome using Roche/454 reads. For these experiments we downloaded 454 reads from an ongoing sequencing project on evolution of antibiotic resistant E. coli (NCBI Short Read Archive, accession code: SRR024126). This dataset includes 110,388 reads amounting to 10x sequence coverage. The reads were downloaded using the clipping option and barcode trimmed. As our reference we used the NCBI reference E. coli sequence (NC_000913.2).

3.1 Error Correction

To test the effect of our error correction we first simulated Sanger reads at different coverage levels from the C. savignyi reference. For these simulations, we downloaded the raw Sanger reads from the C. savignyi sequencing project [18] to use as templates. Using the length of these reads as a distribution, we uniformly sampled Sanger reads generating 13x, 10x, and 7x haploid coverage levels. The templates are also used to model errors by converting the quality scores to error probabilities. For instance, if a base has quality 20, we introduce an error with probability 0.01.

We compare the performance of our error correction method to the method of [17] (referred to as H-SHREC throughout this text). We choose this implementation of SHREC for its ability to handle indel type errors. The results on simulated Sanger reads is summarized in table 1. Even though the reads are sampled from a highly polymorphic reference, Hapsembler is able to correct a large fraction of errors while introducing relatively few errors. H-SHREC fails to reduce the total numbers of errors for all three datasets.

Next we assessed the effect of our error correction on two real datasets. As our first dataset we used a subset of the real Sanger C. savignyi reads as follows. After vector and quality trimming, we mapped the reads to the reference sequence we described above using MUMmer (version 3.22)[5]. A read is considered eligible if at least 90% of its sequence maps to a location with a minimum of 95% identity. For each eligible read, we also included its mate pair. This procedure yielded 558,936 reads totalling 358mbp. As our second dataset we used the 454 E. coli reads described above.

Table 1. *The effect of error correction on simulated Sanger reads.* The reads are first quality trimmed and then subjected to error correction. Total bases is calculated as the sum of all of the reads after trimming. Miscorrections denote errors introduced by the error correction procedure. Number of errors and miscorrections are calculated via alignments to the original error-free reads.

Coverage	Total (mbp)	No. of errors after trimming	Method	No. of errors after correction	No. of miscorrections
13x	194	3,924,331	Hapsembler	598,111	45,911
			H-SHREC	4,750,235	2,221,266
10x	148	3,056,019	Hapsembler	631,957	48,997
			H-SHREC	4,370,374	2,443,059
7x	102	2,199,861	Hapsembler	741,785	51,746
			H-SHREC	2,893,788	1,453,007

Since in this case we do not have the ground truth, we assessed the performance of both methods by mapping the reads to the reference sequences before and after error correction. Results are summarized in table 2. In the C. savignyi dataset, after correcting with Hapsembler, the number of reads mapping perfectly increases by more than 6-folds. H-SHREC moderately improves the number of reads mapping perfectly, however the number of reads mapping at the 95% threshold decreases, suggesting that the overall number of errors might have increased. On the E. coli dataset, Hapsembler and H-SHREC perform similarly.

3.2 Assembly

We first evaluated the performance of Hapsembler on the 454 E. coli dataset. We compare the results achieved by Hapsembler with Velvet [14] and Euler [3], which have support for 454 reads. Since Hapsembler is designed to work with paired reads we also simulated an artificial pairing of these reads as follows. Reads are mapped to the reference sequence and sorted by their mapping positions allowing duplicate mappings. For each read mapping to the forward strand, an unpaired read mapping on the opposite strand that has distance closest to 8000bp is taken. If there is no such read with distance 8000 ± 2400, then the read is left unpaired. This mapping yielded 33,160 pairs with insert size mean and standard deviation of 8534.67 and 713.35 respectively. The rest of the reads are left as single reads. Table 3 show the results for contigs of size 500bp or greater.

To evaluate Hapsembler on a polymorphic genome, we simulated 13x coverage of paired Sanger reads from the C. savignyi reference haplomes with errors added as described above. The start of the first reads are chosen uniformly and the start the paired reads are selected using a normal distribution with mean and standard deviation 10kbp and 1kbp respectively. The results are summarized in table 4. Of particular emphasis for polymorphic assembly is the ability of an algorithm to report haplotype-specific contigs, rather than mozaics from the two

Table 2. *The effect of error correction on real Roche/454 and Sanger reads.* C. savignyi dataset consists of 558,936 Sanger reads and E. coli dataset consists of 110388 Roche/454 reads. Reads are mapped to the reference sequences using MUMmer. Number of reads mapped is calculated by counting the reads that map with at least 95% identity and 95% coverage. A read is considered to be perfect if the entire read maps with 100% identity. The numbers in paranthesis denote the number of discarded reads (by H-SHREC) that map in each category. H-SHREC discards a total of 13600 and 1132 reads from the C. savignyi and E. coli datasets respectively.

