UNIVERSITY OF CALIFORNIA
RIVERSIDE

Algorithms and Data Structures for *de novo* Sequence Assembly

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Hind A. I. AL Hakami

June 2017

Dissertation Committee:

Professor Stefano Lonardi, Chairperson
Professor Giafranco Ciardo
Professor Marek Chrobak
Professor Timothy Close

The Dissertation of Hind A. I. AL Hakami is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am so grateful to have had Dr. Stefano Lonardi and Dr. Giafranco Ciardo as my advisors. Without their kind guidance I would not have made it here.

Many thanks to Dr. Giafranco Ciardo, for offering guidance that helped me develop many skills ranging from constructing sound proofs to using LaTeX. To him I owe everything I know about software verification.

I joined Dr. Lonardi's lab with virtually no knowledge in Bioinformatics. He encouraged me and supported me to take classes and build my background in Computational Biology. I deeply appreciate the freedom he gave me in exploring a wide spectrum of topics, and working on research topics that best suited my interests. To him I owe everything I know in Bioinformatics. It has been a pleasure working with and under the guidance of such a kind and supportive advisor, who constantly offered valuable advice in many aspects from research to career development.

I also would like to thank Dr. Timothy Close and Dr. Maria Munoz-Amatriain from (Department of Botany and Plant Sciences), for the valuable discussions in our weekly meetings. I have learned a lot while working with them. I also value the hands-on experience I gained while accompanying them in the field; and for that I am very grateful.

I am also thankful to Dr. Marek Chrobak for providing help with some proofs in this thesis. I also appreciate the time and effort he and Dr. Timothy invested as committee members.

Last but not least, I am grateful to all UCR professors who contributed to my knowledge; especially, Dr. Tao Jiang, Dr. Thomas Girke, Dr. Rajeev Gupta, and Dr.

Harsha Madhyastha (now at University of Michigan).

Portions of Chapter 2 appeared in *SPIRE*, in the paper titled ”Sequence Decision Diagrams,” co-authored with Giafranco Ciardo and Marek Chrobak [2].

Materials appearing in Chapter 3 are the result of collaboration with Hamid Mirebrahim and Stefano Lonardi. Materials in Chapter 4 are the results of collaboration with Stefano Lonardi.

To my loving parents and brothers.

ABSTRACT OF THE DISSERTATION

Algorithms and Data Structures for *de novo* Sequence Assembly

by

Hind A. I. AL Hakami

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, June 2017
Professor Stefano Lonardi, Chairperson

Despite the prodigious throughput of the sequencing instruments currently on the market, the assembly problem remains computationally very challenging, mainly due to the repetitive content of large genomes, uneven sequencing coverage, and the presence of (non-uniform) sequencing errors and chimeric reads. As a consequence, the final assembly is very rarely entirely finished, with one solid sequence per chromosome.

In this dissertation, we study (1) the problem of merging multiple genome-wide assemblies produced using different assemblers and/or parameters, and (2) the problem of stitching multiple overlapping local assemblies (e.g., assemblies generated by sequencing BAC clones) to create a genome-wide assembly. Both assembly problem involves processing very large set of strings, which in turns requires memory-efficient data structures that allow for efficient comparison operations. In this context, we propose a data structure for the compact encoding of finite sets of strings over a finite alphabet called *sequence decision diagrams* (SeqDDs), which allows for efficient set operations. Next, we study and benchmark several published methods to merge multiple genome-wide assemblies with the objective to produce a higher quality consensus assembly. Our comprehensive comparative study of assembly reconciliation tools is the first of its kind. Finally, we develop, implement and test novel algorithms to stitch locally overlapping assemblies based on the *colored-positioned de Bruijn graph*, a variant of the classic de Bruijn graph.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The sequencing instruments currently on the market have enabled the sequencing of many large, complex genomes. Despite the tremendous throughput of these instruments, the assembly problem is still very challenging, mainly due to the repetitive content of large genomes, uneven sequencing coverage, and the presence of (non-uniform) sequencing errors and chimeric reads. The third generation of sequencing technology, e.g., Pacific Biosciences [27] and Oxford Nanopore [19], offers very long at a higher cost per base, but sequencing error rate is much higher (summary in Table 1.1). As a consequence, long reads are more commonly used for scaffolding contigs created from second generation data, rather than for *de novo* assembly [28].

A significant number of *de novo* genome assemblers are available to the community. The choice of the most appropriate assembler depends on the size and complexity (repeat content, ploidy, etc.) of the genome to be assembled, the type of sequencing technology used to produce the input reads (e.g., Sanger, 454, Illumina, PacBIO, Nanopore, etc.), and the

Platform	Sequencer	Maximal read length	Error rate	Average run duration	Cost per 1 Million bases (US dollars)
Sanger	ABI 3730xl	1000 bp	0.01%	2–3 hours	$2400
454	GS FLX	1000 bp	0.01%	24 hours	$10
Illumina	HiSeq 3000	250 bp	0.01%	4 days	
Illumina	NextSeq500	150 bp	0.01%	30 hours	$0.05 - $0.15
Illumina	MiSeq	300 bp	0.01%	24 hours	
Ion Torrent	PGM 318	400 bp	2%	7 hours	$1
PacBio	RS II	54 kbp	13%	3 hours	$0.13–$0.60
Nanopore	MinION	150 kbp	3% - 8%	n.a.	n.a.

Table 1.1: Summary of sequencing technology platforms

availability of paired-end or long-insert mate-pair reads. Each assembler implements slightly different heuristics to deal with repetitions in the genome, uneven coverage, sequencing errors and chimeric reads. The final assembly is very rarely entirely finished, with one solid sequence per chromosome. Instead, the typical output is an unordered/unoriented set of contiguous regions called *contigs*. If paired-end/mate-pair reads are available, contigs can be ordered and oriented by anchoring paired-end reads to contigs. The length of the gaps between contigs are estimated, then contigs are then joined into *scaffolds*.

BAC-by-BAC vs. whole genome shotgun sequencing

BAC-by-BAC sequencing starts by constructing a physical map of overlapping series of contigs each of which spans a large (150 Kbp on average) contiguous region of the source genome. Each contig is inserted into a host vector as a medium for replication. The host vector is a bacterium, hence the naming *bacterial artificial chromosome* (BAC).

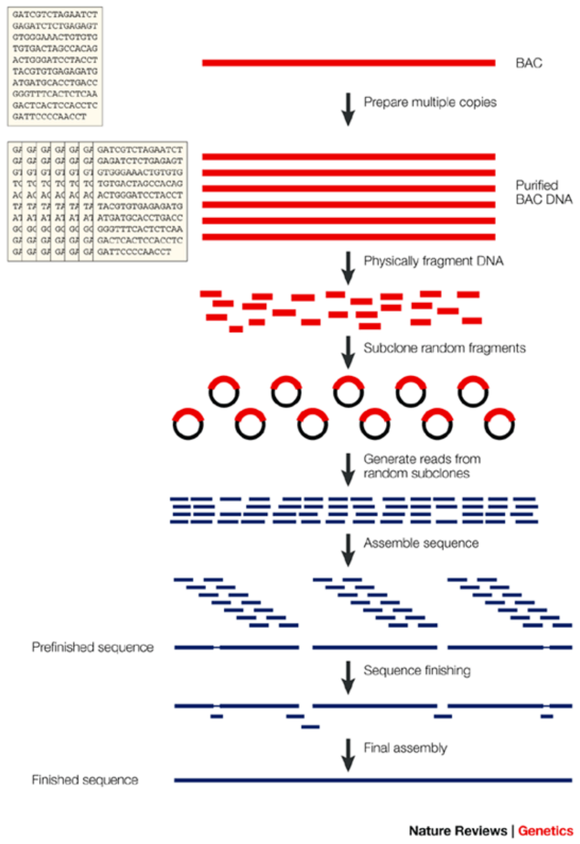


Figure 1.1: BAC cloning involves making copies of specific regions of the genome. Clones are then fragmented and random DNA fragments (typically 2-5 kb in size) are sub-cloned. Sequence reads are then generated from one or both ends of randomly selected sub-clones. Reads are then assembled for each BAC individually. Figure reproduced from [30]

Cloned BACs are then fingerprinted, using restriction enzyme to find common markers and order overlapping contigs. Next, a minimum tiling path is computed to select a minimal number of BACs spanning the genome. Selected BACs are then sub-cloned into smaller-insert libraries, from which sequence reads are randomly derived. Figure 1.1 illustrate this process.

Whole genome shotgun sequencing skip the mapping, fingerprinting, and the selection of a minimum tiling path phases and proceeds using sub-clone libraries prepared from the entire genome. Figure 1.2 show a comparison between BAC-by-BAC and whole

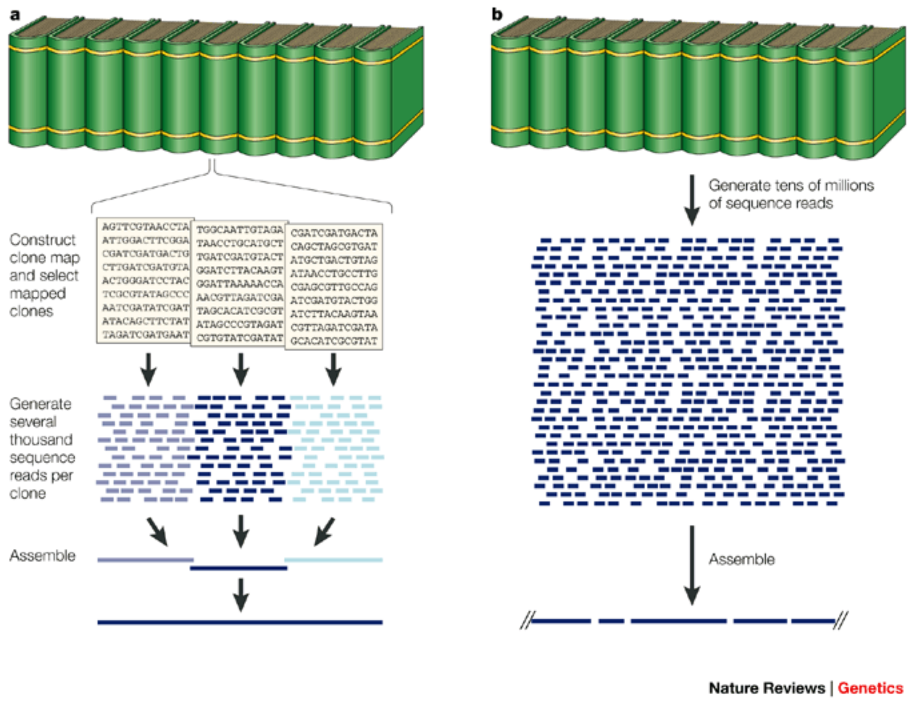

a
b
Construct clone map and select mapped clones
Generate several thousand sequence reads per clone
Assemble
Generate tens of millions of sequence reads
Assemble
Nature Reviews | Genetics

Figure 1.2: In this figure we represent a genome as a large encyclopedia. In (a) BAC-by-BAC sequencing, each page represents a BAC, each BAC is then sub-cloned and reads are generated. In (b) whole genome shotgun sequencing, the entire genome is fragmented and reads are generated from each DNA fragment. Figure reproduced from [30].

genome shotgun sequencing process.

Whole genome sequencing produces a base-by-base resolution, therefore allows for a comprehensive analysis of a genome such as capturing small variants as well as large variants. However, BAC-by-BAC sequencing approach is preferred when dealing with large genome, complex repeated regions, or when the goal is analyzing targeted regions (*selective sequencing*).

De novo sequence assembly

De novo sequence assembly is the reconstruction of a genome sequence from a large set of strings called *reads* without the help of a reference genome. The strategies used by *de novo* sequence assemblers can be classified into three groups

Greedy methods always makes the choice with the greatest immediate benefit; greedy assembler always joins the reads that overlap best, as long as they do not contradict the already constructed assembly. The choices made by the assembler are inherently local and do not take into account the global relationship between the reads. Most greedy assemblers use heuristics designed to avoid misassembling repetitive sequences. Assemblies produced by greedy paradigms are usually not of very quality because they do not take advantage of global information to resolve repetitive regions of the genome. Some examples of greedy assemblers are Phrap [31], SSAKE [77], and VCAKE [42].

Overlap-layout-consensus assemblers starts by identifying all pairs of reads that overlap sufficiently well; overlaps are represented into a graph (called *overlap graph*0, where node represent reads and edges represents an overlaps. Several complex algorithms that take into account the global relationship between the reads have been developed on the overlap graph. This strategy was introduced by Celera [60], a very influential assembler for Sanger sequencing reads. Other overlap-layout-consensus assemblers include, Celera Assembler with the Best Overlap Graph (CABOG) [57], Newbler [54], and Edena [39]. The high throughput of second-generation instrument poses high computational demands on the overlap-layout-consensus paradigm.

String graph. A variant of the OLC approach that simplifies the global overlap graph by removing redundant information (transitive edges) introduced by SGA assembler [69] based on FM-index, an efficient string indexing data structure.

de Bruijn graph represents input reads as a sequence of their subwords of length k (called *k-mers*). Nodes in the graph represent *kmers*, and the edges indicate an overlap by exactly $k-1$ nucleotides. Most de Bruijn graph assemblers use the read information to refine the graph structure and to remove graph patterns that are not consistent with the reads. De Bruijn graphs for genome assembly were first introduced in the EULER assembler [15]. Since then, they have the primary data structure for modern assemblers targeted at short-read sequencing data, e.g., Velvet [84], SOAPdenovo [51] and ALLPATHS-LG [29].

The rest of this Dissertation is organized as follows. In Chapter 2, we introduce *Sequence Decision Diagrams* (SeqDD), which are canonical decision diagrams that do not suffer from ordering problem. SeqDD is a data structure designed to compactly store finite sets of strings sharing substantial amount of common substrings. In that chapter, we present efficient algorithms to carry out set operations using the memoization property, an intrinsic feature of decision diagrams. In Chapter 3, we present a comparative analysis of assembly reconciliation tools. The objective of these tools is to merge multiple draft assemblies to obtain an assembly of higher quality. In Chapter 4 we introduce a novel method called *Sequence Overlap Identification and Assembly* (SequOIA). The objective of SequOIA is to merge overlapping local assemblies, like the ones generated by sequencing BAC clones belonging to a minimum tiling path of a genome.

Chapter 2

Representation and manipulation of large sets of finite sequences

The assembly problem requires memory-efficient data structures that store large sets of strings and allow for efficient set operations on them. In this chapter we introduce *sequence decision diagrams* (SeqDDs), which can encode arbitrary finite sets of strings over a finite alphabet. SeqDDs are a variant of classic decision diagrams such as BDDs and MDDs. Instead of having a fixed number of levels, SeqDDs require that the number of paths and the lengths of these paths to be finite. While MDDs are suited to store and manipulate large sets of constant-length tuples, SeqDDs can store arbitrary finite languages.

2.1 Background

2.1.1 Finite automata

A finite automaton consists of a finite number of states and labeled transitions such that the next state is determined by the current state, the input symbol, and the transition function. Finite automata can be categorized into *deterministic* finite automata (DFA) and *non-deterministic* finite automata (NFA).

A DFA is formally defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where,

- Q is a finite set of states
- Σ is a finite alphabet
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function
- $q_0 \in Q$ is a start state
- $F \subseteq Q$ is a set of accepting states

A NFA is defined similarly to a DFA; the 5-tuple $(Q, \Sigma, \delta, q_0, F)$ has the same definition except for the transition function which is defined as $\delta : Q \times \Sigma \cup \{\epsilon\} \rightarrow 2^Q$, such that, given a current state and a symbol, the transition function leads to a state chosen from a set of states, rather than a unique state. Moreover, ϵ-transitions in NFA allow advancement without reading an input symbol.

We also define a *partial* DFA, as in [10], to be a minimized DFA with partial transition function $\delta \subseteq Q \times \Sigma \rightarrow Q$ such that $\delta(q, a) = \emptyset$ for $q \in Q$ and $a \in \Sigma$ is allowed. In a *partial* DFA, the *trap* state and all transitions leading to it are omitted.

2.1.2 Decision diagrams

A decision diagram is a directed acyclic graph where each node encodes a function. Multi-valued decision diagrams (MDDs) are an extension of the better known binary decision diagram (BDD)[1]. BDDs provide a canonical representation of boolean functions, while MDDs provide a canonical representation of discrete functions. Both decision diagrams consist of

- *Non-terminal nodes:* each non-terminal node recursively encodes a composition of the sub-functions encoded by its children.
- *Terminal nodes:* there exist two terminal nodes, terminal **1** and terminal **0**. The first indicates that assignments of variables along the path from the root to terminal **1** satisfies the function encoded by the decision diagram, while terminal **0** denotes unsatisfiability.
- *Labeled directed edges* correspond to all possible assignments of a variable.

Canonicity is ensured through *ordering* and *reduction* rules. For a function with k variables, a global ordering $x_k \prec x_{k-1} \prec \cdots \prec x_1 \prec x_0$ of the variables should be preserved in all paths. Reduction rules are applied repeatedly on the fly to maintain a canonical minimized decision diagram at any stage of the construction.

- Node merging rule: no *duplicates* nodes are allowed; i.e., if two nodes are isomorphic,

[1]MDDs extend BDDs by allowing the outgoing edges from a node to describe choices that are not necessarily binary. We simply use to "MDDs" from now on, with the understanding that they include BDDs as a special case.

then the two nodes are merged. In an MDD implementation a *unique table* is used to enforce this rule.

- Node deletion rule: no *redundant* nodes are allowed; a node is considered redundant if all its children are identical. Such node is interpreted as a "don't care" node and is skipped.

A *quasi* reduction rule applies node merging without node deletion at any levels, while *full* reduction rule applies both node merging and node deletion (an example of BDDs after applying each reduction rule is shown in Figure 2.1). In addition to reduction rules, a sparse representation of a decision diagram is used. In sparse representation, terminal **0** is not represented, nor any of the edges leading to it.

Another variation of *ROBDD* is *Zero-Suppressed Binary Decision Diagrams (ZB-DDs)* [58], which is basically an ROBDD with a different deletion rule. In a *ZBDD*, a node is bypassed if the *one-child* leads to the **0**-terminal (refer to the example in Figure 2.1 (c)).

Decision diagrams are most efficient when encoding sets that share many subsets. In addition, the recursive structure of decision diagrams makes the use of dynamic programming cost effective. Decision diagram manipulation algorithms exploit this advantageous feature by using an *operation cache*, which eliminate the need to repetitively recompute sub-problems.

2.1.3 Related work

Many data structures have been introduced in the literature to compactly encode finite sets of finite strings. *Substring indices*, such as *tries*, suffix trees [56], suffix arrays

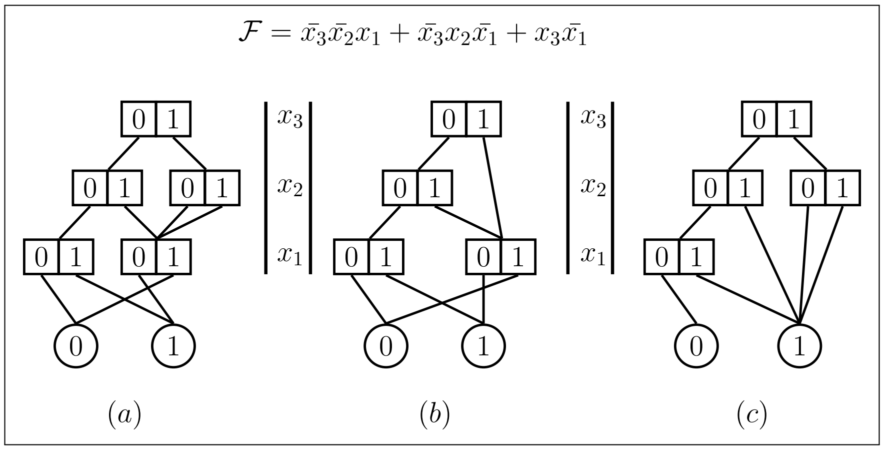

$\mathcal{F} = \bar{x_3}\bar{x_2}x_1 + \bar{x_3}x_2\bar{x_1} + x_3\bar{x_1}$
x_3
x_2
x_1
(a)
(b)
(c)

Figure 2.1: (a) Quasi-reduced BDD, (b) fully-reduced BDD, and (c) ZBDD representation for the same function.

