Bioinformatics Advance Access published December 29, 2012

SCARPA: Scaffolding Reads with Practical Algorithms

Nilgun Donmez[1*] and Michael Brudno[1,2,3]

[1]Department of Computer Science, University of Toronto, Toronto, ON, Canada
[2]Donnelly Centre, University of Toronto, Toronto, ON, Canada
[3]Center for Computational Medicine, Hospital for Sick Children, Toronto, ON, Canada

Associate Editor: Dr. Alex Bateman

ABSTRACT
Motivation: Scaffolding is the process of ordering and orienting contigs produced during genome assembly. Accurate scaffolding is essential for finishing draft assemblies as it facilitates the costly and laborious procedures needed to fill in the gaps between contigs. Conventional formulations of the scaffolding problem are intractable and most scaffolding programs rely on heuristic or approximate solutions with potentially exponential running time.
Results: We present SCARPA, a novel scaffolder which combines fixed-parameter tractable and bounded algorithms with Linear Programming to produce near-optimal scaffolds. We test SCARPA on real datasets in addition to a simulated diploid genome and compare its performance to several state-of-the-art scaffolders. We show that SCARPA produces longer or similar length scaffolds that are highly accurate compared to other scaffolders. SCARPA is also capable of detecting misassembled contigs and reports them during scaffolding.
Availability: SCARPA is open source and available from http://compbio.cs.toronto.edu/scarpa.
Contact: nild@cs.toronto.edu
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 INTRODUCTION

While assemblers developed for High Throughput Sequencing (HTS) platforms can produce high quality draft assemblies for the genomes of bacteria and viruses, *de novo* assemblies of more complex genomes using short reads are typically very fragmented. This fragmentation can be partially alleviated through scaffolding: the process of linking contigs into longer sequences (possibly with gaps) using paired read information. Scaffolding not only improves the contiguity of the initial assembly, but is also helpful for designing experiments for finishing of the genome through additional sequencing of selected regions with lower throughput technology, such as Sanger.

Although many genome assemblers produce scaffolds using paired reads during the assembly process (Zerbino *et al.*, 2009; Simpson *et al.*, 2009; Li *et al.*, 2009), the problems of building contigs and scaffolding them are distinct. A scaffolder takes as input a set of assembled contigs and a set of paired reads. The relative orientation of two paired reads and the approximate distance between them are known. Thus, if the two reads can be unambiguously mapped to different contigs, we can identify the relative ordering and the distance between these contigs. Due to errors in the read pair data (e.g. chimeric pairs) and in the assembly (e.g. misassembled contigs) the ordering achieved from different read pairs can be contradictory. Consequently, the scaffolding problem is often defined as finding an ordering on the contigs that maximizes the number of supporting read pairs. Computationally, this formulation is NP-hard (Huson *et al.*, 2002), leading most scaffolding approaches to use heuristic algorithms with no provable guarantees.

Some scaffolders greedily link contigs by considering them in order of strongest paired read support (Pop *et al.*, 2004) or largest contig length (Boetzer *et al.*, 2011), while rejecting links that contradict those already chosen. Alternatively, the scaffolding problem is often represented as a graph, where nodes denote contigs and edges denote paired read links. SOPRA (Dayarian *et al.*, 2010) partitions this graph into smaller parts and solves the scaffolding problem in each subgraph using statistical optimization. MIP Scaffolder (Salmela *et al.*, 2011) partitions the graph in a similar way; however, it solves the problem for each subgraph exactly using Mixed Integer Programming. To keep the algorithms tractable, both of these scaffolders limit the sizes of the subgraphs. Opera Gao *et al.* (2011) applies an alternate partitioning scheme using a graph contraction procedure and solves the scaffolding problem with a fixed-parameter tractable algorithm based on a graph-bandwidth formulation. These approaches to scaffolding attempt to maximize the number of paired reads that are satisfied, implicitly assuming that paired read links are noisy, and the contigs are error-free. However, in larger and more complex genomes the assembled contigs may well have misassemblies.

In this paper we present a novel method that combines several practical algorithms for the scaffolding problem. Our approach assumes that both erroneous read pairs and contigs are possible, allowing us to detect misassembled contigs and remove these from the scaffolds. This formulation of the problem allows for an algorithm with practical time and memory requirements, while providing an exact solution of bounded error. This algorithm is implemented within SCARPA, a stand-alone scaffolder for HTS data. We have tested SCARPA on real datasets as well as on a simulated diploid genome, and show that it builds highly accurate and longer scaffolds compared to several state-of-the-art scaffolders.