Data	Error correction	No. of reads mapped	No. of reads mapped perfectly	Total size of perfect reads (mbp)
C. savignyi	Hapsembler	421,819	126,306	83.9
	H-SHREC	(11,401) 391,016	(761) 48,994	32.6
	None	411,626	20,689	13.6
E. coli	Hapsembler	89,817	10,292	3.9
	H-SHREC	(738) 88,814	(10) 9,573	3.7
	None	88,624	4,154	1.6

Table 3. *Assembly of E.coli with real and artificially paired 454 reads.* Coverage and accuracy are computed by mapping contigs to E. coli reference sequence (4639kbp) using MUMmer. N50 is defined as the largest contig size such that the sum of contigs at least as long is greater than half the genome size.

Reads	Tool	N50 (kbp)	No. of Contigs	Total Size (kbp)	Coverage (%)	Accuracy (%)
unpaired	Hapsembler	72.4	128	4600	90.3	98.6
	Velvet	41.2	199	4585	89.5	98.4
	Euler	8.1	913	4731	88.6	98.6
paired	Hapsembler	103.7	111	4693	90.9	98.6
	Velvet	41.9	189	4607	89.5	98.2
	Euler	9.4	765	4593	88.4	98.6

haplotypes. However, conserved sequences and low coverage regions make this difficult, and the longer contigs may have several “jumps” between the haplomes. To estimate the long-range linkage information in the contigs, we computed maximal haplotype blocks in the assembly, for which all of the SNPs match one haplome. In Figure 5 we plot the fraction of genome covered by haplotype specific blocks of a certain size or greater, measured in the number of SNPs. The figure shows that half of the genome is covered by haplotype-specific blocks containing 300 adjacent SNPs or more.

Table 4. *Assembly of three hypercontigs of C. savignyi with simulated paired reads.* Contigs are mapped to the reference haplomes using MUMmer. Coverage is calculated by taking only the best hit of each location of the contigs. Accuracy is calculated as percent identity excluding SNPs. N50 is defined as the largest contig size such that such that the sum of contigs at least as long is greater than half the genome size, where the genome size is taken as the sum of the two haplomes.

Tool	N50 (kbp)	No. of Contigs	Total Size (mbp)	Coverage (%)	Accuracy (%)
Hapsembler	23.4	2886	34	87.4	99.4

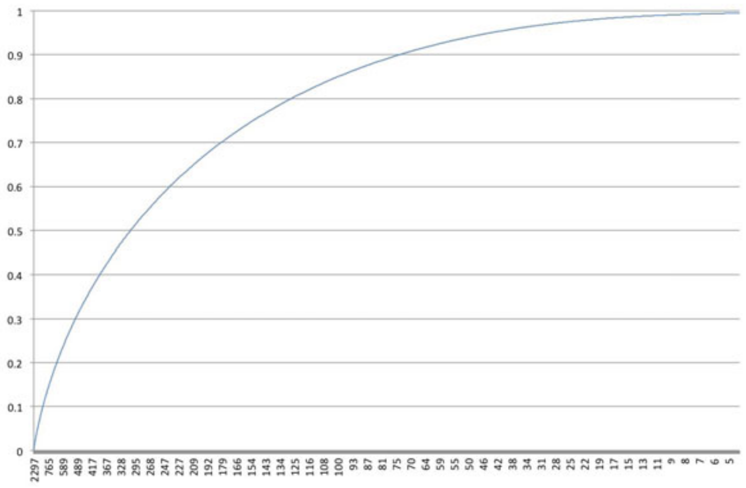

Fig. 5. *Fraction of genome in haplotype blocks.* X axis denote the number of adjacent SNPs covered by a contiguous region of a contig. Y axis shows the fraction of genome covered by the haplotype blocks.

4 Discussion

In this paper we presented Hapsembler, an assembly algorithm for whole genome shotgun data that is optimized for highly polymorphic genomes. Due to the large number of differences between the maternal and paternal copies of the chromosomes, these genomes have classically been difficult to assemble, with custom algorithms [12] and extensive manual intervention [10] required to achieve a high quality assembly. Hapsembler, to our knowledge, is the first tool that specifically targets this problem.