[53] , and DAWGs [11], exploit prefix sharing, suffix sharing, or both to achieve efficient storage of large sets of strings. Beside compactness, the main purpose of substring indices is to efficiently solve the substring matching problem in a fixed text. Exact matching, in most cases, can be achieved in time complexity proportional to the pattern size, not the whole text.

While exact matching on these data structures is very efficient, updating the data structure by adding or deleting strings is hard [5]. Additionally, the lack of efficient set manipulation algorithms or such data structures stimulates the need for data structures that leverage the benefits of substring indices while enabling efficient set manipulation.

In 2009, Loekito *et al.* introduced a new data structure, *sequence* BDD [49], that combines compact storage of finite languages of arbitrary finite strings and, at the same time, provides for efficient set manipulation algorithms. *Sequence* BDD or SeqBDD, for short, is a half-relaxed variation of ZBDDs; variables along *zero-paths* are ordered, while the variables along *one-paths* have no order restrictions; moreover, variables can appear several times along such a path facilitates encoding languages composed of strings of different lengths.

SeqBDD inherits ZBDDs efficient set manipulation algorithms, in addition to other well known techniques of decision diagrams, such as the use of a unique table and an operation cache, to enable dynamic programming. Other algorithms have been introduced to mine frequent substring. In [5] the authors introduced a reversed SeqBDD to match suffixes and proposed SuffixDD, a SeqBDD that encode the set of all suffixes of a given string. In [26], SeqBDD that encode all substrings of a strings in a given language $\mathcal{L}$ is introduced, and named *factor* SDD. In fact, it has been proven in [25] that size of the *factor* SDD is linear in the size of the SeqBDD encoding $\mathcal{L}$.

Size complexity is a crucial issue in decision diagrams, and SeqBDDs are no exception. The importance stems from two factors; first, decision diagrams are usually used to store efficiently an enormous amount of data; second, the time complexity of algorithms applied to decision diagrams is proportional to the size of the arguments. As other reduced ordered decision diagrams, SeqBDDs are sensitive to variable ordering. Since optimal variable ordering is an NP-complete problem [12], heuristics are required to achieve good variable ordering. Sharing common suffixed as well as common prefixes contributes to the compactness of the data structure. Nevertheless, adhering to binary representation degrades compactness of SeqBDDs [64].

Decision diagrams are used extensively in the field of symbolic model checking. One of the most important virtues of symbolic model checking is the generation of counterexample in case a given model violates the tested property. Many heuristics were introduced in the literature that aim at producing counterexamples that are more informative and understandable. A counterexample is simply a trace. Given the state space, starting from a start state, the counterexample shows the sequence of states the system will end in a reachable *bad state*. That trace, or path, can be finite or infinite (if it contains a cycle). For instance, safety properties checked through finite trace, while liveness properties are checked through infinite traces. However, traces do not always consist of one path. In the case of probabilistic model checking, often a vast number of paths compose the counterexample. One way that has been introduced to compactly store the latter type of counterexample is regular expressions [23, 36].

In this chapter, we introduce *sequence decision diagrams* (SeqDDs), which can encode arbitrary finite sets of strings over an alphabet. SeqDDs can be viewed as a multi-valued variation of SeqBDDs. SeqDDs do not constrain a priori the number of levels, in fact, they do not really have an inherent concept of levels (or variables associated to a node). Instead, they simply require that, on any instance of the diagram, the number of paths and the lengths of these paths be finite.

2.1.4 Notation

Given alphabet $\Sigma = \{s_1, \cdots, s_m\}$, with $m \in \mathbb{N}$, let Σ^* be the set of strings over Σ, i.e., $\Sigma^* = \{a_1 \cdots a_k : k \geq 0, \forall h, 1 \leq h \leq k, a_h \in \Sigma\}$. We introduce the following notation to discuss SeqDDs encoding a finite language $\mathcal{Y} \subset \Sigma^*$:

- If $\mathcal{Y} = \emptyset$, then *height*$(\mathcal{Y}) = \perp$, “undefined”. Otherwise, the height of $\mathcal{Y}$ is the length of the longest string in it, *height*$(\mathcal{Y}) = \max\{|\sigma| : \sigma \in \mathcal{Y}\}$.

- *lengths*$(\mathcal{Y}) = \{k \in \mathbb{N} : \exists \sigma \in \mathcal{Y}, |\sigma| = k\}$, the set of all string lengths in $\mathcal{Y}$.

- For $k \in \mathit{lengths}(\mathcal{Y})$, $\mathcal{Y}_k = \{\sigma \in \mathcal{Y} : |\sigma| = k\}$, the strings of length k in $\mathcal{Y}$, and $\mathcal{Y}_{<k} = \{\sigma \in \mathcal{Y} : |\sigma| < k\}$, the strings of length less than k in $\mathcal{Y}$.

- For $a \in \Sigma$, $\mathcal{Y}/a = \{\sigma \in \Sigma^* : a \cdot \sigma \in \mathcal{Y}\}$, the strings that, preceded by a, form a string in $\mathcal{Y}$.

- For $k \in \mathit{lengths}(\mathcal{Y})$ and $a \in \Sigma$, $\mathcal{Y}_k/a = \{\sigma \in \Sigma^{k-1} : a \cdot \sigma \in \mathcal{Y}_k\}$, the strings that, preceded by a, form a string of length k in $\mathcal{Y}$.

- $||\mathcal{Y}|| = \sum_{\sigma \in \mathcal{Y}} |\sigma|$, the total number of symbols in $\mathcal{Y}$, not to be confused with $|\mathcal{Y}|$, the number of strings in $\mathcal{Y}$.

2.2 Sequence decision diagrams

2.2.1 Non-canonical SeqDDs

This section defines a class of decision diagrams that can encode any finite subset of Σ^*, that is any set of the form

$$\{\sigma_1, \cdots, \sigma_n : n \in \mathbb{N}, \forall j, 1 \leq j \leq n, \sigma_j \in \Sigma^*\}.$$

Note that the empty set $\emptyset$ as well as $\{\epsilon\}$, the set containing only the empty string, are two of the the sets that we must be able to encode.

Definition 1 *A sequence decision diagram (SeqDD) is a directed acyclic finite graph in which*

- *there are two* terminal *nodes, with no outgoing edges,* **0** *and* **1***;*

- *a* nonterminal *node p has $m+1$ outgoing edges, each one labeled with a different element from $\Sigma \cup \{\epsilon\}$; we write $p[a] = q$ to indicate that the outgoing edge labeled with $a \in \Sigma \cup \{\epsilon\}$ points to node q, which can be terminal or nonterminal.*

Definition 2 *The set of strings $\mathcal{X}(p)$ encoded by a SeqDD node p is recursively defined as:*

$$\mathcal{X}(p) = \begin{cases} \emptyset, \text{ the empty set} & \text{if } p = \mathbf{0}, \\ \{\epsilon\}, \text{ the set containing only the empty string} & \text{if } p = \mathbf{1}, \\ \bigcup_{a \in \Sigma \cup \{\epsilon\}} \{a \cdot \sigma : \sigma \in \mathcal{X}(p[a])\} & \text{otherwise,} \end{cases}$$

where "$\cdot$" denotes the string concatenation operator.

We now prove that, given an arbitrary finite set of strings $\mathcal{Y} \subset \Sigma^*$, we can encode $\mathcal{Y}$ using a SeqDD. More precisely, we can build a SeqDD with a single root node r (i.e., a node not having any incoming edges), such that $\mathcal{X}(r) = \mathcal{Y}$.

Theorem 1 *Given a finite set of strings $\mathcal{Y} \subset \Sigma^*$, there exists a single-root SeqDD whose root p satisfies $\mathcal{X}(p) = \mathcal{Y}$.*

Proof. The proof proceeds by induction on $\|\mathcal{Y}\|$, the total number of symbols in $\mathcal{Y}$.

If $\|\mathcal{Y}\| = 0$, then $\mathcal{Y} = \emptyset$ or $\mathcal{Y} = \{\epsilon\}$. In the case of $\mathcal{Y} = \emptyset$, we can let p be the $\mathbf{0}$-terminal. In case of $\mathcal{Y} = \{\epsilon\}$, we can let p be the $\mathbf{1}$-terminal.

If $\|\mathcal{Y}\| = k > 0$, assume the theorem holds for any set $\mathcal{Y}'$ with $\|\mathcal{Y}'\| < k$. Clearly, $\|\mathcal{Y}_a\| < k$ and, if $\epsilon \in \mathcal{Y}$, then $\mathcal{Y} = \{\epsilon\} \cup \bigcup_{a \in \Sigma} a \cdot \mathcal{Y}_a$, else $\mathcal{Y} = \bigcup_{a \in \Sigma} a \cdot \mathcal{Y}_a$. Then, if $\epsilon \in \mathcal{Y}$, we can define a node p, with $p[\epsilon] = \mathbf{1}$ and $p[a] = q_a$, where q_a is a node that encodes $\mathcal{Y}_a$, which exists, by induction, since $\|\mathcal{Y}_a\| < k$, for $a \in \Sigma$. The case where $\epsilon \notin \mathcal{Y}$ is exactly analogous, except that we set $p[\epsilon] = \mathbf{0}$. ■

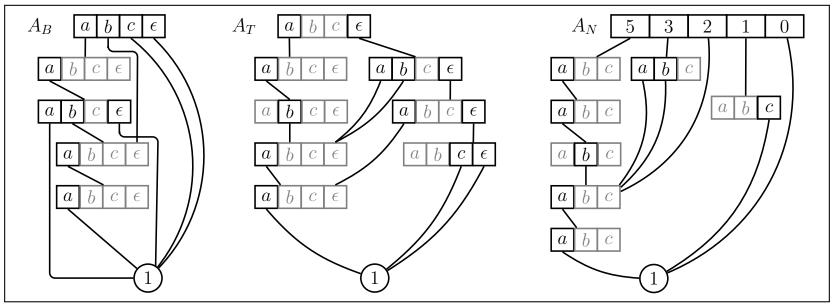


Figure 2.2: A SeqDDB, a SeqDDT, and a SeqDDN encoding $\mathcal{Y} = \{aa, aaa, aabaa, baa, c, \epsilon\}$. Indices in gray point to terminal $\mathbf{0}$ (not represented for clarity).

By definion SeqDDs are general non-canonical encoding of finite languages. Any set $\mathcal{Y} \subset \Sigma^*$ can be encoded by infinitely many SeqDDs because, if a node r encodes $\mathcal{Y}$, any node r' with $r'[a] = \mathbf{0}$ for each $a \in \Sigma$ and $r'[\epsilon] = r$ also encodes $\mathcal{Y}$, and the "insertion" of such "useless nodes" can be repeated at will (indeed, not just above the root, but anywhere along any path in the SeqDD). Thus, we now describe possible sets of restrictions to ensure canonicity, namely

- No *duplicate nodes* are allowed: the SeqDD cannot contain two nonterminal nodes p and q such that $p[a] = q[a]$ for every $a \in \Sigma \cup \{\epsilon\}$.
- No *empty nodes* are allowed: the SeqDD cannot contain a nonterminal node p such that $p[a] = \mathbf{0}$ for every $a \in \Sigma \cup \{\epsilon\}$.
- No *ϵ-nodes* are allowed: the SeqDD cannot contain a nonterminal node p such that $p[a] = \mathbf{0}$ iff $a \in \Sigma$.

Informally, canonicity is achieved by additionally "pushing" ϵ-edges (not pointing to $\mathbf{0}$) toward the bottom, or toward the top, of the diagram (Figure 2.2).

2.2.2 Canonical SeqDDs with ϵ at the bottom

Definition 3 *A (canonical, ϵ-at-the-bottom) SeqDDB is a SeqDD with no duplicate nodes, no empty nodes, no ϵ-nodes, and such that, for any nonterminal node p, either $p[\epsilon] = \mathbf{0}$ or $p[\epsilon] = \mathbf{1}$.*

Theorem 2 *Given a finite set of strings $\mathcal{Y} \subset \Sigma^*$, there is a unique single-root SeqDDB whose root p satisfies $\mathcal{X}(p) = \mathcal{Y}$.*

Proof. If $height(\mathcal{Y}) = \perp$, then $\mathcal{Y} = \emptyset$, and the canonicity restrictions imply that $p = \mathbf{0}$ is the only SeqDDB node encoding $\mathcal{Y}$. If $height(\mathcal{Y}) = 0$, then $\mathcal{Y} = \{\epsilon\}$, and the same restrictions imply that $p = \mathbf{1}$ is the only SeqDDB node encoding $\mathcal{Y}$. If $height(\mathcal{Y}) = k > 0$, assume the theorem holds for any $\mathcal{Y}'$ with $height(\mathcal{Y}') < k$. Clearly, $height(\mathcal{Y}/a) < k$ and, if $\epsilon \in \mathcal{Y}$, then $\mathcal{Y} = \{\epsilon\} \cup \bigcup_{a\in\Sigma} a \cdot \mathcal{Y}/a$, otherwise $\mathcal{Y} = \bigcup_{a\in\Sigma} a \cdot \mathcal{Y}/a$. Then, if $\epsilon \in \mathcal{Y}$, we can define node p, with $p[\epsilon] = \mathbf{1}$ and, for each $a \in \Sigma$, $p[a] = q_a$, where q_a is the unique node encoding $\mathcal{Y}/a$ (by induction, q_a exist since $height(\mathcal{Y}/a) < k$). Note that we might have $\mathcal{Y}/a = \mathcal{Y}/b$ for $a \neq b$, this simply means that the two corresponding edges in p point to the same SeqDDB node (indeed nodes are shared across any of the descendants of p, to avoid duplicates). No other node q encoding $\mathcal{Y}$ can exist because it would have to differ from p in at least one index $a \in \Sigma$, while we must have $p[\epsilon] = q[\epsilon] = \mathbf{1}$. By inductive assumption, SeqDDB's $p[a]$ and $q[a]$ cannot encode the same set, that is, $\mathcal{X}(p[a]) = \mathcal{Y}/a \neq \mathcal{X}(q[a])$, thus there is a string $a \cdot \sigma'$ in $\mathcal{X}(p)$ and not in $\mathcal{X}(q)$, or vice versa. The case where $\epsilon \notin \mathcal{Y}$ is analogous, except that $p[\epsilon] = \mathbf{0}$. ∎

2.2.3 Canonical SeqDDs with ϵ at the top

For the alternative definition where we allow "ϵ at the top", it is easier to recast the definition of quasi-reduced MDDs [18] as a special case of SeqDDs.

Definition 4 *A (canonical, single-root) k-level MDD is the terminal node* $\mathbf{1}$*, if $k = 0$, or, if $k > 0$, it is a single-root SeqDD with no duplicate nodes, no empty nodes, no ϵ-nodes, and with root p such that $p[\epsilon] = \mathbf{0}$ and, for $a \in \Sigma$, $p[a]$ is a $(k-1)$-level MDD or* $\mathbf{0}$*.*

It is easy to see that the root p of a k-level MDD encodes a nonempty set of strings of fixed length k, that is, $\mathcal{X}(p) \subseteq \Sigma^k$.

Definition 5 *A k-level SeqDDT is a SeqDD without duplicate, empty, or ϵ-nodes whose root node p is such that, for $a \in \Sigma$, $p[a]$ is* $\mathbf{0}$ *or the root of a $(k-1)$-level MDD, while $p[\epsilon]$ is* $\mathbf{0}$ *or the root of an h-level SeqDDT, $h < k$.*

Thus, it is easy to prove by induction that the root p of a k-level SeqDDT encodes a nonempty set of strings of length k, $\bigcup_{a\in\Sigma} \mathcal{X}(q[a])$, plus a possibly empty set of strings of length less than k, $\mathcal{X}(q[\epsilon])$.

Theorem 3 *Given a finite language $\mathcal{Y} \subset \Sigma^*$, there exists a unique single-root SeqDDT with root p such that $\mathcal{X}(p) = \mathcal{Y}$.*

Proof. If $height(\mathcal{Y}) = \bot$, then $\mathcal{Y} = \emptyset$, and the canonicity restrictions imply that $p = \mathbf{0}$ is the only SeqDDT encoding $\mathcal{Y}$. If $height(\mathcal{Y}) = 0$, then $\mathcal{Y} = \{\epsilon\}$, and the same restrictions imply that $p = \mathbf{1}$ is the only SeqDDT encoding $\mathcal{Y}$. If instead $height(\mathcal{Y}) = k > 0$, assume that the theorem holds for any set $\mathcal{Y}'$ with $height(\mathcal{Y}') < k$. Since $\mathcal{Y} = \mathcal{Y}_{<k} \cup \bigcup_{a\in\Sigma} a \cdot \mathcal{Y}_k/a$,

we can define node p such that, for $a \in \Sigma$, $p[a] = q_a$ with $\mathcal{X}(q_a) = \mathcal{Y}_k/a$, while $p[\epsilon] = q_\epsilon$ with $\mathcal{X}(q_\epsilon) = \mathcal{Y}_{<k}$. By inductive hypothesis, nodes q_a and q_ϵ are unique, as they all encode sets of height less than k and, since $\mathcal{Y}_k/a$ contains only strings of length $k-1$, q_a is in particular the root of an MDD, i.e., $q_a[\epsilon] = \mathbf{0}$. Then, node p is also the only node encoding $\mathcal{Y}$ since any other node p' would have to differ from p in at least one child. If $p[\epsilon] \neq p'[\epsilon]$, there must exists a string σ of length less than k in $\mathcal{X}(p[\epsilon])$, thus $\mathcal{X}(p)$, and not in $\mathcal{X}(p'[\epsilon])$, thus $\mathcal{X}(p')$, or vice versa. If there is an $a \in \Sigma$ with $p[a] \neq p'[a]$, there must exists a string σ in $\mathcal{X}(p[a])$ and not in $\mathcal{X}(p'[a])$, so that $a \cdot \sigma$ is in $\mathcal{X}(p)$ and not in $\mathcal{X}(p')$, or vice versa ($a \cdot \sigma$ cannot possibly be in $\mathcal{X}(p'[\epsilon])$ as it is of length k). Either way, p' cannot encode the same set as p. ■

2.2.4 An alternative canonical definition without ϵ

Unlike SeqDDBs, SeqDDTs rely on some concept of level for the nodes of the decision diagram. More specifically, the nodes in a SeqDDT encode all the maximum-length strings using the children corresponding to the elements of Σ, and postpone the encoding of the remaining, shorter, strings to the child corresponding to ϵ (Figure 2.2). An almost equivalent encoding for a set $\mathcal{Y}$ is then one where the strings of $\mathcal{Y}$ are partitioned according to their length, and the top node makes a decision based on the length of the string σ being searched, not on the first symbol of σ. This leads us to a third, slightly different in spirit but essentially equivalent, definition.

Definition 6 *A SeqDDN is a set of "sparse" root nodes, each root r having a finite set $\mathcal{R}$ of outgoing edges labeled with different elements $k \in \mathbb{N}$, such that $r[k]$ points to a k-level*

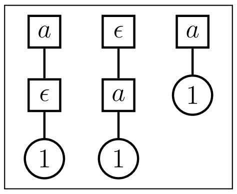


Figure 2.3: Canonicity of sequence decision diagrams.

MDD. The set encoded by r is $\bigcup_{k\in\mathcal{R}} \mathcal{X}(r[k])$.