*to whom correspondence should be addressed

2 METHODS

We implement our methods in a stand-alone scaffolder named SCARPA. As input, SCARPA takes a FASTA file containing a set of contigs and a SAM file containing the mapping positions of one or more paired read libraries. These files can be generated by any software of choice. As a preprocessing step, SCARPA filters ambiguously mapping reads and estimates the mean and standard deviation of the insert size for each library. SCARPA then assigns an orientation to each contig discarding a minimal set of contradictory contigs and paired read links. In the next step, contigs are given a pairwise-consistent order and finally the exact order of the contigs is determined using a Linear Programming framework. We explain these steps in detail below.

2.1 Preprocessing

The first step of SCARPA is to filter and analyze the read mappings. Like other scaffolders, SCARPA discards a read pair if either of the reads maps ambiguously (i.e. has more than one optimal hit). Next, SCARPA analyzes the read mappings to estimate the mean and standard deviation of the insert size for each library. While for most sequencing projects initial estimates of these are available, inaccurate values will cause gap sizes between the contigs to be incorrectly estimated. SCARPA re-estimates the insert size distribution using paired reads that map to the same contig. To make sure this estimation is reliable, we only use contigs that are longer than the contig N50 (see section 3.2 for a definition of N50). If the calculated mean is more than half of the contig N50 we use the provided library statistics instead.

After the library statistics are finalized, we build a scaffolding graph where nodes are contigs and edges are paired read links between the contigs. If there are multiple links between a pair of contigs, we bundle them provided that they suggest the same relative orientation. Each edge is weighted by the number of paired reads supporting the link and edges with support lower than a threshold are discarded. By default this threshold is 2, however, it can be set during program execution. In addition, each edge has an associated estimate for the distance between the contigs it connects. This distance, denoted with α_{ij}, is computed using the formula below:

$$\alpha_{ij} = \frac{1}{n} \sum_{l=1}^{n} m_l \quad (1)$$

where m_l is the estimated distance between the contigs i and j based on the paired read link l and n is the number of paired read links between these two contigs. Here m_l is calculated by subtracting the distance between the mapped positions and the end of the contigs from the mean insert size. Note that it is possible for this value to be negative since the end of the contigs may overlap.

2.2 Contig orientation as Odd Cycle Transversal

Each assembled contig is arbitrarily oriented; it could be mapped to either strand of the genome. The orientation stage of scaffolding attempts to orient the contigs based on the read pairs so that within each scaffold all of the contigs lie on the same strand. This is illustrated in Fig. 1. With error-free data this problem has a feasible solution, easily identified via a greedy algorithm. In the presence of errors, such as chimeric read pairs, mismapped reads and incorrectly assembled contigs, the problem may be infeasible unless we remove some constraints.

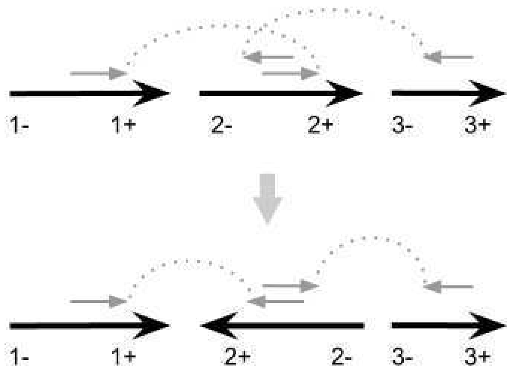

Fig. 1. *Contig orientation.* The relative orientation of contigs with respect to each other is identified via paired read links. Here, we assume the correct orientation of a read pair is forward-reverse (i.e. paired-end orientation). If the orientation of the library is otherwise, reads are reverse complemented to match this orientation prior to scaffolding. Above, we reverse complement the middle contig in order to satisfy the orientation of the paired reads.

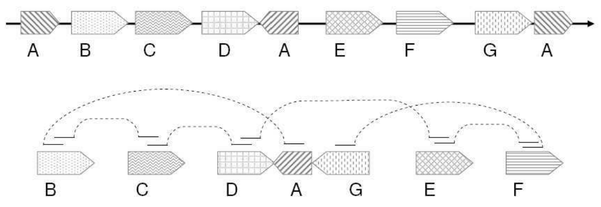

Fig. 2. *An example of a misassembled contig.* **Top:** A genome divided into several regions depicted with letters A-G. The repeat region A has three copies in the genome, one of which is inverted. **Bottom:** Five assembled contigs and a set of paired reads mapped to these contigs. The third contig is misassembled due to overcollapsing of the repeat region A. Here, we have to discard two paired read links so that the contig orientation problem has a feasible solution. Moreover, removing the wrong links will cause further errors in scaffolds.