Hapsembler combines the use of mate pairs with a sophisticated error correction procedure to achieve a better assembly. Nevertheless, the methods required for this improvement are computationally expensive. Currently the most time consuming steps are read overlapping and mate pair graph building stages. While Hapsembler takes 36 minutes to assemble the E. coli dataset, Euler and Velvet take only a few minutes each. Fortunately, both of these bottlenecks are suitable for parallel computation. For example, the overlapping stage of 558k reads from the C. savignyi dataset takes less than 40 minutes using four quad-core Intel 3 GHz Xeon compute nodes. Nevertheless, further improvements are necessary to make Hapsembler work on large scale whole-genome datasets. Similarly, additional future work is necessary to take advantage of high-coverage High Throughput Sequencing reads (Solexa/Illumina or AB/SOLiD) in combination with lower-coverage Sanger reads. Finally we believe that exploring additional representation methods for polymorphic genome assemblies is a fruitful area for future research. We believe that representing haplotypes as paths on a genome graph (similar to Allpaths [2]) may allow for representation of the inherent complexity of polymorphic genomes.

Acknowledgements

This research was supported by an NSERC Discovery Grant to MB.

References

1. Batzoglou, S., Jaffe, D.B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J.P., Lander, E.S.: ARACHNE: A Whole-Genome Shotgun Assembler. Genome Research 12, 177–189 (2002)
2. Butler, J., et al.: ALLPATHS: De novo assembly of whole-genome shotgun microreads. Genome Research 18, 810–820 (2008)
3. Chaisson, M.J., Pevzner, P.A.: Short read fragment assembly of bacterial genomes. Genome Research 18, 324–330 (2008)
4. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
5. Kurtz, S.: Versatile and open software for comparing large genomes. Genome Biology 5, R12 (2004)
6. Medvedev, P., Georgiou, K., Myers, E.W., Brudno, M.: Computability of Models for Sequence Assembly. In: Giancarlo, R., Hannenhalli, S. (eds.) WABI 2007. LNCS (LNBI), vol. 4645, pp. 289–301. Springer, Heidelberg (2007)
7. Myers, E.W.: The fragment assembly string graph. Bioinformatics 21(2), 79–85 (2005)
8. Myers, E.W., et al.: A Whole-Genome Assembly of Drosophila. Science 287(5461), 2196–2204 (2000)
9. Rasmussen, K., Stoye, J., Myers, E.W.: Efficient q-Gram Filters for Finding All e-matches Over a Given Length. J. of Computational Biology 13, 296–308 (2005)
10. Small, K.S., Brudno, M., Hill, M.M., Sidow, A.: A haplome alignment and reference sequence of the highly polymorphic Ciona savignyi genome. Genome Biology 8(1) (2007)
11. Small, K.S., Brudno, M., Hill, M.M., Sidow, A.: Extreme genomic variation in a natural population. PNAS 104(13), 5698–5703 (2007)

12. Sundararajan, M., Brudno, M., Small, K., Sidow, A., Batzoglou, S.: Chaining Algorithms for Alignment of Draft Sequence. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 326–337. Springer, Heidelberg (2004)
13. Weinstock, G.M., et al.: The Genome of the Sea Urchin Strongylocentrotus purpuratus. Science 314, 941–952 (2006)
14. Zerbino, D.R., Birney, E.: Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. Genome Research 18, 821–829 (2008)
15. Domingos, P., Pazzani, M.: On the Optimality of the Simple Bayesian Classifier under Zero-One Loss. Machine Learning 29(2-3), 103–130 (1997)
16. Ewing, B., Hillier, L., Wendl, M.C., Green, P.: Base-calling of automated sequencer traces using phred. II. Error probabilities. Genome Research 8, 175–185 (1998)
17. Salmela, L.: Correction of sequencing errors in a mixed set of reads. Bioinformatics 26, 1284–1290 (2010)
18. Ciona savignyi database at Broad Institute, `http://www.broadinstitute.org/annotation/ciona/`
19. Kececioglu, J.: Exact and Approximation Algorithms for DNA Sequence Reconstruction. PhD dissertation, Technical Report 91-26, Department of Computer Science, University of Arizona (December 1991)
20. Dehal, P., et al.: The Draft Genome of Ciona intestinalis: Insights into Chordate and Vertebrate Origins. Science 298(5601), 2157–2167 (2002)

Appendix 1: Parameters

To run H-SHREC (version 1.0) we use the largest strictness value the program accepts for each dataset. For the C. savignyi datasets, we set the number of iterations to 1 (more iterations introduced more errors). For E. coli, 3 iterations are used. The other parameters are left at defaults. Velvet (version 1.0.13) is tested with all odd kmer sizes between 17 and 27 inclusive. The results are reported with the kmer size (19) that achieved the highest N50 value. The expected coverage is set to 10. For Euler (version 1.1.2), we test all odd kmer sizes between 21 and 27 and choose the size (23) that maximized the N50 value.

Hapsembler is run with an error threshold of 0.07 and minimum overlap size of 30bp. The kmer size is set to 13. Kmers that appear more than 100 times the expected coverage in the data are masked. The minimum number of kmers required to perform Needleman-Wunsch is set to 1. These parameters are kept constant for all the datasets reported.