Note that sharing of nodes across various MDDs of a single-root SeqDDN, as for those of the equivalent SeqDDT, is not only possible, but required, since we seek a canonical form. If the sets $\mathcal{X}(r[k_1]) \subseteq \Sigma^{k_1}$ and $\mathcal{X}(r[k_2]) \subseteq \Sigma^{k_2}$ encoded by MDD nodes p_1 and p_2 satisfy

$$\exists\gamma_1, \gamma_2, \{\sigma\in\Sigma^k : \gamma_1\cdot\sigma\in\mathcal{X}(r[k_1])\} = \{\sigma\in\Sigma^k : \gamma_2\cdot\sigma\in\mathcal{X}(r[k_2])\} = \mathcal{W} \neq \emptyset,$$

then the node p encoding $\mathcal{W}$ is shared by the MDDs rooted at p_1 and p_2.

Theorem 4 *Given a finite set of strings $\mathcal{Y} \subset \Sigma^*$, there is a unique single-root SeqDDN rooted at r such that $\mathcal{X}(r) = \mathcal{Y}$.*

Proof. The proof is immediate. If $\mathcal{Y} = \emptyset$, then only node r with $\mathcal{R} = \emptyset$ encodes $\mathcal{Y}$. Otherwise, write $\mathcal{Y} = \bigcup_{k\in lengths(\mathcal{Y})} \mathcal{Y}_k$. Then, each $\mathcal{Y}_k$ is canonically encoded by an MDD rooted at a node p_k, and the root node of the SeqDDN is simply r with a set $\mathcal{R} = lengths(\mathcal{Y})$, and such that $r[k] = p_k$, for each $k \in lengths(\mathcal{Y})$. Of course, MDD nodes must be shared across MDDs, not just within each MDD. It is obvious that this SeqDDN is the unique encoding of $\mathcal{Y}$. ■

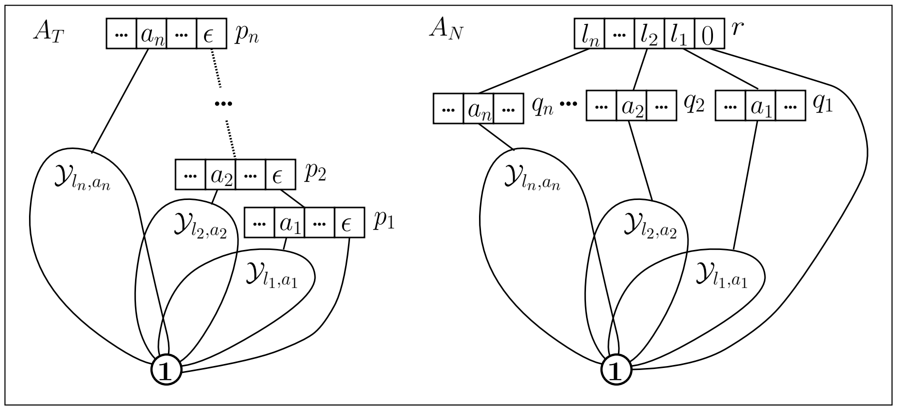


Figure 2.4: The structure of a SeqDDT and a SeqDDN encoding the same set.

2.2.5 Comparing compactness of SeqDDT and SeqDDN

We begin by comparing the size of the SeqDDT and SeqDDN encoding a set $\mathcal{Y}$, since both definitions rely on the length of the strings in $\mathcal{Y}$.

Theorem 5 *Given a finite set of strings $\mathcal{Y} \subset \Sigma^*$, the numbers of edges in SeqDDT A_T and SeqDDN A_N encoding $\mathcal{Y}$ satisfy*

$$edges(A_T) + 1 \geq edges(A_N) \geq edges(A_T) - (|lengths(\mathcal{Y})| - 2)|\Sigma| + 1.$$

Proof. The proof is based on the common structure exhibited by A_T and A_N. Consider first the case where $\epsilon \in \mathcal{Y}$, shown in Figure 2.4, where $n + 1 = |lengths(\mathcal{Y})|$, i.e., n is the number of different string lengths in $\mathcal{Y}$ not counting the length 0 of the empty string. The key observation is that A_T and A_N are largely the same. Namely, the MDDs encoding any of the non-empty sets $\mathcal{Y}_{l_k,a_k}$, for $l_k \in lengths(\mathcal{Y})$ and $a_k \in \Sigma$, are present in both A_T and A_N, so we can simply let e be the number of edges needed to encode them as a whole, in either representation. Then, $edges(A_T) = e + (x_n + 1) + \cdots + (x_1 + 1)$, where x_k is the number of edges leaving node p_k not counting its ϵ-edge, thus it is also the number of edges leaving q_k in A_N. On the other hand, $edges(A_N) = e + (n+1) + x_n + \delta_{n-1}x_{n-1} + \cdots + \delta_1 x_1$, where the term $(n+1)$ counts the edges leaving the root r, while $\delta_k = 0$ if q_k is a node already present in the encoding of the MDDs $\mathcal{Y}_{l_m, a_m}$, for $l_m \in lengths(\mathcal{Y})$ with $l_m > l_k$ and $a_k \in \Sigma$, and $\delta_k = 1$ otherwise. In other words, the indicators δ_k's are needed because, except for q_n, any other q_k might happen to duplicate an already existing node in the MDD portion of A_N, while this is not possible for any node p_k, as having an ϵ-edge makes it for sure different from any MDD node. Then, since x_k can be as large as $|\Sigma|$, we can conclude that

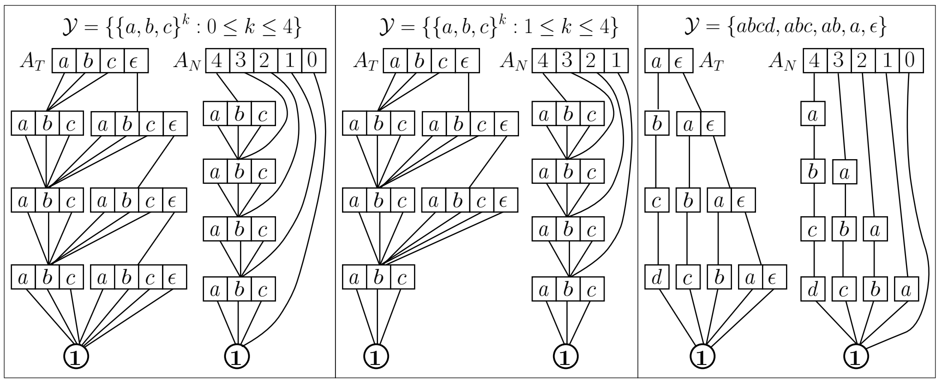
$$edges(A_T) + 1 \geq edges(A_N) \geq edges(A_T) + 1 - (n-1)|\Sigma|.$$

If instead $\epsilon \notin \mathcal{Y}$, the same approach is applicable, except that $n = |lengths(\mathcal{Y})|$, p_1 in A_T does not contain an ϵ-edge, and r does not contain a **0**-edge. We can then write $edges(A_T) = e + (x_n + 1) + \cdots + (x_2 + 1) + \delta_1 x_1$, since now q_1 not only does not have an ϵ-edge, but could be already present in the MDD portion of A_T, and $edges(A_N) = e + n + x_n + \delta_{n-1}x_{n-1} + \cdots + \delta_1 x_1$, since r does not have the **0**-edge (it is important to note that q_1 and p_1 coincide when $\epsilon \notin \mathcal{Y}$, thus either they both coincide with an existing MDD node, or neither of them does, that is, δ_1 is the correct indicator for both). Then, we can conclude that

$$edges(A_T) + 1 \geq edges(A_N) \geq edges(A_T) + 1 - (n-2)|\Sigma|.$$

Recalling that $n = |lengths(\mathcal{Y})| - 1$ when $\epsilon \in \mathcal{Y}$ and $n = |lengths(\mathcal{Y})|$ when $\epsilon \notin \mathcal{Y}$, we conclude that the theorem always holds. ■

Figure 2.5 shows that both the lower and upper bounds on the size of A_N with respect to A_T can actually be achieved. Specifically, the first two panels show how we



$\mathcal{Y} = \{\{a,b,c\}^k : 0 \le k \le 4\}$
$\mathcal{Y} = \{\{a,b,c\}^k : 1 \le k \le 4\}$
$\mathcal{Y} = \{abcd, abc, ab, a, \epsilon\}$
A_T
A_N

Figure 2.5: Examples achieving the bounds of Theorem 5.

can have $edges(A_N) = edges(A_T) - (|lengths(\mathcal{Y})| - 2)|\Sigma| + 1$, assuming $\Sigma = \{a, b, c\}$, for the cases $\epsilon \in \mathcal{Y}$ and $\epsilon \notin \mathcal{Y}$, respectively, while the third panel shows how we can have $edges(A_N) = edges(A_T) + 1$.

2.3 Compactness of canonical SeqDDs

We now discuss the size of our SeqDDs, where the size of a SeqDD A is the number of edges it contains, $edges(A)$, rather than the number of nodes. Given the structural differences between a SeqDDB and a SeqDDT, we compare them by thinking of them as finite automata. A closer look at a SeqDDB shows that it can be easily converted into a DFA (Theorem 6). On the other hand, a SeqDDT can be converted into a restricted type of NFA.

2.3.1 DFA representation of SeqDDB

Given a SeqDDB A_B encoding a finite language $\mathcal{Y} \subset \Sigma^*$, we can build an equivalent DFA $M = (Q, \Sigma, \delta, q_0, F)$. If $A_B = \mathbf{0}$ then $M = (\{q_0\}, \Sigma, \delta, q_0, \emptyset)$. Otherwise, we first define the states Q in terms of the nodes in A_B: every nonterminal node q in A_B corresponds to a state $q \in Q$, while node $\mathbf{1}$ in A_B corresponds to new state $f \in Q$ and node $\mathbf{0}$ corresponds to a new trap state $t \in Q$.

The initial state q_0 corresponds to A_B's root while the transition function $\delta : Q \times \Sigma \to Q$ is such that, for every $a \in \Sigma$ and edge $q[a] = p$ in A_B, there is a corresponding transition $\delta(q, a) = p$ and, if $q[\epsilon] = \mathbf{1}$, no transition is added, but q is added to the accepting states F. Lastly, state f is also added to F.

Theorem 6 *Given a SeqDDB A_B encoding a finite language $\mathcal{Y} \subset \Sigma^*$, building an equivalent minimized DFA M requires linear time in the size of A_B.*

Proof. The proof is direct from the translation algorithm above. ■

For memory efficiency, decision diagrams can be stored in a sparse form. In the case of a sparse SeqDDB, this corresponds to a *partial* DFA, and the translation is analogous to the non-sparse version just discussed. From now on, we consider sparse representations for all canonical forms of SeqDD and for partial DFAs.

2.3.2 NFA representation of SeqDDT

To discuss the translation of a SeqDDT into an equivalent NFA, we first define RNFAs, a restricted version of NFAs, keeping in mind that our goal is to facilitate size comparisons between a SeqDDB and a SeqDDT. To that end, our RNFA definition resembles the structure of SeqDDT while respecting the key characteristics of ordinary NFAs when encoding a finite language.

Definition 7 *A restricted NFA (RNFA) is an acyclic NFA $N = (Q, \Sigma, \delta, Q_I, Q_F)$, where both Q_I and Q_F are singletons sets and, for each state $q \in Q$, the following condition holds: at most one outgoing ϵ-transition is allowed, and if $k = \max(lengths(L(q)))$ then all strings in $\bigcup_{a\in\Sigma} L(\delta(q,a))$ have length $k-1$ and all strings in $L(\delta(q,\epsilon))$ have length at most $k-1$. This value k is called the* level of q.

A *minimized* RNFA enforces the following restriction rules.

- No *duplicate states* are allowed: An RNFA cannot contain q and p such that $L(q) = L(p)$.
- No *empty states* are allowed: An RNFA cannot contain a state $q \in Q \setminus Q_I$ such that $L(q) = \emptyset$.
- No *ϵ-states* are allowed: An RNFA cannot contain a state $q \in Q \setminus Q_F$ such that $L(q) = \{\epsilon\}$.

Any RNFA can be converted to an equivalent minimized RNFA using Algorithm 2.1, an adaptation of the bucket-sort based OBDD reduction algorithm proposed in [68]. The minimized RNFA for a given language is unique.

The following lemma affirms that RNFAs, like DFAs, can recognize any finite language (unlike DFAs, they obviously cannot accept any infinite language).

Lemma 7 *If $\mathcal{Y} \subset \Sigma^*$ is a finite language, there exists an RNFA N to accept $\mathcal{Y}$.*

Proof. The proof of existence is analogous to the one of Theorem 3. ∎

Algorithm 2.1 Algorithm to canonize a RNFA.

```
1: function CANONIZE( $p$ : RNFA, $q$ : SeqDDB)
2:     declare local $RNFA\ s,\ v$
3:     declare local $vector\ t$      ▷ sorted vector according to a predefined alphabet order
4:     declare local $list\langle s, t\rangle\ L$
5:     declare local $list\langle bucket\rangle\ nonempty$
6:     $devide$ $p$'s nodes by levels s.t. the final state $f$ is at level-0 and a node $n$
   recognizing strings of length $k$ is at level-$k$.
7:     for $k = 1$ to $lengths(\mathcal{Y})$ do
8:         create $L$ containing nodes $s$ of level-$k$ and the associated vector of
   successors $v$ for each $s$ in $L$.
9:         create bucket$_0$ containing all $s$ in $L$ ▷ starting with all nodes in one bucket
10:        add bucket$_0$ to $nonempty$ list
11:        for $a \in \Sigma$ do                    ▷ run an $|\Sigma|$-phase bucket sort algorithm
12:            for bucket $\mathcal{U}$ in the $nonempty$ list do
13:                create new bucket-$a$
14:                for $s \in \mathcal{U}$ do      ▷ eventually divide into buckets of equivalent nodes
15:                    add $s$ to bucket-$a[v]$ s.t. $(s[a] = v$ or $(s[a] = t$ and $R[t] = v))$ ▷
   $v$ is the minimized representation of $t_i$
16:                    add bucket-$a[v]$ to $nonempty$ list, if not added yet
17:                    delete bucket $\mathcal{U}$ from the $nonempty$ list
18:        create new list $R$ or clear the old one, if exists.
19:        for bucket $\mathcal{U}$ in the $nonempty$ list do          ▷ merge equivalent nodes
20:            let $v$ be any $s \in \mathcal{U}$
21:            for $s \in \mathcal{U}$ do
22:                add $\langle s, v\rangle$ to $R$        ▷ mark duplicate nodes by their new equivalent
23:        clear lists and vectors except $R$
24:    delete unreachable nodes
```

If SeqDDT A_T with a single root node r encodes a finite language $\mathcal{Y} \subset \Sigma^*$, the equivalent RNFA $T = (Q, \Sigma, \delta, Q_I, Q_F)$ is built as follows. Each nonterminal node q of A_T corresponds to a state $q \in Q$; terminal node $\mathbf{1}$ of A_T corresponds to a new state $\mathbf{1} \in Q$, and $F = \{\mathbf{1}\}$; finally, $Q_I = \{r\}$ (note that, if $r = \mathbf{0}$, we also must add r to Q). The transition function $\delta : Q \times \Sigma \cup \{\epsilon\} \to Q$ is such that, for every edge $q[a] = p$ in A_T with $a \in \Sigma \cup \{\epsilon\}$, there is a corresponding transition $\delta(q, a) = p$. Thus, in particular, if $r = \mathbf{0}$, then $T = (\{\mathbf{0}\}, \Sigma, \emptyset, \{\mathbf{0}\}, \{\mathbf{1}\})$, and the encoded language is $\mathcal{Y} = \emptyset$, while, if $A_T = \mathbf{1}$, then

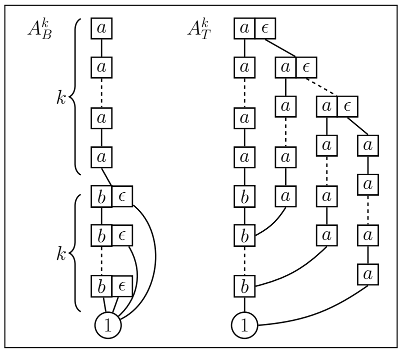


Figure 2.6: Example of quadratic growth when translating SeqDDB into SeqDDT.

$T = (\{\mathbf{1}\}, \Sigma, \emptyset, \{\mathbf{1}\}, \{\mathbf{1}\})$ and the encoded language is $\mathcal{Y} = \{\epsilon\}$.

From the conversion process, it is easy to see that the number of transitions in the resulting DFA equals the number of edges in the equivalent SeqDDB excluding ϵ-edges. Hence, we can define the DFA size to be equal to the number of transitions plus the number of final states excluding the one corresponding to terminal **1**, $|M| = |\delta| + |F| - 1$. On the other hand, since the number of transitions in the resulting RNFA equal the number of edges in SeqDDN minus *lengths*($\mathcal{Y}$), we can define the size of an RNFA to be equals to the number of transitions plus the number of initial states, $|N| = |\delta| + |V_0|$.

2.3.3 SeqDD Compactness Comparison by Means of Finite Automata

To study the relative compactness of canonical SeqDDs, we first discussed bounds on the number of states for equivalent DFAs and RNFAs; these are trivially reflected in similar bounds for SeqDDB's and SeqDDT's. To obtain bounds on the number of transitions, one could just multiply the state bounds by the alphabet size, but we are really interested in the actual number of edges for equivalent SeqDDs, thus partial FAs. This section shows that bounds similar to those for states hold also for edges.

Theorem 8 *Given a DFA $M = (Q, \Sigma, \delta_D, q_0, F)$ with n states encoding a finite language $\mathcal{Y} \subset \Sigma^*$, an equivalent minimized RNFA N has $O(n^2)$ states.*

Proof. For each state $q \in Q$ and $k = 0, \ldots, \mathit{height}(\mathcal{Y})$, let $L(q, k) = L(q) \cap \Sigma^k$. Then, we build an equivalent RNFA N with states organized by level:

- Level 0 of the RNFA contains a single accepting state f.
- Level k contains a state $\langle q,k \rangle$ for each nonempty $L(q, k)$.
- The initial state of N is $\langle q_0, \max \mathit{lengths}(\mathcal{Y}) \rangle$.
- The transition function δ_N of N satisfies
  - For each state $\langle q,k \rangle$ with $k > 0$ in N and for each $a \in \Sigma$:

    $\langle p,k-1 \rangle \in \delta_N(\langle q,k \rangle, a)$ iff $\delta_D(q, a) = p$.
  - For each state $\langle q,k \rangle$ in N, let h be the largest integer less than k such that state $\langle q,h \rangle$ exists in N; if such state exists, then $\langle q,h \rangle \in \delta_N(\langle q,k \rangle, \epsilon)$.