The orientation problem is usually formulated as follows: assign an orientation for each contig so that the maximum number of paired reads is satisfied. This formulation, adapted by most scaffolders (Dayarian *et al.*, 2010; Pop *et al.*, 2004; Salmela *et al.*, 2011), is motivated by the assumption that the majority of incorrect links are due to chimeric pairs or mismapped reads. However, contradictory links may also be due to misassembled contigs (see Fig. 2). In such cases, it might be desirable to remove contigs instead of links. Furthermore, links that are due to chimeric pairs tend to have low support - such errors are expected to occur independently - and can often be identified during preprocessing.

Instead, we adopt an approach that allows removal of contigs as well as paired read links. We will first illustrate how to optimize the number of contigs that are removed, and then generalize the approach to paired read links. First, we build an undirected graph

G, where each contig c is represented by two nodes c^- and c^+ corresponding to the 5' and 3' ends of the contig respectively. For each contig c, we add an edge between c^- and c^+. For each read pair r_1 and r_2 mapping to contigs x and y respectively, we add an edge between[1]:

- x^+ and y^- if r_1 maps to x on the forward strand and r_2 maps to y on the reverse strand
- x^- and y^+ if r_1 maps to x on the reverse strand and r_2 maps to y on the forward strand
- x^+ and y^+ if both r_1 and r_2 map on the forward strands
- x^- and y^- if both r_1 and r_2 map on the reverse strands

Note that the contig orientation problem has a feasible solution if and only if G has no cycles containing an odd number of nodes. We thus attempt to find the smallest set of nodes which can be removed from the graph to allow for a feasible solution. In graph theory this is known as finding a minimum odd cycle transversal, and while this problem is NP-hard in general, it can be solved efficiently if the number of nodes to be removed is small. Reed *et al.* (2004) developed a fixed-parameter tractable algorithm which identifies a set X of nodes with $|X| \leq k$, for any fixed k, such that $G - X$ has no odd cycles or asserts that no such set exists. This algorithm runs in time $O(3^k kmn)$ (Lokshtanov *et al.*, 2009). The value of k, i.e. the maximum number of nodes to remove, is first set to 0 and then iteratively increased until a feasible solution is found.

2.2.1 Removing paired read links in addition to contigs The algorithm we describe above is based on removal of nodes (contigs). To allow removal of edges (paired read links) in addition to nodes, we build another graph G', which is derived from G by inserting auxiliary nodes. Briefly, we insert two nodes for each edge that connects two contig nodes (but not for an edge that connects the two ends of the same contig). Figure 3 illustrates this process. It is easy to see that this transformation does not alter the parity of existing cycles or create new cycles.

If there is a tie between discarding a contig node versus discarding an auxiliary node representing a paired read link, we would like the algorithm to remove the auxiliary node. In order to encourage the algorithm to remove paired read links before removing contigs, we order the nodes of G' such that the auxiliary nodes are considered before any contig node. Note that the algorithm will never choose to discard both auxiliary nodes representing the same paired read link, since this would contradict the optimality of the algorithm.

2.2.2 Assigning the orientation of the contigs Once the graph G is free of odd cycles, we transform it into a directed graph T, while simultaneously assigning each contig an orientation. To perform this task, we start with an arbitrary contig x. Without loss of generality, we assign the orientation of this contig as "forward". This assignment is reflected in the graph T by setting the direction of the edge (x^-, x^+) as $x^- \rightarrow x^+$. This also means that all other edges incident to x^+ must be outgoing edges. Similarly, all other edges incident to x^- must be incoming edges. This information is propagated to the rest of the graph via a breadth-first-search. For example, if there is an edge (x^-, y^-), the direction of this edge is set to $y^- \rightarrow x^-$. In turn, the direction of the edge (y^-, y^+) is set to $y^+ \rightarrow y^-$ and therefore the orientation of contig y is assigned as "reverse". This process is illustrated in Fig. 4.

[1] In practice, we bundle the paired reads that suggest the same orientation and represent them as a single edge.