Note that the resulting RNFA might not be minimized, in the sense that it is possible that $\langle q,k \rangle$ and $\langle p,k \rangle$ encode the same language, in which case they should be merged. In any case, however, the number of states of the RNFA is at most equal to the number of states of the DFA times the maximum length of a string in $\mathcal{Y}$, which, again, is at most equal to the number of states. Thus the number of RNFA states is at most quadratic the number of DFA states. As the two automata obviously accept the same language $\mathcal{Y}$, the proof is complete. ■

To show that the growth of Theorem 8 is indeed possible, consider the family of

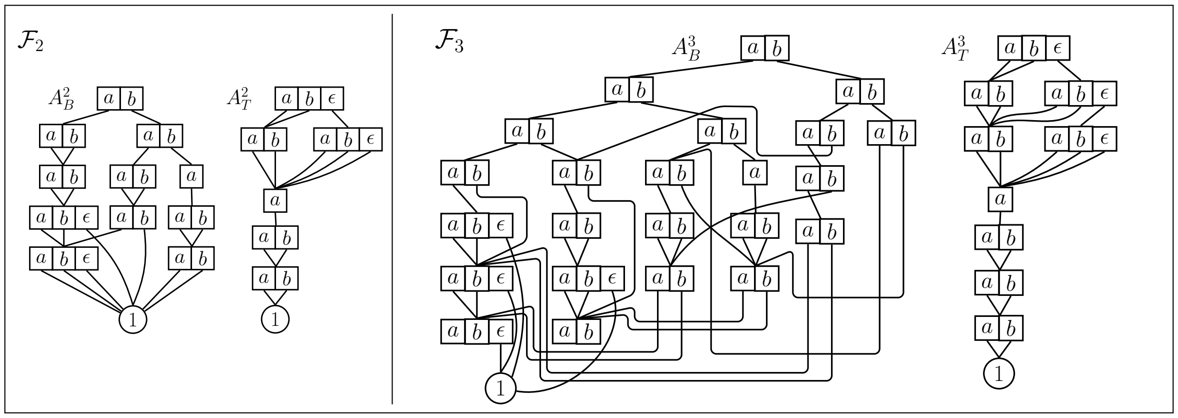


Figure 2.7: Example of exponential growth when translating SeqDDT into SeqDDB.

languages $\mathcal{G} = \{\mathcal{G}_k : k \in \mathbb{N}\}$ over $\{a, b\}$. Let $\mathcal{G}_k = \{a^k b^k, a^k b^{k-1}, \cdots, a^k b, a^k\}$, so that $||\mathcal{G}_k|| = 3(k+1)k/2$. Then, the SeqDDT A_T^k encoding $\mathcal{G}_k$ contains $k^2 + 3k$ edges, while the SeqDDB A_B^k encoding $\mathcal{G}_k$ contains $3k$ edges (see Figure 2.6).

Theorem 9 *Given a minimized RNFA N with n states encoding a finite language $\mathcal{Y} \subset \Sigma^*$, an equivalent minimized DFA has at most $O(2^n)$ states.*

Proof. The proof is immediate given the well known fact that an NFA-to-DFA conversion may result in an exponential increase in the number of states. ■

Since RNFAs are a restricted form of NFAs, however, one may wonder whether an exponential growth can actually occur. To show that this is the case, consider the family of languages $\{\mathcal{F}_k : k \in \mathbb{N}\}$ with $\mathcal{F}_k = \{xay : x, y \in \{a, b\}^*, |x| \leq k, |y| = k\}$. Then, the SeqDDT A_T^k encoding $\mathcal{G}_k$ contains $7k - 1$ edges while the SeqDDB A_B^k encoding $\mathcal{G}_k$ contains $\Omega(2^k)$ edges (see Figure 2.7). This is similar to the well-known construction that demonstrates the proof of Theorem 9.

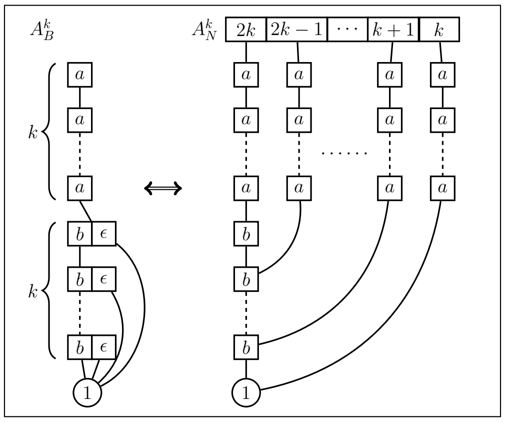


Figure 2.8: The family of languages demonstrating Theorem 10.

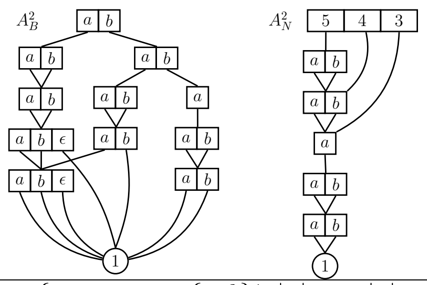
Theorem 10 *There exists a family of finite languages $\mathcal{G} = \{\mathcal{G}_k : k \in \mathbb{N}\}$ over $\{a, b\}$ such that the number of edges in the SeqDDN A_N^k encoding $\mathcal{G}_k$ is $O(k^2)$ while the number of edges in the SeqDDB A_B^k encoding $\mathcal{G}_k$ is $O(k)$.*

Proof. We exhibit such a family. Let $\mathcal{G}_k = \{a^k b^k, a^k b^{k-1}, \cdots, a^k b, a^k\}$, so that $||\mathcal{G}_k|| = 3(k+1)k/2$. Then, the SeqDDN A_N^k encoding $\mathcal{G}_k$ contains $k^2 + 3k + 1$ edges while the SeqDDB A_B^k encoding $\mathcal{G}_k$ contains $3k$ edges (see Figure 2.8). ■

Theorem 11 *There exists a family of finite languages $\mathcal{F} = \{\mathcal{F}_k : k \in \mathbb{N}\}$ over $\{a, b\}$ such that the number of edges in the SeqDDN A_N^k encoding any $\mathcal{F}_k$ is $O(k)$ while the number of edges in the SeqDDB A_B^k encoding $\mathcal{F}_k$ is $O(2^k)$.*

Proof. Again, we exhibit such a family. Let $\mathcal{F}_k = \{xay : x, y \in \{a, b\}^*, |x| \leq k, |y| = k\}$. Then, the SeqDDN A_N^k encoding $\mathcal{G}_k$ contains $5k + 2$ edges while the SeqDDB A_B^k encoding $\mathcal{G}_k$ contains $O(2^k)$ edges (see Figure 2.9). ■

$$\mathcal{F}_2 = \{xay : x, y \in \{a,b\}^*, |x| \leq 2, |y| = 2\}.$$



$$\mathcal{F}_3 = \{xay : x, y \in \{a,b\}^*, |x| \leq 3, |y| = 3\}.$$

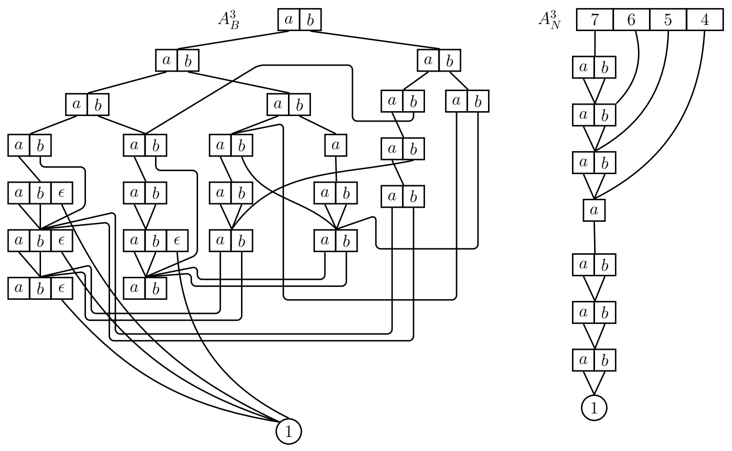


Figure 2.9: The family of languages demonstrating Theorem 11.

2.3.4 Summary

We showed in Theorem 5 that SeqDDTs and SeqDDNs are similar is size and structure. Next, we selected SeqDDNs to compare their compactness with SeqDDBs. It follows from Theorems 8 and 9 that there is no winner between SeqDDBs and SeqDDNs. Rather, SeqDDBs are more compact for certain languages and SeqDDNs are more compact for others. Thus, we need to design algorithms for both. The selection between the two canonical forms is left to the user, depending on the language to be encoded.

2.4 Algorithms on SeqDDs

We consider two types of algorithms: *set manipulation algorithms* and *substring manipulation algorithms*. Those of the first type take two or more canonical SeqDDs with the same canonicity rule and perform set operations such as *union* or *intersection*. Those of the second type input a canonical SeqDD and a string, and select strings satisfying a criterion for matching a substring, changing a substring into another, or shorten or lengthen a string.

As with all decision diagram algorithms, we adopt a recursive style. SeqDD nodes are stored in a *unique table* to ensure canonicity. An *operation cache* ensures efficiency by virtually eliminating repeated computations. Each of the following *set manipulation algorithms* has been developed for SeqDDB and SeqDDN representations: union, intersection, set difference, symmetric set difference, and concatenation. For instance, the *Intersection* algorithm for two SeqDDB's traverses them top-down and builds the resulting SeqDDB bottom-up (see the pseudo-code in Figure 2.2). SeqDDN set manipulation algorithms can be considered as shared MDD algorithms, since a SeqDDN is organized by the length of the strings encoded.

Various string manipulations can be performed. For example, the classical membership problem can be solved by a single trace, no longer than the *query size* $+ 1$, starting

Algorithm 2.2 *Intersection* operation on SeqDDBs.

```
function INTERSECTION( $p$ : SeqDDB, $q$ : SeqDDB)
    declare local SeqDDB $r$
    declare local integer $count$
    if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return 0                ▷ deal with the base cases
    if $p = q$ then return $p$
    if $p = \mathbf{1}$ then
        if $q[\epsilon] = \mathbf{1}$ then return $p$
        else return 0
    if $q = \mathbf{1}$ then return $Intersection(q, p)$
    if $Cache$ contains ⟨ Intersection, $\{p, q\} : r$⟩ then return $r$
    $count \leftarrow 0$
    for $a \in \Sigma$ do              ▷ Otherwise, perform $Intersection$ for each index $a \in \Sigma$
        $r[a] \leftarrow$ Intersection($p[a]$, $q[a]$)
        if $r[a] = \mathbf{0}$ then $count \leftarrow count + 1$
    if $count = |\Sigma|$ then $r \leftarrow \mathbf{0}$
    if $p[\epsilon] = \mathbf{1}$ and $q[\epsilon] = \mathbf{1}$ then                ▷ deal with $\epsilon$ case
        if $r = \mathbf{0}$ or $r = \mathbf{1}$ then $r \leftarrow \mathbf{1}$
        else $r[\epsilon] \leftarrow \mathbf{1}$
    $UniqueTableInsert(r)$
    $Cache \leftarrow$ ⟨ Intersection, $\{p, q\} : r$⟩
    return $r$
```

from the root and ending in either terminal **1** or **0**. Set manipulation algorithms can also become handy in performing string manipulations; for instance, the membership problem is solved by a set intersection, and string replacement can be solved using a combination of set difference, intersection, and union. However, if we want to perform substring manipulations, the use of set manipulation algorithms becomes inefficient, hence we developed specific substring manipulation algorithms.

The main advantage of using SeqDDs for substring manipulation lies in the ability to search or modify a set of strings at once, thanks to node sharing and *memoization*. For example, in a SeqDDB, replacing the first occurrence of a substring t with t' is done once for all strings sharing a prefix that contains t. Moreover, a shared suffix is processed the

Algorithm 2.3 *Union* operation on SeqDDBs

```
1: function Union( $p$ : SeqDDB, $q$ : SeqDDB)
2:     declare local SeqDDB $r$
3:     declare local integer $count$                         ▷ deal with the base cases
4:     if $p = \mathbf{0}$ then return $q$
5:     if $q = \mathbf{0}$ or $p = q$ then return $p$
6:     if $p = \mathbf{1}$ then
7:         if $q[\epsilon] = \mathbf{1}$ then return $q$
8:         else
9:             $r \leftarrow$ NewNode($q$)                   ▷ create a node $r$ equals to $q$
10:            $r[\epsilon] \leftarrow \mathbf{1}$
11:            $r \leftarrow$ UniqueTableInsert($r$)
12:            return $r$
13:    if $q = \mathbf{1}$ then return Union($q, p$)
14:    if $Cache$ contains $\langle$ $Union$, $\{p, q\} : r\rangle$ then return $r$
15:    $count \leftarrow 0$
16:    for $a \in \mathcal{S}$ do              ▷ Otherwise, perform $Union$ for each index $a \in \mathcal{S}$
17:        $r[a] \leftarrow$ Union(p[a], q[a])
18:        if $r[a] = \mathbf{0}$ then $count \leftarrow count + 1$
19:    if $count = |\Sigma|$ then $r \leftarrow \mathbf{0}$
20:    if $p[\epsilon] = \mathbf{1}$ or $q[\epsilon] = \mathbf{1}$ then $r[\epsilon] \leftarrow \mathbf{1}$      ▷ deal with $\epsilon$ case
21:    UniqueTableInsert($r$)
22:    $Cache \leftarrow \langle$ $Union$, $\{p, q\} : r\rangle$
23:    return $r$
```

first time we explore it; for other strings sharing that suffix the algorithm simply checks the *operation cache* for the result. A universal algorithm *replace* can replace, insert, or delete a specific substring: replacing ϵ by a string $t \neq \epsilon$ performs an insertion, while replacing t by ϵ performs a deletion. Of course, this can be refined by additionally providing to the algorithm specific substrings that must be found before and after the replacement location.

Algorithm 2.4 *Difference* operation on SeqDDBs

```
1: function DIFFERENCE( $p$ : SeqDDB, $q$ : SeqDDB)
2:     declare local SeqDDB $r$
3:     declare local integer $count$                              ▷ deal with the base cases.
4:     if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return $p$
5:     if $p = q$ then return $\mathbf{0}$
6:     if $p = \mathbf{1}$ then
7:         if $q[\epsilon] = \mathbf{1}$ then return $\mathbf{0}$
8:         else return $p$
9:     if $q = \mathbf{0}$ then return $p$
10:        if $p[\epsilon] = \mathbf{0}$ then return $p$
11:        else $r \leftarrow$ NEWNODE($p$)                       ▷ create a node $r$ equals to $p$
12:            $r[\epsilon] \leftarrow \mathbf{0}$
13:            $r \leftarrow$ UNIQUETABLEINSERT($r$)
14:            return $r$
15:    if $Cache$ contains $\langle$ $Difference$, $p, q : r\rangle$ then return $r$
16:    $count \leftarrow 0$
17:    for $a \in \mathcal{S}$ do               ▷ Otherwise, perform $Difference$ for each index $a \in \mathcal{S}$
18:        $r[a] \leftarrow$ DIFFERENCE(p[a], q[a])
19:        if $r[a] = \mathbf{0}$ then $count \leftarrow count + 1$
20:    if $count = |\Sigma|$ then $r \leftarrow \mathbf{0}$
21:    if $p[\epsilon] = \mathbf{1}$ and not($q = \mathbf{1}$ or $q[\epsilon] = \mathbf{1}$) then      ▷ deal with $\epsilon$ case
22:        if $r = \mathbf{0}$ or $r = \mathbf{1}$ then $r \leftarrow \mathbf{1}$
23:        else $r[\epsilon] \leftarrow \mathbf{1}$
24:    UNIQUETABLEINSERT($r$)
25:    $Cache \leftarrow \langle$ $Difference$, $p, q : r\rangle$
26:    return $r$
```

2.5 Applications of sequence decision diagrams

SeqDDs inherit the symbolic characteristics of decision diagrams, but with the additional ability to encode a set of strings of different lengths. SeqDDs are useful for applications that need to store and manipulate large sets of strings.

Algorithm 2.5 *Symmetric Difference* operation on SeqDDBs

```
function DIFFERENCE( $p$ : SeqDDB, $q$ : SeqDDB)
    declare local SeqDDB $r$
    declare local integer $count$                         ▷ deal with the base cases.
    if $p = \mathbf{0}$ then return $q$
    if $q = \mathbf{0}$ then return $p$
    if $p = q$ then return $\mathbf{0}$
    if $p = \mathbf{1}$ then
        if $q[\epsilon] = \mathbf{0}$ then return $q$
        $r \leftarrow$ NEWNODE($q$)                        ▷ create a node $r$ equals to $q$
        $r[\epsilon] \leftarrow \mathbf{0}$
        UNIQUETABLEINSERT($r$)
        return $r$
    if $q = \mathbf{1}$ then return XOR($q, p$)
    if $Cache$ contains $\langle$ XOR, $\{p, q\} : r\rangle$ then return $r$
    $count \leftarrow 0$
    for $a \in \mathcal{S}$ do                ▷ Otherwise, perform $Xor$ for each index $a \in \mathcal{S}$
        $r[a] \leftarrow$ XOR(p[a], q[a])
        if $r[a] = \mathbf{0}$ then $count \leftarrow count + 1$
    if $count = |\Sigma|$ then $r \leftarrow \mathbf{0}$
    if $\big(p[\epsilon] = \mathbf{1}$ and not$(q = \mathbf{1}$ or $q[\epsilon] = \mathbf{1})\big)$ or $\big(q[\epsilon] = \mathbf{1}$ and not$(p = \mathbf{1}$ or $p[\epsilon] = \mathbf{1})\big)$
then                                                        ▷ deal with $\epsilon$ case
        if $r = \mathbf{0}$ or $r = \mathbf{1}$ then $r \leftarrow \mathbf{1}$
        else $r[\epsilon] \leftarrow \mathbf{1}$
    UNIQUETABLEINSERT($r$)
    $Cache \leftarrow \langle$ XOR, $\{p, q\} : r\rangle$
    return $r$
```

2.5.1 Probabilistic witness generation

Probabilistic model checking aims to verify whether a probabilistic model satisfies a certain property [46]. We consider discrete states probabilistic models, namely, discrete-time Markov chains *(DTMCs).* Formally, a DTMC is defined by a 4-tuple $(Q, q_0, \mathbf{P}, L)$ where,

- Q is a finite set of states.

Algorithm 2.6 *Concatenation* operation on SeqDDBs

```
function $\textsc{Concatenate}(\ p : \text{SeqDDB},\ q : \text{SeqDDB})$
    declare local SeqDDB $r$
    declare local integer $count$                    ▷ deal with the base cases.
    if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return $\mathbf{0}$
    if $p = \mathbf{1}$ then return $q$
    if $q = \mathbf{1}$ then return $p$
    if $Cache$ contains $\langle$ Concatenate, $p, q : r\rangle$ then return $r$
    $count \leftarrow 0$
    for $a \in \mathcal{S}$ do          ▷ Otherwise, perform Concatenate for each index $i \in \mathcal{S}$
        $r[a] \leftarrow$ $\textsc{Concatenate}$(p[a], q)
        if $r[a] = \mathbf{0}$ then $count \leftarrow count + 1$
    if $count = |\Sigma|$ then $r \leftarrow \mathbf{0}$
    if $p[\epsilon] = \mathbf{1}$ then
        $r[\epsilon] \leftarrow \mathbf{0}$
        $r \leftarrow Union(r, q)$
    $\textsc{UniqueTableInsert}(r)$
    $Cache \leftarrow \langle$ Concatenate, $p, q : r\rangle$
    return $r$
```

- $q_0 \in Q$ is a start state.
- $\mathbf{P} : Q \times Q \rightarrow [0, 1]$ is a stochastic matrix.
- $L : Q \times \rightarrow 2^{AP}$ is a labeling function, where AP is a set of atomic propositions.

DTMCs admit probabilistic choices to resolve race conditions, which arise when multiple events are enabled and ready to fire; in this case, which event fires next is determined by a probabilistic choice. Moreover, DTMCs inherits the *Markovian property*, also known as the *memoryless property*, where the next state after a state transition only depends on the current state.