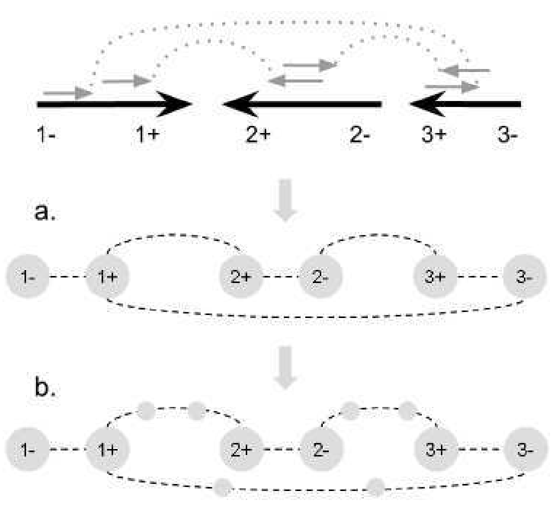

Fig. 3. *Formulation of the contig orientation problem as an odd cycle transversal problem.* **a.** We create two nodes for each contig corresponding to the two ends of the contig and connect these nodes with an edge. Then the paired read links are used to connect the ends of the contigs. Conflicting links create odd length cycles in the resulting graph. **b.** To allow removal of paired read links in addition to contigs, we modify the graph by creating two auxiliary nodes on each edge induced by these links. This modification preserves the parity of the cycles of the original graph.

2.2.3 A note on the relation of bidirected graphs and the odd cycle transversal problem The scaffolding problem is sometimes represented as a bidirected graph (Salmela *et al.*, 2011). It is easy to see that the initial undirected graph G we construct above is equivalent to a bidirected graph. For interested readers, we hereby note that this section also provides a general algorithm to convert an arbitrary bidirected graph into a directed graph by removing a minimal set of nodes and edges.

2.3 Ordering

Although the orientation step removes all odd cycles from G, the directed graph T may still have cycles (see Fig. 4). To place the contigs into a linear order, we need to eliminate all directed cycles from T. The problem of finding a minimal set of edges, whose removal makes a directed graph acyclic is known as the feedback arc set problem. For arbitrary graphs, this problem is NP-hard (Karp, 1972). We use a heuristic algorithm which runs in $O(m)$ time where m is the number of edges and guarantees an asymptotically optimal error bound for sparse graphs (Eades *et al.*, 1993).

2.4 Spacing

During the ordering stage, T is transformed into a directed acyclic graph and is now guaranteed to have an ordering of the contigs so that the remaining links are satisfied. In other words, each connected

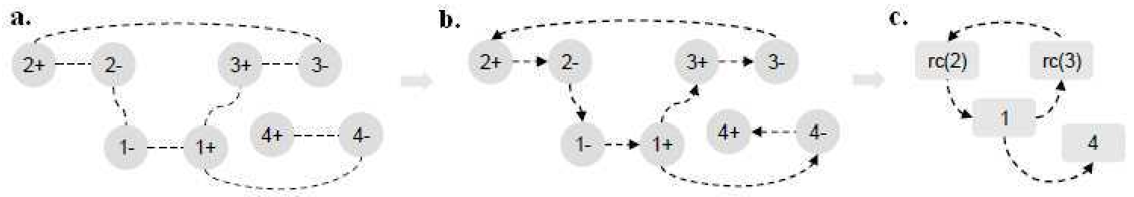

Fig. 4. *Assigning orientations.* **a.** An undirected scaffolding graph G after odd cycles are removed. **b.** The edges are given directions in a greedy manner starting from an arbitrary contig (in this example, from contig 1). **c.** Once the directions of the edges are assigned, we merge the nodes corresponding to the ends of the same contig into a single node. The contigs labeled with **rc(.)** are reverse complemented.

component of T admits a topological ordering. Yet, this ordering may not be unique. In the last stage of scaffolding, we try to find a placement of contigs within each scaffold such that the distances between the contigs agrees best with the size of the gaps as suggested by the paired read links. This task can be formulated as a Linear Programming (LP) problem as follows. For each contig $1 \leq i \leq N$, where N is the number of contigs, we have a real valued free variable x_i that represents the 5' end of the contig. Without loss of generality, we set x_1 to 0. For each paired read link, we introduce the following constraints:

$$x_i - x_j + d_{ij} \leq C(1 - \delta_{ij}) \tag{2}$$

$$x_j - x_i - d_{ij} \leq C(1 - \delta_{ij}) \tag{3}$$

where d_{ij} is the distance between the 5' ends of the contigs i and j suggested by the paired read link. δ_{ij} is a real valued slack variable in the range $[0, 1]$. C is a large constant set to the sum of all the contig lengths. Subject to the set of constraints as constructed above, we maximize $\sum \delta_{ij}$.