Probabilistic Computational Tree Logic*(PCTL)* is a variation of the well known CTL formulas where path quantifiers are replaced by a probability operator of the form $\mathcal{P}_{\Diamond p}(\varphi)$, where $\Diamond \in \{\leq, <, \geq, >\}$ is a relational operator, $p \in [0, 1]$ is a probability, and φ is

Algorithm 2.7 *Union* operation on SeqDDNs

```
function UNION( $p$ : SeqDDN, $q$ : SeqDDN)
    declare local SeqDDN $r$
    for $l \in$ LENGTHS($p$) $\cup$ LENGTHS($q$) do
        if $l \notin$ LENGTHS($p$) then    $r[l] \leftarrow q[l]$
        else if $l \notin$ LENGTHS($q$) then $r[l] \leftarrow p[l]$
        else $r[l] \leftarrow$ MDDUNION(l, p[l], q[l])
    UNIQUETABLEINSERT($r$)
    return $r$

function MDDUNION($k$ : lvl, $p$ : Mdd, $q$ : Mdd)
    declare local Mdd $m$                                  ▷ deal with the base cases.
    if $p = 0$ then return $q$
    if $q = 0$ or $q = p$ then return $p$
    if Cache contains $\langle$ Union, $\{p, q\} : m\rangle$ then return $m$
    for $a \in \mathcal{S}$ do
        $m \leftarrow$ MDDUNION(k-1, p[a], q[a])
    $UniqueTableInsert(k, m)$
    $Cache \leftarrow \langle$ Union, $\{p, q\} : m\rangle$
    return $m$
```

a path formula of the form $\phi \lhd^{\Diamond t} \psi$, where $\lhd \in \{X, U, F, G\}$ is a CTL temporal operator and $t \in \mathbb{N} \cup \{\infty\}$ denotes a bound on the number of transitions, so that $t = \infty$ corresponds to unbounded model checking.

In CTL model checking, a witness to an existential formula, or a counterexample to a universal formula, is simply a path in the state space of the system corresponding to finite and legal evolution of the system starting from an initial state. In PCTL (CSL) model checking, however, the system is modeled by a discrete- (continuous)-time Markov chain and a "probabilistic witness" to a formula is a finite set of finite paths such that the sum of their probabilities exceeds some bound. For example, to disprove that the probability of reaching a deadlock is less than 10^{-8}, we need to show enough paths from the initial state to a deadlock state so that their cumulative probability is at least 10^{-8}.

Algorithm 2.8 *Intersection* operation on SeqDDNs

```
function INTERSECTION( $p$ : SeqDDN, $q$ : SeqDDN)
    declare local SeqDDN $r$
    for $l \in$ LENGTHS($p$) $\cap$ LENGTHS($q$) do
        $r[l] \leftarrow$ MDDINTERSECTION(l, p[l], q[l])
    UNIQUETABLEINSERT($r$)
    return $r$

function MDDINTERSECTION($k$ : lvl, $p$ : Mdd, $q$ : Mdd)
    declare local Mdd $m$                          ▷ deal with the base cases.
    if $p = 1$ then return $q$
    if $q = 1$ or $q = p$ then return $p$
    if Cache contains ⟨ Intersection, $\{p, q\} : m$⟩ then return $m$
    for $a \in \mathcal{S}$ do
        $m \leftarrow$ MDDINTERSECTION(k-1, p[a], q[a])
    UNIQUETABLEINSERT($k, m$)
    Cache ← ⟨ Intersection, $\{p, q\} : m$⟩
    return $m$
```

In practice, such a set of paths might be quite large and will usually have paths of different lengths. An experiment conducted by [36], shows that counterexamples can reach double exponential growth in size with respect to the number of input variables. One way to store counterexamples succinctly is via regular expressions [23, 36]. In this case, the proof of correctness is achieved by recursive evaluation of the resulting regular expression to compute its probability. However, converting a counterexample into a minimized regular expression is a tedious process that requires converting the underlying DTMC model into a DFA and incrementally eliminating variables to generate the corresponding regular expression. In fact, the order of variable elimination affects the size of the resulting regular expression and heuristics are needed to select a good ordering that will result in a succinct regular expression.

Now, let us consider how counterexamples are generated in the first place. Aljazzar

Algorithm 2.9 *Difference* operation on SeqDDNs

```
function DIFFERENCE( $p$ : SeqDDN, $q$ : SeqDDN)
    declare local SeqDDN $r$
    for $l \in$ LENGTHS($p$) do
        if $l \notin$ LENGTHS($q$) then $r[l] \leftarrow p[l]$
        else $r[l] \leftarrow$ MDDDIFFERENCE(l, p[l], q[l])
    UNIQUETABLEINSERT($r$)
    return $r$

function MDDDIFFERENCE($k$ : lvl, $p$ : Mdd, $q$ : Mdd)
    declare local Mdd $m$                                  ▷ deal with the base cases
    if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return $p$
    if $p = q$ then return $\mathbf{0}$
    if Cache contains ⟨ Difference, $p, q : m$⟩ then return $m$
    for $a \in \mathcal{S}$ do
        $m \leftarrow$ DIFFERENCE($k-1, p[a], q[a]$)
    UNIQUETABLEINSERT($k, m$)
    Cache ← ⟨ Difference, $p, q : m$⟩
    return $m$
```

et al. in [3, 4] used A.I. techniques such as Best First Search *(BFS)* and *Z**, a specialized directed search algorithm, to incrementally generate a counterexample that consists of the most probable paths. With the same goal of generating a smallest, most expressive counterexample and under the assumption that the states refuting a given property are already known, Han *et al.* [37, 36], showed that the strongest evidences could be generated via a simple single source shortest path algorithm such as Dijkstra's algorithm for unbounded model checking and by using either the Bellman-Ford or Viterbi algorithms for bounded model checking. The strongest evidence is usually not enough to serve as a counterexample. The next step is to construct a smallest counterexample by exploiting a recursive enumeration algorithm for which the number of the needed paths to refute the property is determined on the fly. All the mentioned algorithms are explicit, therefore do not scale well for large

Algorithm 2.10 *Symmetric Difference* operation on SeqDDNs

```
1:  function XOR( $p$ : SeqDDN, $q$ : SeqDDN)
2:      declare local SeqDDN $r$
3:      for $l \in$ LENGTHS($p$) $\cup$ LENGTHS($q$) do
4:          if $l \notin$ LENGTHS($p$) then $r[l] \leftarrow q[l]$
5:          else if $l \notin$ LENGTHS($q$) then $r[l] \leftarrow p[l]$
6:          else $r[l] \leftarrow$ MDDXOR(l, p[l], q[l])
7:      UNIQUETABLEINSERT($r$)
8:      return $r$

1:  function MDDXOR($k$ : lvl, $p$ : Mdd, $q$ : Mdd)
2:      declare local Mdd $m$
3:      if $p = \mathbf{0}$ then return $q$                    ▷ deal with the base cases
4:      if $q = \mathbf{0}$ then return $p$
5:      if $p = q$ then return $\mathbf{0}$
6:      if Cache contains $\langle$ XOR, $\{p, q\} : m\rangle$ then return $m$
7:      for $a \in \mathcal{S}$ do
8:          $m \leftarrow$ MDDXOR($k - 1, p[a], q[a]$)
9:      UNIQUETABLEINSERT($k, m$)
10:     Cache $\leftarrow \langle$ XOR, $\{p, q\} : m\rangle$
11:     return $m$
```

models. The need for a symbolic (e.g., decision-diagram based) approach for probabilistic counterexample generation remains a challenge that we plan to address in future work.

2.5.2 Biological sequence analysis

Indexing

Advancements in sequencing instruments and lower cost associated with sequencing DNA, have resulted in an exponential increase in the amount of sequencing data and the number of genomes stored in public databases. According to [79], genomic databases are doubling in size every 15 to 16 months. Due to the size of these dataset, computation is a bottleneck in the analysis pipeline.

Algorithm 2.11 *Concatenation* operation on SeqDDNs

```
function CONCATENATE( $p$ : SeqDDN, $q$ : SeqDDN)
    declare local $SeqDDN\ r$
    declare local $mdd\ m$
    for $k \in$ LENGTHS($p$) do
        for $l \in$ LENGTHS($q$) do
            $m \leftarrow$ mddCONCATENATE(p[k], q[l])
            $r[k+l] \leftarrow$ mddUNION(k+l, r[k+l], m)
    UNIQUETABLEINSERT($r$)
    return $r$

function mddCONCATENATE($k$ : lvl, $p$ : Mdd, $q$ : Mdd)
    declare local $m$                                   ▷ deal with the base cases
    if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return $\mathbf{0}$
    if $p = \mathbf{1}$ then return $q$
    if $q = \mathbf{1}$ then return $p$
    if $Cache$ contains $\langle$ Concatenate, $p, q : m\rangle$ then return $m$
    for $a \in \mathcal{S}$ do
        $m[a] \leftarrow$ mddCONCATENATE(k-1,p[a],q)
    UNIQUETABLEINSERT($k, m$)
    $Cache \leftarrow \langle$ Concatenate, $p, q : m\rangle$
    return $m$
```

A memory-efficient representation of these dataset that allows for efficient data manipulation is needed. For instance, the *suffix tree* [53] is a memory-efficient data structure in which common prefixes are represented in same paths along the tree. The suffix tree be built in linear time [73], and allows one to answer to queries in time proportional to the size of the pattern. Although the space required by the suffix tree is linear in the size of the text, the number of bytes requires is 20-25 times the size of the input DNA string, making the suffix impractical for large eukaryotic genomes.

Another indexing structure is the *directed acyclic word graph* (*DAWGs*), which can be built online in linear time [10]. A DAWG is the DFA that recognizes the set of all suffixes of a given string. By making all its states accepting, DAWG recognizes the set of all subwords of the encoded string. DAWG achieves similar query time complexity as suffix trees with lower memory cost due to the fact that shared suffixes use common paths in the DAWG. However, this comes at the cost of losing location information. While insertion of a new word into an existing DAWG can be done in linear time in the size of the data structure [67], set manipulation algorithms are not done efficiently. Since DAWGs are DFAs, the result of set manipulation is not guaranteed to be minimal; therefore, an additional minimization step should be performed separately. In general, substring indices data structures lack efficient set manipulation algorithms [24].

Binary decision diagrams are instead designed for efficient set manipulation algorithms. As mentioned earlier in Section 1.3, SeqBDDs inherit BDDs and ZBDDs set manipulation algorithms, yet still have the ability to store any finite language of finite strings; where a sequence is represented as a bit vector, each bit represents an alphabet element per position. This representation requires $\lg |\Sigma|$ boolean variables per position, given $|\Sigma| > 1$.

The authors of [64] adapt the Set Decision Diagrams *(SDD)* introduced in [20], to overcome the drawback of binary representation used by SeqBDDs and achieve more compact storage for large databases of biological sequences. The goal is to maximize similarities between encoded sequences to maximize sharing and minimize branching. This is done by global reordering of each sequence in the set to be encoded. Further reduction is achieved by swapping, merging, and concatenating nodes to reduce the number of nodes and edges in the resulting diagram. To ensure canonicity, these reduction rules are applied iteratively in a predefined order. Their results show a 90% improvement, in the size of their data structure over SeqBDD in terms of the number of nodes. SeqBDDs encode bits, while SDDs encode characters. However, this comparison ignores the number of edges and the size of data associated with each edge. While the number of nodes and edges might be smaller in the proposed data structure, the information associated with the edges is more complicated since it consists of symbols, sequences, or sets. Moreover, since the sequences are reordered, the permutation needs to be stored to recover the original data.

We have previously introduced SeqDDBs and SeqDDNs, which are multi-valued (unlike SeqBDDs) yet still maintain a simpler structure than SDDs. Simple structures promote a more comprehensible development of complex functions. In terms of SeqDDs compactness in regards to sequence indexing, we will start by discussing SeqDDBs. When encoding a set of suffixes or a set of subwords of a string w, the compactness of SeqDDBs is comparable to that of DAWGs. Recall that a DAWG is defined as a minimal *partial* DFA and the size of a SeqDDB, in terms of the number of edges, equals the size of a minimized *partial* DFA plus the number of accepting states. Given the fact that the size of the smallest automaton accepting the set of all suffixes of a string w is linear in the size of w [22]; more specifically, the number of transition is at most $3n - 4$, where $n = |w| > 3$ [21], we can conclude that the size of a SeqDDB encoding w's suffixes is bounded by $4n - 3$, where the number of accepting states is at most $n + 1$. As for the size of a SeqDDB encoding w's subwords, Blumer *et al.* proved in [10] that a partial minimized DFA recognizing the set of all subwords consists of $2n - 2$ states and $3n - 4$ transitions; therefore, the size of a SeqDDB encoding w's subwords equals $5n - 6$, given that all states are accepting. In the case of encoding a set of prefixes, the size equals $2n$ (refer to the example in Figure 2.10(a)).

Encoded set	SeqDDB size	SeqDDN size
Suffixes	$4n-3$	$2n+1$
Subwords	$5n-6$	$\frac{1}{6}(n^3+3n^2+8n+6)$
Prefixes	$2n$	n^2+1

Table 2.1: Summary of the upper bound size of a SeqDDB/N encoding a set of all prefixes, suffixes, or subwords of a certain string of size n.

On the other hand, the size of a SeqDDN encoding a set of suffixes is $2n + 1$, where the SeqDDN will consist of a MDD, with one node per level, of size n and $n+1$ handles pointing to the corresponding suffix (refer to an example in Figure 2.10(b)). A SeqDDN encoding a set of prefixes is of size up to $n^2 + 1$, while the size of a SeqDDN encoding the set of substrings, assuming no nodes are shared, equals $n + 1 + \sum_{j=1}^{n} j(n - j + 1)$, which simplifies to $\frac{1}{6}(n^3 + 3n^2 + 8n + 6)$. In practice, the size is often smaller due to suffix sharing (Table 2.1 shows a summary of these results).

Using SeqDDBs and SeqDDNs for indexing sequences allows for efficient set manipulations. Moreover, the membership problem can be solved in a time proportional to the size of the query. Future work will employ edge-valued SeqDDs to preserve information about substring locations.

Sequence alignment

In molecular biology, similar DNA or protein sequences tend to carry the same function. Sequence similarity allows one to detect homologies and to predict the functionality of novel genes or protein sequences. There are three kinds of sequence alignments: global, local, and semi-global.

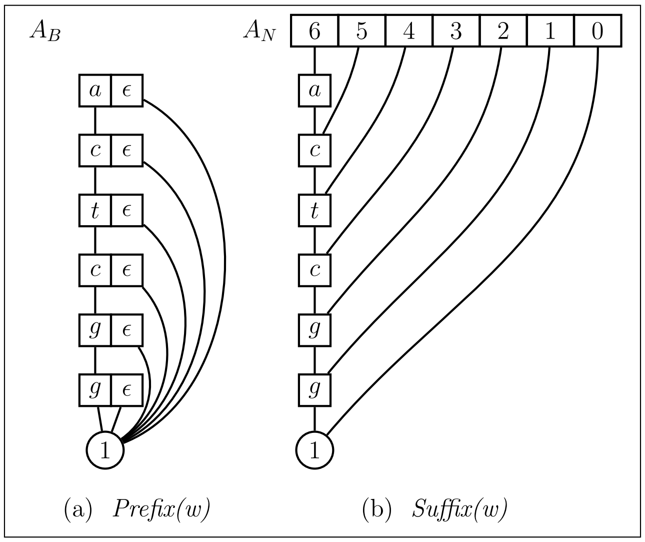


Figure 2.10: Example shows a SeqDDB encoding the set of all prefixes and a SeqDDN encoding the set of all suffixes of w = "*actcgg*".

An alignment between two sequences is formed by inserting gaps, such that the two sequences becomes of the same length. The similarity between two aligned sequences is measured by the number of matches, mismatches, and gaps. A global alignment between two sequences aims produce an alignment with the highest similarity score. A local alignment between two sequences is a pair of aligned substrings with the highest similarity score among all other substring pairs of the two sequences. A semi-global alignment is a variation of global alignment that do not penalize gaps at the end of any of the two sequences.

Global alignment is used to check if two sequences are entirely homologous, i.e., entirely aligned. Local alignment is used to discover conserved regions. Semi-global alignment is usually used in the context of shotgun genome assembly, where the ends of the sequences are matched.

The alignment between two sequences is called pairwise alignment; if it is carried out among multiple sequences, it is called multiple sequence alignment [38, 79]. Next, we show how we can take advantage of the SeqBDDs and their variants to solve two sequence alignment problems.

First, we consider the case of a pairwise local/semi-global alignment under the assumption that there is a single gap in the pattern that is known a priori. Given a query of the form "$s{*}v$", where $*$ stands for zero or more extra characters, and a SeqBDD encoding a set of sequences, the single wild card query method proposed in [5] can answer such a query in time linear in the size of the query. The algorithm returns the intersection of sequences having prefix s with the reverse of the sequences having a prefix v-reversed. However, the algorithm does not take into account the time and memory complexity associated with creating a reversed SeqBDD. This can be done efficiently by incrementally constructing a reversed SeqBDD in time linear to the size of the original SeqBDD by visiting each node in the topological ordering of the nodes. Since their fast method to build the reversed SeqBDD requires an intermediate SeqBDD (representing visited paths) attached to each node of the original SeqBDD, the memory requirement could be prohibitive for large SeqBDDs.

The more general case of multiple local sequence alignment is related to the *frequent subsequence mining* problem addressed by [49]. In this chapter, where SeqBDD were first introduced, a weighted variation was required to accomplish the mining process. The purpose is to mine subsequences appear at frequency exceed a predefined minimum support. Given a weighted SeqBDD p encoding arbitrary set of finite strings, they construct x-conditional databases, each as a SeqBDD, exploiting decision diagrams techniques, such as a unique table to share nodes among different SeqBDDs and operation cache for efficient manipulation.

Biclustering for gene-expression analysis

Conventional clustering approaches compose coherent clusters of objects that are grouped according to their weights regarding some attributes. In biclustering techniques, however, objects and attributes are symmetric and the goal shift to clustering them simultaneously [35]. Biclustering gene expression aims to identify groups of genes that exhibit similar reactions to different stimuli [82].

We consider the *ZCluster* algorithm [83], which uses symbolic manipulation to discover all biclusters in a given microarray matrix without the need for exhaustive enumeration, thus, coping with the computational challenges of an NP-hard problem [82, 17]. The *ZCluster* algorithm inherits the *pScore* system from the *PCluster* algorithm to score sub-matrices and generate pairwise maximal biclusters, which are divided into two types: *horizonal seeds* for every two genes to show a maximal set of experiments to which they responded similarly, and *vertical seeds*, analogously, for every two experiment conditions. Considering that the number of experiments is much smaller than the number of genes, usually 10^3 to 10^4 genes in a microarray and fewer than 100 experiments [17], *ZCluster* starts with generating the *vertical seeds* and represents them as ZBDDs, then generates the corresponding *horizontal seeds* represented as a *trie*, to generate the final biclusters.

In [82], Yoon *et al.* represented both vertical and horizontal seeds as ZBDDs. In their follow-up paper [83] they represented horizontal seeds as a set of strings of different length and encode them using *trie*. As future work, our goal is to explore the benefits of storing both vertical and horizontal seeds as SeqDDs, and compare the *ZCluster* algorithm efficiency with different combinations of representations.

All-pair suffix-prefix overlap

Detecting suffix prefix overlap is a vital step in genome assembly, especially for third generation sequencing where reads are long (but noisy). According to [34, 33], the *all-pair suffix-prefix overlap* problem is defined as follows.

Definition 8 *Given two strings S_i and S_j, any suffix of S_i that matches a prefix of S_j is called a suffix-prefix match of S_i, S_j. Given a set of strings $= \{S_1, S_2, \cdots, S_k\}$, all-pair suffix-prefix problem is the problem of finding, for each ordered pair (S_i, S_j), the longest suffix-prefix match.*

To find all-pair suffix-prefix overlaps in DNA sequences using SeqDDB, we build two shared SeqDDBs such that for a given finite set of DNA sequences $\mathcal{S} = \{s_1, s_2, \cdots, s_k\}$, let $\hat{\mathcal{S}}$ be a set composed of reverse complements $\forall s_i \in \mathcal{S}$. And let u and v be two canonical shared SeqDDBs, where u encodes $\mathcal{S} \cup \hat{\mathcal{S}}$ and v encodes $\mathit{suffix}_{\geq\tau}(\mathcal{S})$. And let p and q be two canonical SeqDDBs, where p encodes $s_i \cup \hat{s_i}$ and q encodes $\mathit{suffix}_{\geq\tau}(s_i)$, for all $s_i \in \mathcal{S}$. Algorithm 2.12 introduces a set operation SUFFIX_PREFIX_OVERLAP to find all pair suffix-prefix overlaps of length $\geq \tau$.

The algorithm was tested on a set of simulated reads that for chromosome 1 of *Saccharomyces cerevisiae* (yeast) genome (which is approximately 230 kbp). Simulated reads were generated using ART [40], which generated 3,068 reads of length 150 bp each (about 2x sequencing depth). For $\tau = 33$, the result contained 1,992 overlaps.

To verify that detected overlaps are indeed the longest overlap, we conducted the following test; assume that an overlap of length y is detected between a pair of sequences $(s_i,\, s_j)$, where $|s_i| = x$ and $|s_j| = z$, then the test follows one of the three cases below.

$$\textsc{longest overlap} = \begin{cases} \mathit{true} & \text{if } |\mathit{overlap}| = \mathit{min}(|s_i|, |s_j|), \\ \mathit{true} & \text{if } \mathit{suffix}_{=z+1}(s_j) \neq \mathit{prefix}(s_i), \\ \mathit{false} & \text{otherwise} \end{cases}$$

Note that our test detects false positives, while false negatives are not detected. Running the test on out dataset shows that all detected overlaps are the longest.

2.6 Conclusion

We introduced SeqDDs, a multi-valued sequence decision diagrams, which can be perceived as MDDs with no variable ordering but are still, nevertheless, canonical. In our setting the notion of levels is not applicable, hence our representation is not sensitive to variable ordering, therefore the "size explosion" depends merely on the encoded set. SeqDDs are ideal for encoding a finite set of strings of arbitrary lengths. To ensure canonicity, we proposed two canonical versions, with ϵ restricted towards the bottom or with ϵ restricted towards the top. The latter version is analogous to a shared MDD, which we adapt into what we called SeqDDN. The compactness of our representations were studied in relation to finite automata. The results showed that there is no winner between the two versions; therefore, we proposed algorithms for both SeqDDBs and SeqDDNs. SeqDDs are useful for applications that require compact storage and efficient manipulation of large sets of strings with high sharing rate.

Algorithm 2.12 Suffix-Prefix overlap

```
1: function allPairSuffixPrefixOverlap( $u$ : shared SeqDDB, $v$ : shared SeqDDB)
2:     declare local int $len$
3:     for $s_j \in \mathcal{S}$ do
4:         $q \leftarrow v[s-j]$        ▷ SeqDDB q points to the root of SeqDDB encoding $suffix_{\geq_\tau} s_j$
5:         for $s_i \in \mathcal{S}$ do
6:             if $s_i = s_j$ then
7:                 continue
8:             $p \leftarrow u[s-i]$        ▷ SeqDDB p point to the root of SeqDDB encoding $s_i \cup \hat{s_i}$
9:             $len \leftarrow 0$
10:            $len \leftarrow$ SuffixPrefixOverlap($p$,$q$, $len$)
11:            if $len > 0$ then
12:                output overlap info
13:    return

1: function SuffixPrefixOverlap( $p$ : SeqDDB, $q$ : SeqDDB, $len$ : int)
2:     declare local SeqDDB $r$
3:     declare local int $count$
4:     if $p$=0 or $q$=0 then return 0        ▷ base case: empty set
5:     if $q$=1 then return 1
6:     if $p$=1 then        ▷ base case:$\epsilon$
7:         if $q[\epsilon]$=1 then return 1
8:         else return 0
9:     if $Cache$ contains $\langle SefPrefOverlap, (p,q){:}r\rangle$ then return $r$        ▷ check if already computed
10:    $count \leftarrow 0$        ▷ initialize counter
11:    for $a \in \Sigma$ do        ▷ Compute by recursively call SefPrefOverlap for each $a \in \Sigma$
12:        $r[a] \leftarrow$ SefPrefOverlap($p[a]$, $q[a]$)
13:        if $r[a]$=0 then $count \leftarrow count + 1$        ▷ count edges pointing to terminal 0
14:    $len \leftarrow len + 1$
15:    if $q[\epsilon] = 1$ and ( $p[\epsilon] = 1$ or $\forall_{a\in\Sigma}\ p[a] \neq 0 \Leftrightarrow q[a] = 0$) then
16:        if $count = |\Sigma|$ then return 1        ▷ $\epsilon$-node
17:        else $r[\epsilon] = 1$
18:    else if $count = |\Sigma|$ then $r \leftarrow 0$    $len \leftarrow 0$        ▷ empty-node
19:    $UniqueTableInsert(r)$        ▷ insert to unique table to ensure canonicity
20:    $Cache \leftarrow \langle\ SefPrefOverlap, (p,q){:}r\rangle$        ▷ cache result to avoid re-computation
21:    return $r$
```

Chapter 3

A Comparative Evaluation of Assembly Reconciliation Tools

While the number of sequenced genome keeps increasing, the majority of eukaryotic genomes are unfinished due to the algorithmic challenges of assembling them. A variety of assembly and scaffolding tools are available, but it is not always obvious which tool or parameters to use for a specific genome size and complexity. As a consequence, it is common practice to produce multiple assemblies using different assemblers/parameters, then select the best one for public release. A more compelling approach would allow one to merge multiple assemblies with the intent to produce a higher quality *consensus* assembly, which is the objective of *assembly reconciliation.*

Several assembly reconciliation tools have been proposed in the literature, but their strengths and weaknesses have never been compared on a common dataset. We fill this need with the work presented in this chapter, in which we report on an extensive comparative evaluation of CISA, GAA, GAM_NGS, GARM, Metassembler, MIX, and ZORRO. Specifically, we evaluate contiguity, correctness, coverage, and duplication ratio of the merged assembly compared to the individual assemblies provided in input.

None of the tools we tested consistently improved the quality of the input GAGE and synthetic assemblies. Our experiments show an increase in contiguity in the consensus assembly only if the original assemblies already have high quality. In terms of correctness, the quality of the results depends on the specific tool, as well as on the quality and the ranking of the input assemblies. In general, the number of misassemblies range from being comparable to the best of the input assembly to being comparable to the worst of the input assembly.

3.1 Background

Despite the prodigious throughput of the sequencing instruments currently on the market, the assembly problem remains very challenging, mainly due to the repetitive content of large genomes, uneven sequencing coverage, and the presence of (non-uniform) sequencing errors and chimeric reads. The third generation of sequencing technology, e.g., Pacific Biosciences [27] and Oxford Nanopore [19], offers very long reads at a higher cost per base, but sequencing error rate is much higher.

A significant number of *de novo* genome assemblers are available to the community. The choice of the most appropriate assembler depends on the size and complexity (repeat content, ploidy, etc.) of the genome to be assembled, the type of sequencing technology used to produce the input reads (e.g., Sanger, 454, Illumina, PacBio, Nanopore, etc.), and the availability of paired-end or long-insert mate-pair reads. Each assembler implements slightly different heuristics to deal with repetitions in the genome, uneven coverage, sequencing errors and chimeric reads. The final assembly is very rarely entirely finished, with one solid sequence per chromosome. Instead, the typical output is an unordered/unoriented set of contiguous regions called *contigs*. If paired-end or mate-pair reads are available, some of contigs can be ordered and oriented by anchoring paired-end reads to contigs. In some cases, the length of the gaps between contigs can be estimated and contigs can be joined together to create *scaffolds*.

As said, selecting which assembler to use in order to produce the best quality assembly is not a trivial task. Assembly competitions such as Genome Assembly Gold-Standard Evaluation (GAGE) [66] and Assemblathon [13] have been held to evaluate multiple assemblers on common data sets. Such comparative evaluations can provide general guidelines, but there is no systematic way to determine which assembler and what parameters settings to use to produce the "best" assembly for a specific genome and a specific dataset. As a consequence, it is common practice to generate multiple genome assemblies from a few different assemblers and/or parameters (e.g., the k-mer size for the de Bruijn graph), and then try to guess the "best" assembly based on assembly statistics, spot-checking, homology analysis, etc.

In fact, the notion of "best" assembly is not well defined. Since it is unlikely to obtain a "perfect" assembly that covers the entire genome with no assembly errors, one has to decide whether it is more important to maximize contig/scaffold length (at the expense of possibly introducing more mis-assemblies) or minimize the number of mis-assemblies (at the expense of possibly generating shorter contigs/scaffolds). Typically, the quality assessment for draft assemblies is carried out via statistical measurements and alignment to a reference genome (if one is available). N50 is a widely used metrics to assess the contiguity of an assembly, which is defined by the length of the shortest contig for which longer and equal length contigs cover at least 50% of the assembly. NG50 is similar to N50 except the metrics relates to the genome size rather than the assembly size. NA50 and NGA50 are analogous to N50 and NG50 where the contigs are replaced by blocks that can be aligned to the reference. Correctness is measured by detecting misassemblies such as mismatches, indels, and misjoins. Misjoins are considered the least desirable type of misassemblies [72], where loci that are far apart in the genome are improperly joined in the assembly. Misjoins include inversions, relocations, and translocations. An *inversion* occurs when the orientation of a contig is inverted with respect to the reference. A *relocation* occurs when a contig is misplaced within the chromosome it belongs to, and a *translocation* occurs when a contig is misplaced in a different chromosome.

Assembly reconciliation algorithms attempt to take one step further towards a finished genome. Rather than arbitrarily try to guess the best assemblies among several draft assemblies, assembly reconciliation tools offer a compelling alternative. These tools promise to produce a higher quality *consensus* assembly by merging two or more draft assemblies. The main goal of assembly reconciliation algorithms is to enhance contiguity of the resulting assembly while at the same time, avoid introducing assembly errors. In this chapter, we carry out the first comprehensive evaluation of assembly reconciliation tools by measuring the quality of the consensus assembly on several common input datasets with different quality attributes.

3.1.1 Assembly reconciliation tools

The concept of assembly reconciliation was first introduced by Zimin *et al.* [88]. In that work, the authors also introduced an assembly reconciliation tool called Reconciliator, which is no longer maintained (last updated in 2007). Other reconciliation tools in the literature that are no longer maintained and/or have no documentation were excluded from our evaluation. We also excluded GAM, because it was superseded by GAM_NGS. Other tools such as eRGA [74], MAIA [62], and Minimus2 [71] were also not included in our comparative evaluation because these tools address different problems. Reference-guided assembly (eRGA and MAIA) and hybrid assembly (Minimus2) are related to the problem of assembly reconciliation, but not quite the same. The former uses a closely related reference to assemble the conserved regions of the genome, which reduces the complexity of *de novo* assembly to the non-conserved portions. Hybrid assembly allows users to incorporate reads from different sequencing technologies (e.g., short Illumina reads with long PacBio reads). MAIA has also the ability to merge *de novo* assemblies if several closely related reference genomes are available. QuickMerge [16] is a tool that allows users to merge an assembly obtained from Pacific Bioscience reads with another assembly based on second generation reads. We excluded QuickMerge from our evaluations due the lack of publicly available PacBio-based assemblies with a corresponding high quality reference genome that would allow us to assess the results.

In this work we benchmarked seven assembly reconciliation tools, namely CISA, GAA, GAM_NGS, GARM, Metassembler, MIX, and ZORRO, which are briefly described next. Table 3.1 summarizes the main goals and features of the seven assembly reconciliation tools evaluated in this study. Several of these algorithms take advantage of *compression-expansion* (CE) statistic, which allows them to detect assembly compression (due to an incorrect deletion) or assembly expansion (due to an incorrect insertion) [88]. In order to obtain the CE statistics, paired-end or mate-pair reads are mapped to the assembly to be evaluated. The CE statistics is computed by comparing the distance between the mapped mates and the expected insert size.

The objective of CISA is to reconcile bacterial genome assemblies [48]. Given the contigs for each of the input draft assemblies, CISA selects *representative* contigs (i.e., longest contigs) and discards (nearly) contained contigs. CISA then tries to extend representative contigs, and detects mis-assembly in the representative contigs by aligning them to query contigs. Contigs that align to multiple positions are considered misassembled and another representative contig is selected. Contig with an unaligned portion are split. Finally, the resulting contigs are iteratively merged. We should note that CISA's objective is to merge more than two assemblies, but we have also tested it on two inputs for consistency with other tools.

Users of GAA have to specify a target and a query assembly [81] where the "target" assembly is expected to be of higher quality. The objective of GAA is to close gaps in target assemblies using the query assembly. Query contigs that are not anchored to at least two contigs target are not utilized.

The input to GAM_NGS is one or more alignments between each library of reads and each assembly [75]. GAM_NGS first identifies maximal portions of both input assembly (called *blocks*) that share the same set of uniquely mapped reads. GAM_NGS then constructs a weighted undirected graph where each vertex corresponds to a contig, and an edge connects two contigs if (i) they belong to different assemblies and (ii) they share at least one block. From this graph, GAM_NGS computes a consistent ordering and orientation of blocks with respect to both input assemblies. Then, GAM_NGS builds another directed weighted graph (called *assembly graph*) where each vertex represents a block, and each edge connects two blocks if they belong to the same contig of at least one of the assemblies. After resolving conflicts in the *assembly graph*, GAM_NGS computes a semi-global alignment between any two contigs that share at least one block. If two contigs have at least 95% identity, GAM_NGS "merges" the assemblies by selecting the assembly with the better compression-expansion statistics.

GARM [55] also manipulates assemblies asymmetrically, but users do not need to know in advance which one is the better assembly. The tool decides which one is the "reference" assembly based on a variety of assembly statistics. GARM then (i) aligns the assemblies to each other to detect overlaps (using *nucmer* [45]), (ii) removes ambiguous overlaps and contigs which are (nearly) completely contained in each another, (iii) generates layout and consensus scores, (iv) merges contigs, (v) orders merged contigs to match the order and the orientation of the original scaffolds (if scaffolds are available) – if a contig that is a part of a scaffold is not merged, the contig is placed within the resulting scaffold in a location that corresponds to the original scaffold and the gap length is recomputed.

Compression-expansion statistics on the two input assemblies are also used in Metassembler [78]. First, Metassembler uses *nucmer* [45] to align the two input assemblies;

the boundaries of these alignments are called *break points*. For each region between the break points, one of two assemblies is selected based on its compression-expansion statistics. Metassembler allows users to input more than two assemblies, but merges them in an progressive pairwise fashion.

MIX [72] uses a directed weighted graph called *extension graph* which is annotated with a variety of weights to represent prefix-suffix overlaps between contigs in the input assemblies. MIX determines a set of non-overlapping *maximal independent longest paths* on the extension graph to merge contigs. Contigs not included in any path are examined for duplications, contigs that are contained or nearly contained are removed, and the rest are added to the assembly. MIX does not performs error correction, but rather focuses on enhancing contiguity.

ZORRO [6] starts by masking repetitive regions which are identified using k-mer statistics. Once the repetitive regions are masked, the overlap between the two assemblies is detected using Minimus [71]. ZORRO then unmasks the repetitive regions and merges the overlapping contigs. Lastly, ZORRO uses the tool Bambus [63] to order and orient contigs using paired-end reads.

3.2 Datasets and Experimental Results

Since the quality of the input assemblies is expected to directly affect the quality of the final merged assembly, we explored the performance of assembly reconciliation tools under different input quality.

To carry out a comparative evaluation of the seven assembly reconciliation tools

Table 3.1: Features of the assembly reconciliation tools evaluated in this study.

	CISA	GAA	GAM_NGS	GARM	Metassembler	MIX	ZORRO
Inputs							
Contigs allowed	✓	✓	✓[a]	✓	✓	✓	✓
Scaffolds allowed	✓	✓[b]	✓[a]	✓	✓		
Short reads allowed			✓[a]				
Paired-end reads allowed			✓[a]				✓
Mate-pair reads allowed			✓[a]		✓		
Alignments allowed		✓	✓				
Reads required			✓[a]		✓		✓
Reference input assembly required			✓		✓		
Input assemblies treaded symmetrically	✓					✓	
Only two input assemblies		✓	✓	✓	✓[c]		✓
More than two input assemblies	✓					✓	
Can handle bacterial/small genomes	✓	✓	✓	✓	✓	✓	✓
Can handle large eukaryotic genomes		✓	✓	✓	✓		✓
Goals							
To increase assembly contiguity	✓	✓	✓	✓	✓	✓	✓
To decrease number of assembly errors	✓		✓				
Methods							
Compression-expansion statistics		✓	✓		✓		
Scaffolding information		✓		✓			
Use single reads			✓				
Use paired-end/mate-pair reads		✓	✓		✓		✓
Can split assembly misjoin	✓	✓					
Can detect/avoid repetitive regions	✓		✓				✓
Output							
Contigs	✓	✓	✓	✓	✓	✓	✓
Scaffolds				✓[d]	✓		

[a]Optional, GAM_NGS requires alignment file.
[b]Scaffolds should be broken into contigs. A gap file and contig naming conveys scaffolding information
[c]performs iterative pairwise
[d]when input contains scaffolds

listed above, we used publicly-available assemblies for the GAGE competition [66] and we created synthetic assemblies of *Saccharomyces cerevisiae S288c* [6] including structural variants. The choice of the GAGE assemblies was motivated by the fact that this dataset has been the most commonly used for assembly reconciliation tools. The authors of GAM_NGS used this dataset in their experimental results, CISA was tested on assemblies of *Staphylococcus aureus* and *Rhodobacter sphaeroides*, and MIX used GAGE_B [52] which includes the assemblies of *Staphylococcus aureus* and *Rhodobacter sphaeroides*. Other assembly reconciliation tools used the Assemblathon dataset [13], which was a similar assembly competition to GAGE.

All assembly reconciliation tools were ran with default parameters, unless otherwise noted. We explored how other parameter settings affected the experimental results in section 3.2.6. Since some assembly reconciliation tools can take advantage of scaffold information, we carried out experiments on both contig-based assemblies and scaffold-based assemblies.

Outputs of assembly reconciliation tools were processed by our scripts, then fed into Quast [32] (GAGE option activated) to obtain assembly statistics. Quality scores were also computed using Quast on the input assemblies. We first collected assembly statistics related to contiguity, namely N50, number of contigs, longest contig, and total assembly size. By comparing the assemblies to the reference genome we also collected NGA50, number of misassemblies, the total length of contigs affected by misassemblies, the number of mismatches and indels between the assembly and the reference, the percentage of the reference genome covered by the assembly, and the duplication ratio. In addition to genome-wide analyses, we also studied the ability of these tools to assemble the primary sequence of annotated genes. Specifically, we computed the fraction of each gene sequence covered by contigs, for both input and merged assemblies. Details about the procedure used to compute gene coverage can be found in Subsection 3.2.4. A complete report on these statistics is reported in Tables A.1–A.18. Here, we only summarize the results using a graphical representation of the contiguity/correctness tradeoff (see Figures 3.2-3.7). Input and output assemblies are represented as points on the scatter plot where the x-coordinate

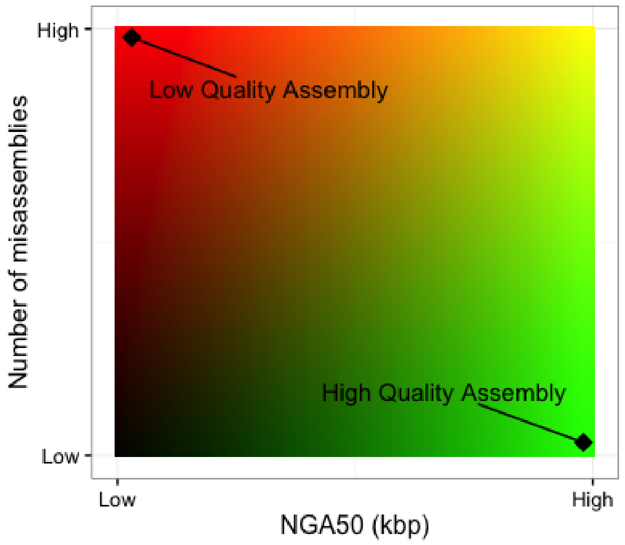


Figure 3.1: The performance of assembly reconciliation algorithms is summarized as points on a 2D scatter-plot, in which the x-axis represents contiguity (NGA50) and the y-axis represents the number of misassemblies.

represents the contiguity (NGA50), and their y-coordinate is the number of misassemblies. Figure 3.1 illustrates how to interpret the plots. We want assembly reconciliation tools to "move" the input points towards the bottom right corner of the plot, i.e., increase the contiguity and reduce the number of assembly errors.