Note that the LP formulation is designed to place the contigs so that the paired read links are satisfied best, however, it may allow two contigs to occupy the same coordinates. In practice, we do not use the coordinates returned by the LP solver; rather, we use these coordinates to order the contigs in linear paths as follows. If a contig i is followed by a contig j according to the coordinates returned by the solver and i and j are already connected by an edge in T we keep this edge. If the two nodes are not connected by an edge, we compute the shortest path between i and j in T. If the length of this path is less than a small threshold, we create an edge between i and j. The length of the gap between these contigs is computed using the coordinates returned by the solver. If the shortest path between i and j is longer than the threshold, we infer that these contigs are not supposed to be adjacent. In this case, the contigs following j are considered in order until one of them passes these criteria. If such a contig is found, then it follows i in the path and a new path is created for j. The resulting linear paths are output as scaffolds. If the estimated length of the gap between two adjacent contigs is negative, we align the ends of these contigs to see whether an overlap is present. If a high identity overlap is present, we merge these contigs. Otherwise, a fixed gap length of 10bp is assigned.

2.5 Components

Trivially, all the steps we describe above can be solved separately for each connected component of the relevant graph. In order to keep the running time of SCARPA within practical limits (for orientation) and to improve accuracy (for ordering and spacing), we further divide the graph into biconnected components. The biconnected components of an arbitrary graph can be computed in linear time using the classical algorithm by Hopcroft and Tarjan (1973). This algorithm works by finding a set of nodes, called the *articulation points*, whose removal from the graph increases the number of connected components. For the orientation and ordering steps, we have to ensure that each biconnected component can be solved independently without violating the correctness of the algorithms. To accomplish this, we only use those articulation points with in and out degrees equal to 1. Such nodes can never form cycles, hence their removal does not violate the correctness of the odd cycle transversal and the feedback arc set algorithms.

Note that highly connected graphs may not admit any articulation points. These graphs often contain several repeat contigs that act as hubs. To avoid this scenario, we limit the maximum number of links a contig can make. If a contig exceeds the threshold, it is disconnected from the graph. This threshold is adjusted automatically depending on the component sizes.

2.6 Multiple libraries

In the presence of two or more libraries, SCARPA starts with the library of the smallest insert size. Remaining libraries are processed in order of increasing insert size, where scaffolds from the previous stages are treated as contigs.

3 RESULTS

3.1 Datasets

In our first set of experiments, we compare SCARPA to other state-of-the-art scaffolders on two real Illumina datasets sampled from the bacterium *E. coli* (strain K-12 substrain MG1655) and the fungus *G. clavigera*. For *E. coli*, we evaluate the scaffolders using the high quality finished sequence available from NCBI (accession code: NC_000913.2). A finished reference sequence for *G. clavigera* is not available, so we evaluate the scaffolders on the draft sequence assembled using Sanger, Roche/454 and Illumina data as described by DiGuistini *et al.* (2009). For both genomes, we use two Illumina paired-end libraries downloaded from the NCBI Short Read Archive.

In order to estimate the performance of SCARPA on a larger dataset, we also test the scaffolders using a simulated paired-end library taken from the first Assemblathon experiment (Dent *et al*,

Table 1. Datasets used for evaluation. For *G. clavigera*, the size of the available draft sequence is given in place of the genome size. For the Assemblathon1 dataset, the genome size is given as an average of the haploid reference sequences.

Organism	Genome size (mbp)	Accession code	No. of reads	Sequence Coverage	Spanning Coverage	Read length (bp)	Insert size (bp)
Escherichia coli	4.6	SRR001665	2x10.4m	162x	452x	36	200
		SRR001666	2x7.0m	109x	760x	36	500
Grosmannia clavigera	29.7	SRR018008	2x10.2m	28x	68x	42	200
		SRR018012	2x12.0m	40x	282x	50	700
Assemblathon1	112.5	-	2x22.5m	40x	60x	100	300

Table 2. Scaffolding statistics for the *E. coli* dataset. The last column denotes the number of multi-contig scaffolds that contain breakpoints (i.e. scaffolds that do not admit a co-linear mapping to the reference). The numbers in parantheses denote the total number of multi-contig scaffolds.

	No. of scaffolds	Coverage (%)	Largest (kbp)	N50 (kbp)	Total (mbp)	No. of Breaks
Contigs	489	97.89	103	24	4.54	-
Velvet	138	98.37	312	132	4.56	-
SCARPA	156	98.06	268	136	4.55	5 (56)
SSPACE	81	98.36	275	132	4.55	9 (57)
MIP	85	98.47	226	95	4.55	10 (63)
SOPRA	294	97.98	188	50	4.55	1 (73)

Table 3. Scaffolding statistics for the *G. clavigera* dataset.