All experiments were performed on a Linux Ubuntu 12.10 server with a 20 cores Intel Xeon CPU E5-2690v2 3GHz and 512GB of RAM. Multithreading was used when available. A detailed analysis of run time, memory consumption, CPU utilization for all the tools and genomes is reported in Subsection 3.2.7. A companion website `http://reconciliation.cs.ucr.edu/` provides links to the all the datasets and the scripts used in this study.

3.2.1 GAGE assemblies

The GAGE competition evaluated eight assemblers (ABySS [70], ALLPATHS-LG [29], Bambus2 [63], Celera Assembler [60, 57], MaSuRCA [87], SGA [69], SOAPdenovo [47], and Velvet [85]) on whole-genome shotgun sequence data of four genomes, namely *Staphylococcus aureus* (genome size ≈2.8 Mbp), *Rhodobacter sphaeroides* (≈4.6 Mbp), *Homo sapiens*' chromosome 14 (≈88 Mbp), and *Bombus impatiens* (≈250 Mbp). *Staphylococcus aureus* has one main chromosome and a small plasmid, while *Rhodobacter sphaeroides* has two chromosomes and five plasmids. In our experiments we mainly used the first hree genomes, because at the time of writing *Bombus impatiens* did not have a high quality reference genome. We only used the assemblies for *Bombus impatiens* to determine which tools would be able to handle large inputs. Out of the 4×8 genome-assembler pairs, the GAGE competition included 27 assemblies (available from `http://gage.cbcb.umd.edu`).

Running each assembly reconciliation tool on all pairs of assemblies (out of the 27 available) would generate several hundred merged assemblies and it would be difficult to draw general conclusions. We decided instead to select input assembly pairs based on six different criteria and compare the results on the selected pairs. To streamline the presentation, we will not comment on tools that did not run successfully.

3.2.2 Limitations

Here are some practical limitations related to the execution of benchmarked tools. MIX and CISA: we did not run these two tools on the *Hg_chr14* dataset because they were designed for bacteria-sized genome and they would not handle such a large input.

GARM: while GARM's manual claims that the tool can accept two contigs, two scaffolds, or contig/scaffold combination as an input, we were only successful to run the tool using one contig and one scaffold; in most cases, running with two contigs produced an empty FASTA file, while using two scaffolds produced FASTA files with all nucleotides set to N.

3.2.3 Usage of reads

Some of the tools can take advantage of the raw reads, in addition to the input assemblies. For GAA, while the paper mentions using paired-end reads for error correction, there is no option to provide them. Therefore, we didn't use them for GAA. We used reads in the following cases. For GAM_NGS, we used paired-end reads with a 155-180bp insert (Library 1 in GAGE). For Metassembler, in the case of bacterial genomes we used the available short-jump library (insert size of 3500bp); for *Hg_chr14* we used the available long-jump library (insert size is approximately 35 kbp), and for *Bombus impatiens* we used the available short-jump library 2 (insert size is approximately 8 kbp). For ZORRO, we used paired-end reads with a 155-180bp insert (Library 1 in GAGE)

3.2.4 Gene coverage analysis

We used the following reference genomes and their corresponding gene annotations

- *Staphylococcus aureus* subsp. aureus USA300_TCH1516 found at `http://bacteria.ensembl.org/staphylococcus_aureus_subsp_aureus_usa300_tch1516/Info/Index` (2844 Genes)
- *Rhodobacter sphaeroides* KD131

`http://bacteria.ensembl.org/rhodobacter_sphaeroides_kd131/Info/Index` (4474 Genes)

- *Homo sapiens, chromosome 14* GRCh38.p2

  `http://uswest.ensembl.org/Homo_sapiens/Info/Index` (ftp release 80) (2289 Genes)

First, we created a BLAST database for each of the GAGE reference genome assemblies and each of the merged output assemblies. Then, we used BLASTn to align the primary sequence of each gene against each database (using default parameters). For each hit reported in BLASTn output, we chose the best ranked alignment with 75% minimum identity. The total gene coverage reported is the cumulative sum of the coverage of each hit minus any overlaps between the hits.

3.2.5 Experimental results

High contiguity, high correctness inputs (GAGE)

In the first set of experiments, the objective was to explore the contiguity/correctness tradeoff. Specifically, we wanted to test the ability of reconciliation tools to take advantage of the contiguity of the first input assembly and the correctness of the second in order to create a merged assembly with a number of misassemblies comparable to the second assembly and a contiguity comparable to the first assembly. The two input assemblies to be merged were chosen so that one has high N50 value (but possibly a relatively high number of misassembly errors) and the other has few misassembly errors (and possibly a lower N50).

Figure 3.2 and Table A.1 reports the results of merging the SOAPdenovo assembly (high N50) with the ABySS assembly (low misassembly errors) for the three chosen

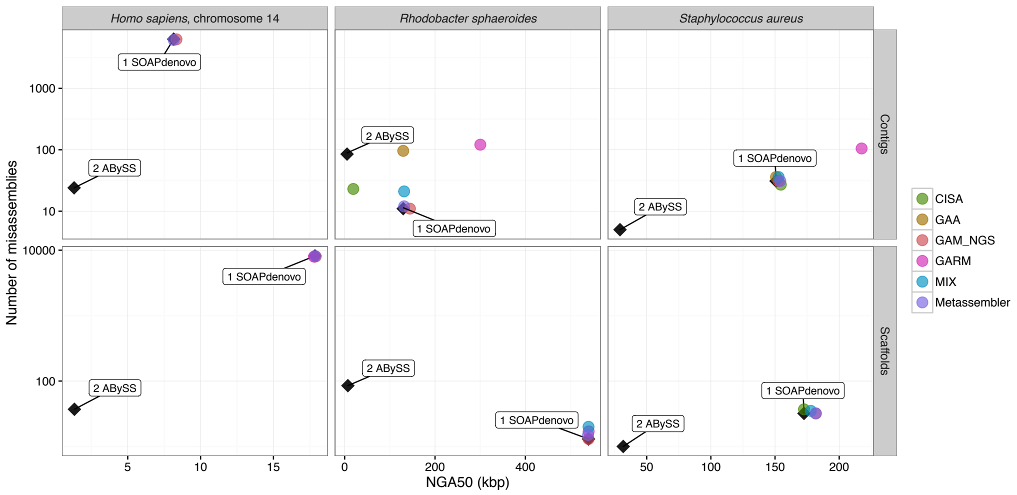


Figure 3.2: Contiguity-correctness experimental results when inputs are contigs (top row) or scaffolds (bottom row); assembly reconciliation tools are given two assembled genomes to merge (*Homo sapiens, chromosome 14*, *Rhodobacter sphaeroides*, *Staphylococcus aureus*), in which the first assembly has high contiguity, the second has high correctness; tools were ran using default parameters

genomes. Since the assembly produced by ABySS on the *Rhodobacter sphaeroides* genome has more misassembly errors than the assembly generated by SOAPdenovo, we also reported in Table A.5 the results produced by ALLPATHS-LG and SGA on *Rhodobacter sphaeroides* assemblies. The SOAPdenovo assembly was used as the "master" assembly in all tools that require a ranking of the inputs.

Observe in Figure 3.2 that on the *Staphylococcus aureus* genome, all tools increase the contiguity marginally (in fact, by less than 3%). While none of the tools was able to improve assembly errors compared to the ABySS assembly, GAA and MIX produced more errors than SOAPdenovo. CISA produced the lowest number of misassemblies (13% less than SOAPdenovo). Otherwise, GAM_NGS and Metassembler maintained quality statistics close to that of SOAPdenovo.

GAA created a merged assembly in which number of misassemblies was very close to the sum of those statistics for the input assemblies. In terms of NGA50 the contiguity was at least as good as the most contiguous input assembly.

When the input was composed of scaffolds (bottom panel in Figure 3.2), all tools improved contiguity marginally (in fact, by less than 5%). Table A.1 show that GARM's and MIX's merged assemblies covered less than 50% of the reference sequence. None of the tools was able to reduce the number of misassembly errors compared to ABySS; in fact, CISA produced more errors than SOAPdenovo.

Despite the fact that ABySS's assembly for *Rhodobacter sphaeroides* had a higher number of misassembly errors than SOAPdenovo, none of the merged assemblies improved on the number of misassemblies compared to SOAPdenovo. Except for GAA, the number of misassembly errors produced by all tools were closer to the master (SOAPdenovo). As expected, tools that rely on a master assembly had a lower number of misassemblies than those that did not rank the inputs. With scaffolds as inputs, changes in NGA50 were negligible for all tools except for CISA. With contigs as inputs, GAM_NGS improved the contiguity by at most by 11%, Metassembler and MIX increased it by 2%, and CISA dropped it by 85%. MIX and Metassembler, and GARM maintained the same NGA50 as SOAPdenovo.

In the majority of the cases, experimental results obtained with ALLPATHS-LG (high N50) and SGA (low misassembly errors) on the *Rhodobacter sphaeroides* genome (reported in Appendix A Table A.5) followed similar patterns to the ones we observed in Figure 3.2. CISA decreased the contiguity (although the reduction was far less this time). GAA followed the same general pattern mentioned earlier. GAM_NGS did not increase contiguity but rather maintained it close to that of the master assembly. Metassembler and MIX also did not increase contiguity. ZORRO worked for this experiment: although it decreased contiguity by 10%, it produced a smaller number of misassembly errors than ALLPATHS-LG (but still higher that SGA).

With scaffolds as input assemblies, GAM_NGS retained the quality statistics of the master assembly. Observe in Figure 3.2 that GARM retained NGA50 close to SOAPdenovo (the master assembly). Also note that in Table A.5 that GARM maintained ALLPATHS-LG's contiguity statistics.

Experimental results on the *Hg_chr14* with contigs as input assemblies (Figure 3.2), show that (i) GAM_NGS slightly improved contiguity, (ii) Metassembler maintained contiguity, (iii) GAA crashed, (iv) the number of misassemblies was closer to SOAPdenovo. With scaffolds as inputs, GAM_NGS and Metassembler produced assemblies with quality statistics close to SOAPdenovo.

Reordering the inputs (GAGE)

As mentioned above, some of the assembly reconciliation tools assume that the first input assembly is the master assembly, and should be trusted more (we call these tools *asymmetric*). The goal of this set of experiments is determine how the quality of the merged assembly depends on the specific order of the inputs.

To determine how the ranking affected the results, we repeated the same experiments reported in the previous section but switched the order of the inputs. A comparative analysis of Figure 3.3 and Table A.2 with the results discussed in the previous section

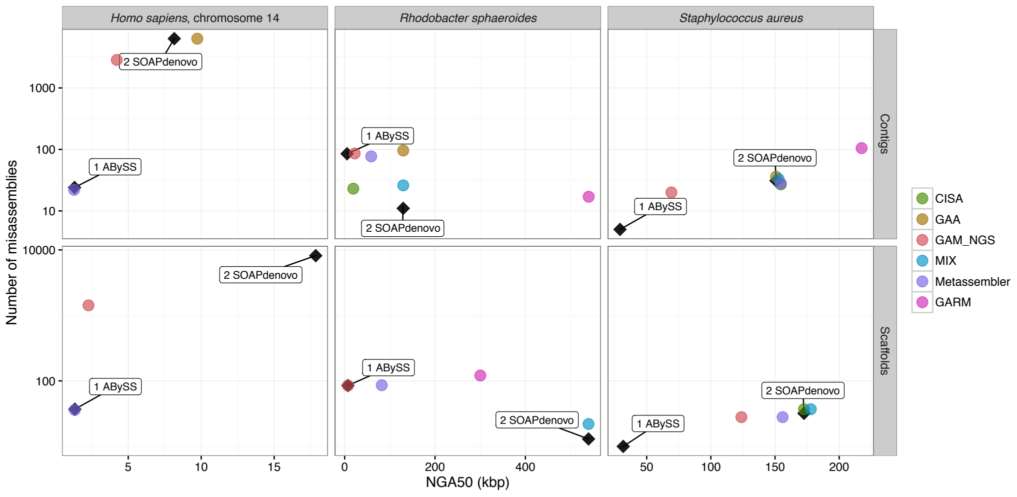


Figure 3.3: Contiguity-correctness experimental results when inputs are contigs (top row) or scaffolds (bottom row); compared to Figure 3.2, the order of the inputs was swapped.

prompts a few observations. First, we note that CISA, MIX, and GARM are *symmetric* (i.e., they do not require users to rank the inputs, see Table 3.1), hence they are expected to be unaffected by the reordering. Experimental results confirm that CISA and GARM are indeed unaffected. The reordering however affected MIX results, albeit only slightly.

For *Staphylococcus aureus*, MIX's contiguity statistics (N50 and NGA50) was not affected by the reordering of the inputs. However, we observed a small change in the number of misassemblies, although still higher than SOAPdenovo in both cases.

On *Rhodobacter sphaeroides*, all statistics remained unchanged except for the number of misassemblies that increased after reordering. In addition, with contigs as inputs we did not observe an increase in NGA50 after the reordering.

Despite the fact that GAA requires input ranking, the results for *Staphylococcus aureus* and *Rhodobacter sphaeroides* were similar. The output statistics of GAA followed the general pattern mentioned in the previous section. For *Hg_chr14*, GAA crashed in one ordering but not on the other. For all three genome, GAM_NGS and Metassembler produced consensus assemblies with quality statistics close to the master assembly.

Note that the merged assemblies have higher contiguity in Figure 3.2, in which the master has higher N50. In contrast, the number of misassemblies were lower in Figure 3.3 for both *Staphylococcus aureus* and *Hg_chr14* in which the master had lower errors (with the exception of MIX). Merged assemblies for *Rhodobacter sphaeroides* had higher contiguity and lower number of misassemblies, in which the master had higher N50 and lower number of misassemblies (see Figure 3.2).

High-quality inputs (GAGE)

In the third set of experiments we tested the ability of the reconciliation tools to merge two high quality assemblies. We selected two highly contiguous assemblies (i.e., small number of contigs and scaffolds, high N50 values) and low number of misassembly errors. Figure 3.4 and Table A.3 show the result of merging assemblies produced by ALLPATHS-LG as first input and either MSR-CA, SOAPdenovo, or CABOG as the second assembly.

Observe that for *Staphylococcus aureus* with contigs as inputs, GAM_NGS produced an improved assembly that had no misassemblies, and was 66% more contiguous. The next best assembly was by Metassembler with a 107% increase, but it had a slight increase in the number of misassemblies compared to ALLPATHS-LG. MIX produced a high number of misassemblies (higher than MSR-CA) but managed to increase contiguity by 4%. CISA improved contiguity by 11%, but it produced a number of errors higher than ALLPATHS-LG. ZORRO decreased contiguity by 30%.

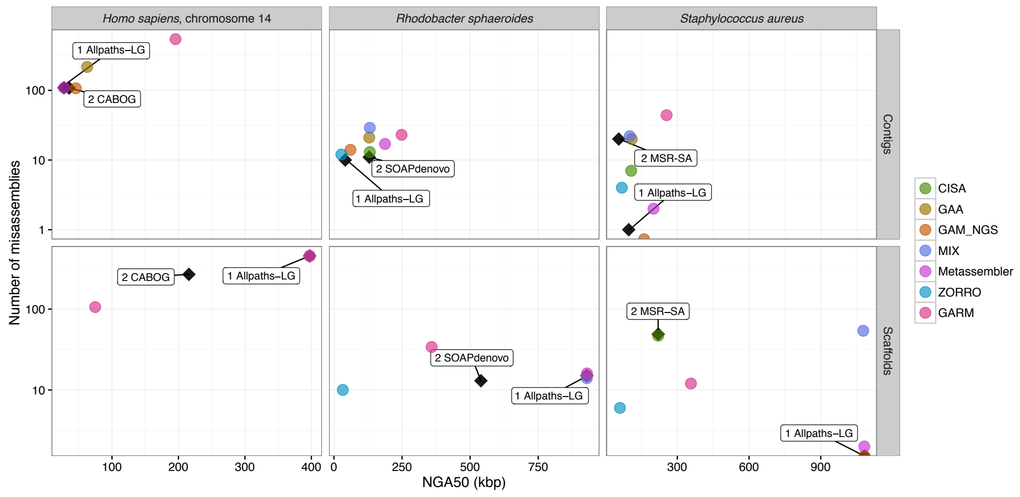


Figure 3.4: Experimental results on merging high-quality assemblies (top row for input contigs, bottom row for input scaffolds); tools were ran using default parameters

With scaffolds as inputs, ALLPATHS-LG has no misassemblies and higher NGA50 than MSR-CA. In general, asymmetric tools produced a lower number of misassemblies and decreased the N50. For instance, GAM_NGS maintained quality statistics of ALLPATHS-LG. Although ZORRO is asymmetric it decreased contiguity by more than 90%. On the other hand, symmetric tools had a higher number of misassemblies. GARM achieved the highest increase of NGA50 (16%).

The contiguity of the merged assemblies improved $11\% - 108\%$ with the exception of ZORRO, which decreased the contiguity by 30%. GARM increased contiguity the most (108%) at the expense of a number of misassemblies close to MSR-CA. MIX introduced no misassemblies, but covered only 25% of the genome sequence. Notably, both GAM_NGS and Metassembler improved contiguity by 66.5% and introduced no misassemblies, These are two rare examples in which we observed an unquestionable improvement in the merged assembly.

On the *Rhodobacter sphaeroides* genome, the two input assemblies had almost the same number of misassemblies but the assembly produced by SOAPdenovo was much less fragmented. Only Metassembler increasing NGA50 significantly. All other tools decreased the contiguity. In terms of correctness, ZORRO and CISA (using scaffolds as inputs) reduced the number of misassemblies but also decreased the contiguity by 99% and 60%, respectively. Other tools produced merged assemblies with a number of misassemblies not better than the inputs.

GARM improved the contiguity by 38% while CISA increased it by less than 2%. MIX is the only tool that reduced the number of misassemblies, but again its assembly only covered about half of the genome. None of the tools improved both contiguity and the number of misassemblies.

In *Hg_chr14*, GAA improved the NGA50 by 76%, but it produced a number of misassemblies equal to the sum of the number of misassemblies in the two inputs. GAM_NGS improved the contiguity (28% increase in NGA50) and slightly reduced the number of misassemblies. Metassembler produced quality statistics that are very close to ALLPATHS-LG.

With scaffolds as inputs, GAM_NGS and Metassembler maintained similar quality statistics to ALLPATHS-LG. GARM decreased NGA50 by 9%. It also increased the number of misassemblies.