	No. of scaffolds	Coverage (%)	Largest (kbp)	N50 (kbp)	Total (mbp)
Contigs	5298	89.09	109	18	26.59
Velvet	2084	89.65	1068	164	27.03
SCARPA	1691	89.24	863	234	26.74
SSPACE	1570	89.20	817	169	26.84
MIP	1839	89.21	367	53	26.75
SOPRA	2305	89.16	855	194	26.70

Table 4. Scaffolding statistics for the Assemblathon1 dataset.

	No. of scaffolds	Coverage (%)	Largest (kbp)	N50 (kbp)	Total (mbp)
Contigs	13364	99.19	85	18	114.53
SCARPA	5620	99.30	324	54	114.14
SSPACE	7936	99.25	144	34	114.55
MIP	5193	99.28	325	58	114.01
SOPRA	6258	99.28	280	48	114.55

2011). This library consists of 100bp long reads with 300bp insert size sampled from an artificially evolved diploid genome. Reads are simulated with sequencing errors and correspond to 40x coverage. The characteristics of the diploid reference and the simulation process are described in detail by Dent *et al* (2011). The statistics of all datasets are given in Table 1.

For the *E. coli* and *G. clavigera* datasets, we assemble the reads into contigs using Velvet (Zerbino *et al.*, 2009) and report the contigs with the kmer size that achieves the highest N50 value (29 and 27 respectively). For these datasets, we set the expected coverage and coverage cutoff to automatic, and only report contigs that are 100bp or longer. For the Assemblathon1 dataset, we assemble the reads using Hapsembler (Donmez and Brudno, 2011), which has support for diploid datasets. Since the reads are longer, we set the minimum contig size to 200bp. For this dataset, we also discard read pairs that map to the *E. coli* genome prior to assembly and scaffolding in order to remove contamination. The total number of pairs removed by this process is 864,758 corresponding to $\sim$1.5x reduction in coverage.

3.2 Evaluation

We compare SCARPA to three other scaffolders; SSPACE (Boetzer *et al.*, 2011), MIP Scaffolder (Salmela *et al.*, 2011) and SOPRA (Dayarian *et al.*, 2010). We also report the scaffolds produced by Velvet on the *E. coli* and *G. clavigera* datasets. The standard deviation for each library is set to 10% of the mean insert size. For MIP Scaffolder, the minimum and maximum insert size values are set to 3 standard deviations below and above the mean respectively. The other parameters are left at default values. We let SCARPA adjust all parameters automatically for each dataset. For all scaffolders, the reads are mapped with Bowtie using the same options (Langmead

Table 5. Number of inversion type errors computed using the same sampling of sequence pairs as in accuracy.

	SCARPA	SSPACE	MIP	SOPRA	Velvet
E.coli	0	6	5	0	1
G.clavigera	3	4	28	1	3
Assemblathon1	38	27	53	43	-

Table 6. Wall-clock running times of the scaffolders in minutes. Mapping is performed using Bowtie with 16 threads and mapping time is included for all scaffolders. For information, the total wall-clock time taken by Bowtie to index the reference and write read mappings in SAM format is also reported.

	Bowtie	SCARPA	SSPACE	MIP	SOPRA
E.coli	3	12	3	29	97
G.clavigera	9	15	5	51	172
Assemblathon1	18	27	15	90	515

et al., 2009). For all evaluations reported in this section, the mapping of scaffolds is performed using the **nucmer** and **delta-filer** utilities of the MUMmer package (Version 3.22) (Kurtz *et al.*, 2004)

The scaffold length statistics are summarized in Tables 2, 3 and 4. N50 is calculated as the largest scaffold length such that the sum of scaffolds at least as long is greater than half the total scaffold size. The coverage is measured by mapping the scaffolds to the reference sequence and includes gaps. Statistics regarding the number of contigs merged in scaffolds are included in the online supplementary materials.

For the *E. coli* dataset, we additonally report the number of multi-contig scaffolds that contain breakpoints in the mappings. Note that we can not compute these numbers reliably in the *G. clavigera* and Assemblathon1 datasets. In the former case, a finished reference sequence is not available: The draft assembly we use for evaluation consists of 289 scaffolds. In the latter, the presence of two haplotypes implies that there may be haplotype switches within the contigs as well as the scaffolds, making it difficult to estimate the real number of breakpoints.

To estimate the accuracy of the scaffolds in the other datasets, we employ a method similar to the one used by Salmela *et al.* (2011). Briefly, this method works by extracting pairs of sequences separated by a certain distance from the scaffolds. These pairs are then mapped to the reference and the proportion of pairs that map with the correct orientation and within 10% of the correct distance is reported. In our experiments, we use a tiling of 1000bp long sequences separated by a distance of 3000bp. Figure 5 shows the accuracy versus the N50 measure for each dataset. We also report the number of inversion type errors in Table 5. An inversion error is said to occur when one of the tiling pairs map to the forward strand of the reference while the other maps to the reverse strand.

SCARPA produces highly accurate scaffolds that are at least as long or longer than the other tools. We also find that for the *E. coli* dataset, out of the three contigs with lengths 282bp, 426bp and 428bp removed by SCARPA during scaffolding, two of them (with lengths 282bp and 428bp) do not map to the reference sequence. For the other datasets, SCARPA only removed paired read links.

The running times of the scaffolders on a server with 20 cores operating at 2.67GHz and 80GB memory are given in Table 6. On these datasets, SSPACE is the fastest, followed by SCARPA and MIP Scaffolder, while SOPRA is the slowest. Note that SSPACE actually takes less time than Bowtie, probably due to the fact that it runs Bowtie internally avoiding the extra time needed to process the read mappings.

4 CONCLUSION

Scaffolding improves the contiguity of an assembly and facilitates the finishing of a genome by establishing an order and orientation of contigs. In this paper, we have presented SCARPA, a novel scaffolder for HTS data that combines graph algorithms with Linear Programming. Using simulated and real datasets, we show that SCARPA produces as long or longer scaffolds than the current state-of-the-art tools, while at the same time achieving high accuracy.

A novel feature of SCARPA is the ability to detect misassembled contigs. Although this procedure may produce false positives, SCARPA reports only a few such contigs per dataset, which can be manually investigated if necessary. For instance, SCARPA discards no contigs in the *G. clavigera* and Assemblathon1 datasets and only three contigs in the *E. coli* dataset, two of which are indeed found to be erroneous.

We also show that SCARPA has favorable running time on these datasets, although it is slightly slower than SSPACE. In addition, SCARPA has a small memory footprint, requiring less than 2GB on the Assemblathon1 dataset.

Within SCARPA, the most time consuming step is the contig orientation task. While we believe our method typically produces more accurate scaffolds compared to greedy or heuristic based approaches and has the advantage of detecting misassemblies, it can be computationally expensive for large and complex genomes. On the other hand, the fixed-parameter tractable algorithm we employ is suitable for parallel computation. Although our current implementation is single-threaded, we plan to explore this direction in a future version.

ACKNOWLEDGEMENT

We would like to thank members of the Computational Biology Lab at the University of Toronto for assistance with this manuscript.

Funding: This work was supported by an NSERC Discovery Grant to MB.

REFERENCES

Boetzer, M., Henkel, C., Jansen, H., Butler, D., and Pirovano, W. (2011). Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics*, **27**(4), 578–579.

Dayarian, A., Michael, T., and Sengupta, A. M. (2010). SOPRA: Scaffolding algorithm for paired reads via statistical optimization. *BMC Bioinformatics*, **11**(345).

Dent *et al* (2011). Assemblathon 1: A competitive assessment of *de novo* short read assembly methods. *Genome Research*, **21**(12), 2224–2241.

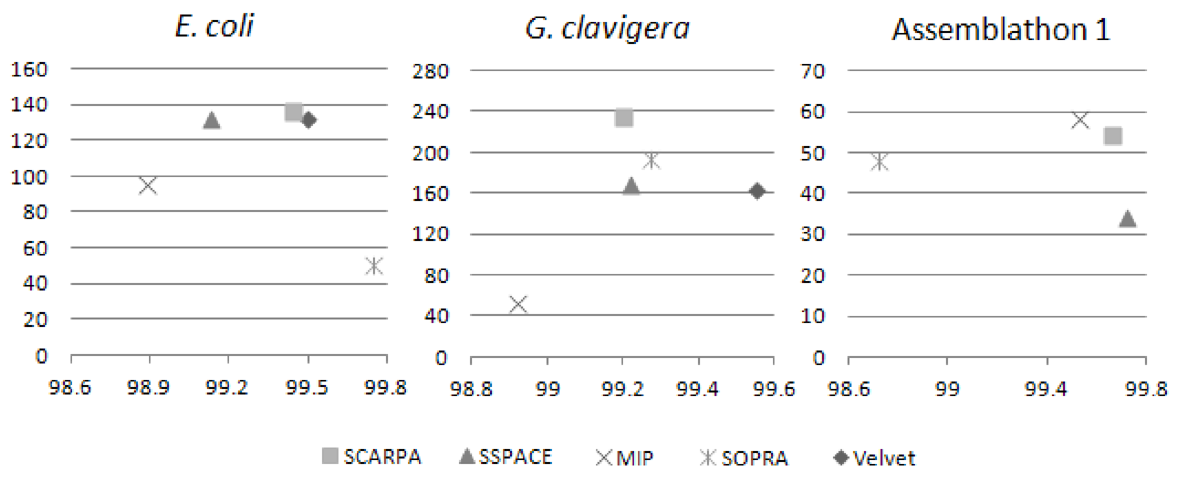

Fig. 5. Scaffold N50 (kbp) versus accuracy (%) for each dataset. For the Assemblathon1 dataset, we map the pairs to both haplomes (i.e. haploid copy of the genome) and consider a pair correct if it maps to either haplome with the correct orientation and within 10% of the expected distance. We perform all mappings using MUMmer (version 3.22) (Kurtz *et al.*, 2004).

DiGuistini, S., Liao, N., Platt, D., Robertson, G., Seidel, M., Chan, S., Docking, T. R., Birol, I., Holt, R., Hirst, M., Mardis, E., Marra, M., Hamelin, R., Bohlmann, J., Breuil, C., and Jones, S. (2009). *De novo* genome sequence assembly of a filamentous fungus using sanger, 454 and illumina sequence data. *Genome Biology*, **10**(9).

Donmez, N. and Brudno, M. (2011). Hapsembler: an assembler for highly polymorphic genomes. In V. Bafna and S. Sahinalp, editors, *Research in Computational Molecular Biology*, volume 6577 of *Lecture Notes in Computer Science*, pages 38–52. Springer Berlin / Heidelberg.

Eades, P., Lin, X., and Smyth, W. F. (1993). A fast effective heuristic for the feedback arc set problem. *Information Processing Letters*, **47**, 319–323.

Gao, S., Nagarajan, N., and Sung, W.-K. (2011). Opera: Reconstructing optimal genomic scaffolds with high-throughput paired-end sequences. In V. Bafna and S. Sahinalp, editors, *Research in Computational Molecular Biology*, volume 6577 of *Lecture Notes in Computer Science*, pages 437–451. Springer Berlin / Heidelberg.

Hopcroft, J. and Tarjan, R. (1973). Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, **16**(6), 372–378.

Huson, D. H., Reinert, K., and Myers, E. W. (2002). The greedy path merging algorithm for contig scaffolding. *Journal of the ACM*, **49**(5), 6003–615.

Karp, R. M. (1972). Reducibility among combinatorial problems. *Complexity of computer computations*, pages 85–103.

Kurtz, S., Phillippy, A., Delcher, A., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. (2004). Versatile and open software for comparing large genomes. *Genome Biology*, **5**(2), R12.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, **10**(3).

Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J., and Wang, J. (2009). *De novo* assembly of human genomes with massively parallel short read sequencing. *Gemome Research*, **20**(2), 265–272.

Lokshtanov, D., Saurabh, S., and Sikdar, S. (2009). Simpler parameterized algorithm for OCT. In J. Fiala, J. Kratochvíl, and M. Miller, editors, *Combinatorial Algorithms*, pages 380–384. Springer Berlin / Heidelberg.

Pop, M., Kosack, D., and Salzberg, S. (2004). Hierarchical scaffolding with Bambus. *Genome Research*, **14**, 149–159.

Reed, B., Smith, K., and Vetta, A. (2004). Finding odd cycle transversals. *Operations Research Letters*, **32**, 299–301.

Salmela, L., Makinen, V., Valimaki, N., Ylinen, J., and Ukkonen, E. (2011). Fast scaffolding with small independent mixed integer programs. *Bioinformatics*, **27**(23), 3259–3265.

Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J., and Birol, I. (2009). ABySS: A parallel assembler for short read sequence data. *Genome Research*, **19**(6), 1117–1123.

Zerbino, D. R., McEwen, G. K., Margulies, E. H., and Birney, E. (2009). Pebble and Rock Band: Heuristic resolution of repeats and scaffolding in the Velvet short-read *de novo* assembler. *PLoS ONE*, **4**(12).