Highly-fragmented inputs (GAGE)

The goal of this set of experiments was to evaluate the performance of assembly reconciliation tools when provided with two highly fragmented input assemblies. Input assemblies were selected to have a high percentage of contigs shorter than 200 bps, a high

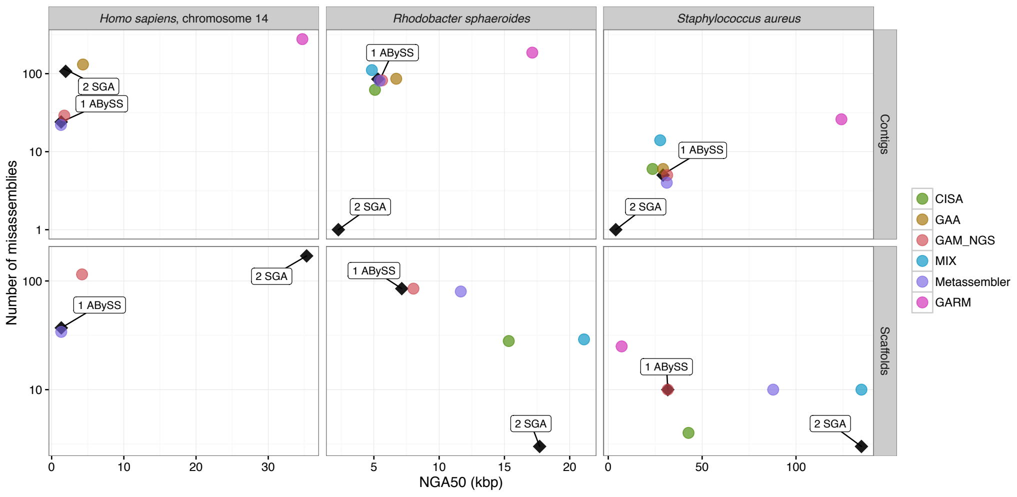


Figure 3.5: Experimental results on merging highly fragmented assemblies (top row for input contigs, bottom row for input scaffolds); tools were ran using default parameters

number of contigs and scaffolds, and low N50.

Figure 3.5 and Table A.4 shows the results of merging ABySS assembly and SGA assembly. Observe that when we used contigs as inputs, ABySS had a higher contiguity than SGA (except in *Hg_chr14*). The opposite, however, was observed when scaffolds were provided in input. In *Staphylococcus aureus* and *Rhodobacter sphaeroides* with contigs as inputs, only asymmetric tools maintained or improved over NGA50 of the better input assembly (in *Staphylococcus aureus* we observed up to 8% increase, and up to 17% in *Rhodobacter sphaeroides*). However, in *Hg_chr14* (with contigs as inputs) GAA produced a 123% increase over SGA, while GAM_NGS did not improve NGA50 over SGA, but it increased it 33% over ABySS.

With scaffolds as inputs, we observed a decrease in NGA50 except for MIX and GARM (when SGA inputs are scaffolds). MIX, GARM, and CISA are symmetric tools, hence they are expected to perform better than other tools when the non-master input has

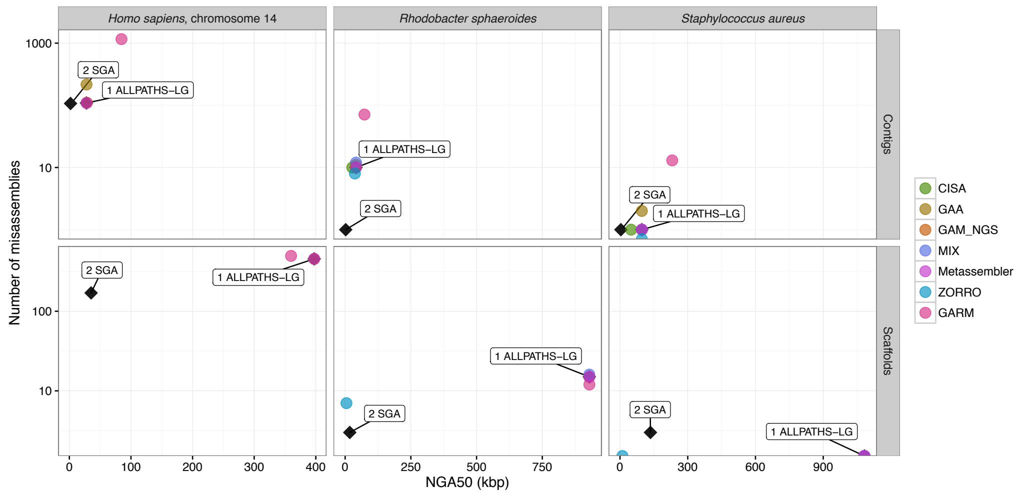


Figure 3.6: Experimental results on merging assemblies produced by assemblers based on the de Bruijn graph compared to string graph (top row for input contigs, bottom row for input scaffolds); tools were ran using default parameters

better quality. CISA, however, produced inferior results with scaffolds as inputs in most experiments. We discovered that CISA with default parameters break scaffolds into contigs when a scaffold contains more than ten consecutive Ns. MIX maintained NGA50 of SGA, while GARM slightly decreased it compared to SGA (yet still higher than ABySS).

De Bruijn vs. string graph assembly (GAGE)

Here we tested the effect of merging assemblies generated using different assembly strategies. Specifically, we merged an assembly generated by an assembler that uses a de Bruijn graphs with an assembly produced by an assembler based on the string graph. Figure 3.6 and Table A.5 shows the result of merging an assembly produced by ALLPATHS-LG (based on the de Bruijn graph) with an assembly produced by SGA (based on the string graph). Overall, GAM_NGS, Metassembler, and MIX maintained similar assembly statistics as ALLPATHS-LG.

Note that *Staphylococcus aureus* input assemblies (as contigs) had only one misassembly. The merged assemblies also have one misassembly, with the exception of GAA (two) and ZORRO (none). ZORRO corrected the assembly error without affecting NGA50. CISA decreased NGA50 by 49%. With scaffolds as inputs, ALLPATHS-LG's assembly has no assembly errors. In fact, observe that all merged assemblies did not have any misassemblies. GARM kept NGA50 close to ALLPATHS-LG. CISA covered less than 40% of the genome, while ZORRO decreased the contiguity by 99%.

On *Rhodobacter sphaeroides* with contigs as inputs, CISA and ZORRO decreased the contiguity by 34% and 10%, respectively. GAM_NGS and Metassembler maintained ALLPATHS-LG's quality statistics. All tools produced a relatively high number of misassemblies (similar to ALLPATHS-LG). With scaffolds as inputs, CISA, ZORRO, and GARM's assembly statistics followed the same of statistics of *Staphylococcus aureus*. All assemblies, with the exception of CISA and ZORRO, had a number of misassemblies closer to ALLPATHS-LG. CISA again covered less than one fifth of the genome and ZORRO decreased the contiguity by 99%. GAM_NGS, Metassembler, and MIX produced consensus assemblies with quality statistics comparable to ALLPATHS-LG.

In *Hg_chr14* (with contigs as inputs) GAM_NGS increased NGA50 by 2%. With scaffolds as inputs, GAM_NGS and Metassembler maintained assembly statistics close to ALLPATHS-LG. GARM increased the number of misassemblies by 9% (compared to ALLPATHS-LG) and decreased NGA50 by 9%.

With scaffolds as inputs, GARM increased contiguity by 58%, while other tools improved it by less than 3%. GAM_NGS and Metassembler produced about the same number of misassembly errors as the higher of the two inputs. GARM improved NGA50 the most, but also increased the number of misassemblies by 42%.

Multiple inputs (GAGE)

In this set of experiments we tested the ability of the tools to merge more than two assemblies. When an assembly reconciliation tool allowed no more than two assemblies in input (see Table 3.1 for a list), we merged them in an iterative fashion. For instance, to merge three assemblies, we first merged two assemblies, then merged the result to the third assemblies. Metassembler uses a similar strategy: when the user provides multiple assemblies the tool iteratively performs pairwise reconciliation, where the output of one iteration is the input of the next. We ordered the input assemblies based on *feature response* curve (FR curve), which is an assembly quality metric proposed in [61]. The FR curve represents the dependency between contigs that contains no more than τ features and the corresponding genome coverage. The x-axis represents τ and the y-axis represent genome coverage: the "steeper" is the curve, the better is the assembly. We used the FR curves in [75] to determine the merging order of the GAGE assemblies, starting with the assemblies with highest quality. Results for an alternative ordering is discussed in Note A.1.7 and corresponding Tables A.12-A.15. For tools that allowed to merge more than two assemblies (e.g., CISA and MIX), the merging was done in one step from the original assemblies. Here we were interested in measuring the contiguity and correctness of the resulting assemblies as the number of input assemblies increases.

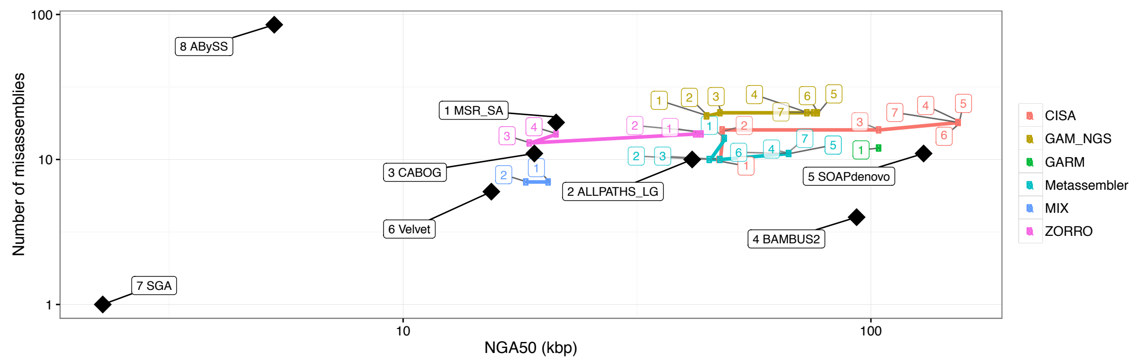


Figure 3.7: Experimental results on merging multiple assemblies of *Staphylococcus aureus*(black diamonds); the input order was determined using the FRCurve score (see text for details); integer labels indicates successive merging steps; tools were ran using default parameters

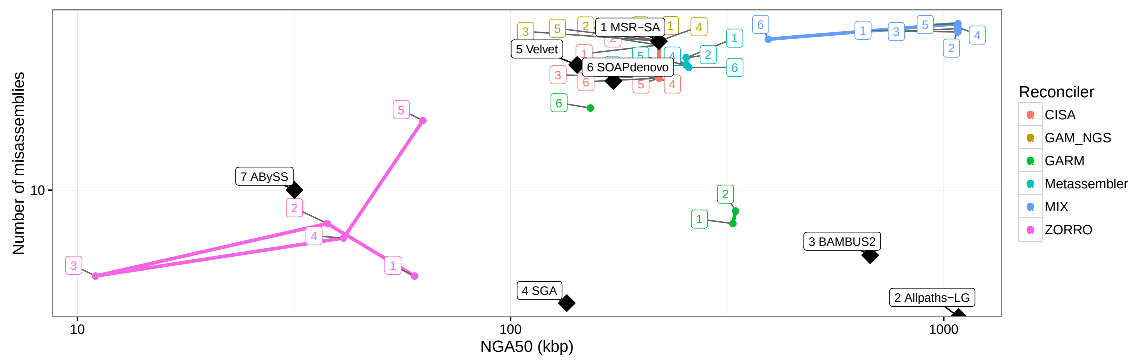


Figure 3.8: Experimental results on merging more than two assemblies (as scaffolds) ordered by the FRCurve score (*Staphylococcus aureus*, genome size 2,903,081 bp). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

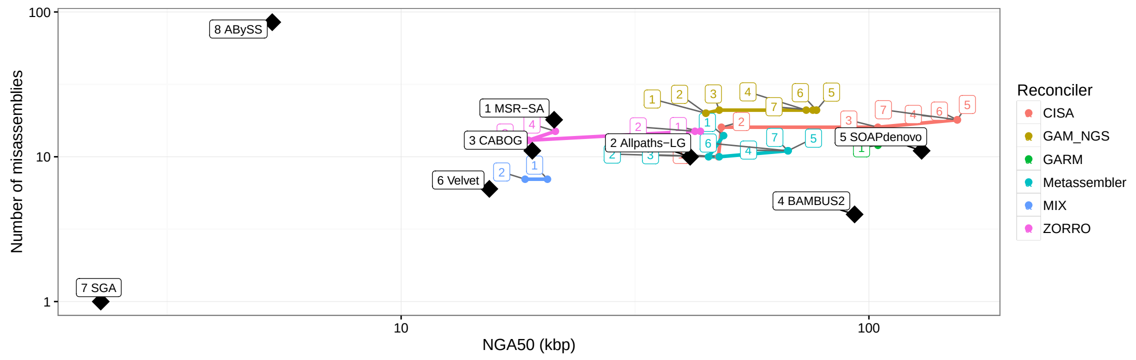


Figure 3.9: Experimental results on merging more than two assemblies (as contigs) ordered by the FRCurve score (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

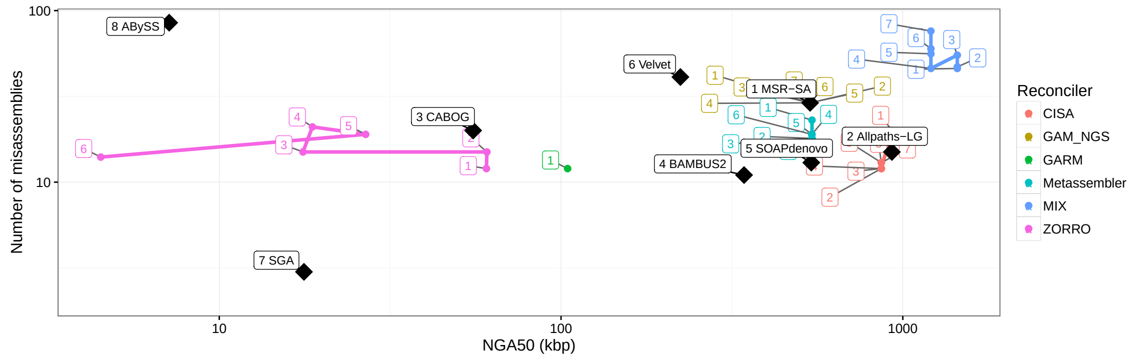


Figure 3.10: Experimental results on merging more than two assemblies (as scaffolds) ordered by the FRCurve score (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

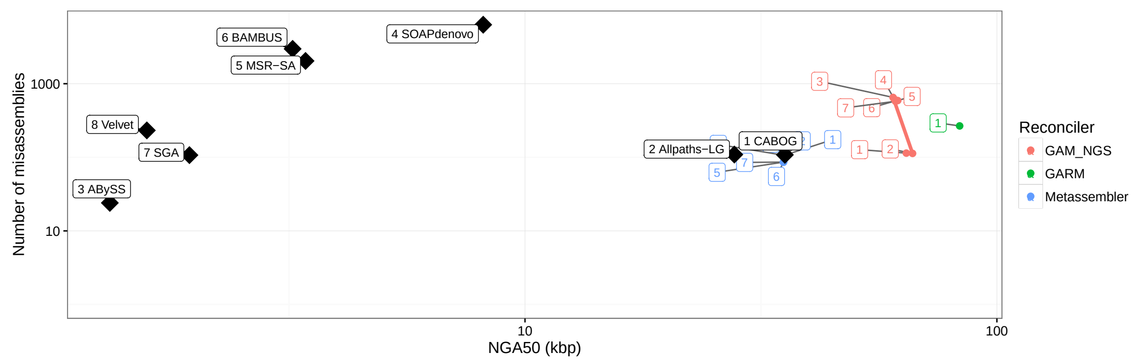


Figure 3.11: Experimental results on merging more than two assemblies (as contigs) ordered by the FRCurve score (*Hg_chr14*, genome size 107,349,540 bp). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

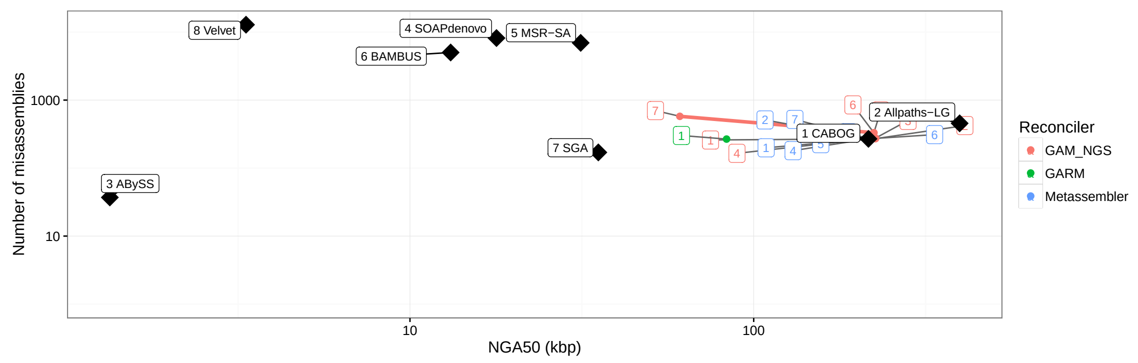


Figure 3.12: Experimental results on merging more than two assemblies (as scaffolds) ordered by the FRCurve score (*Hg_chr14*, genome size: 107,349,540). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

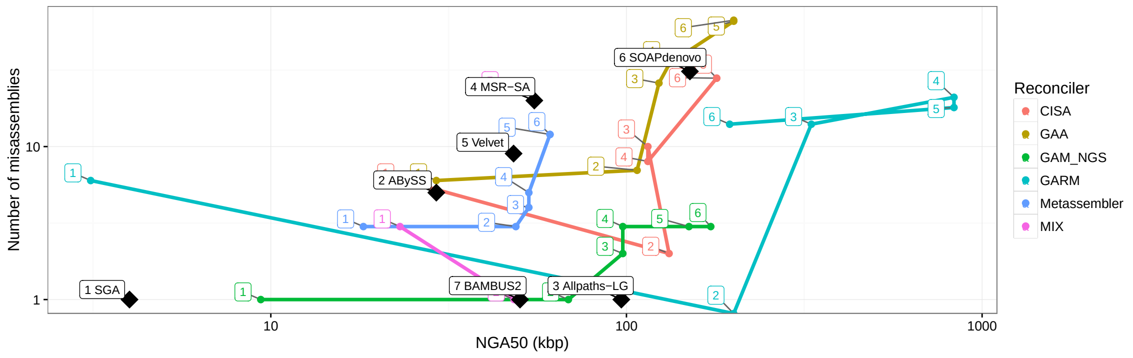


Figure 3.13: Experimental results on merging more than two assemblies (as contigs) – alternative ordering (*Staphylococcus aureus*, genome size 2,903,081 bp). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

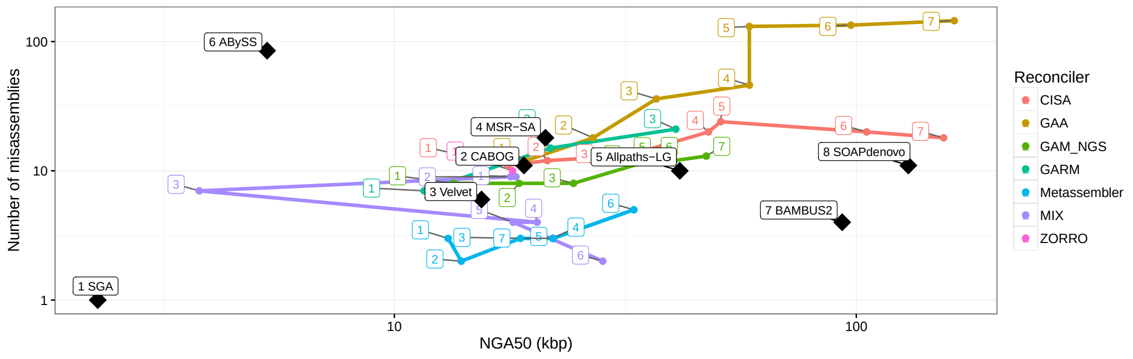


Figure 3.14: Experimental results on merging more than two assemblies (as contigs) – alternative ordering (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

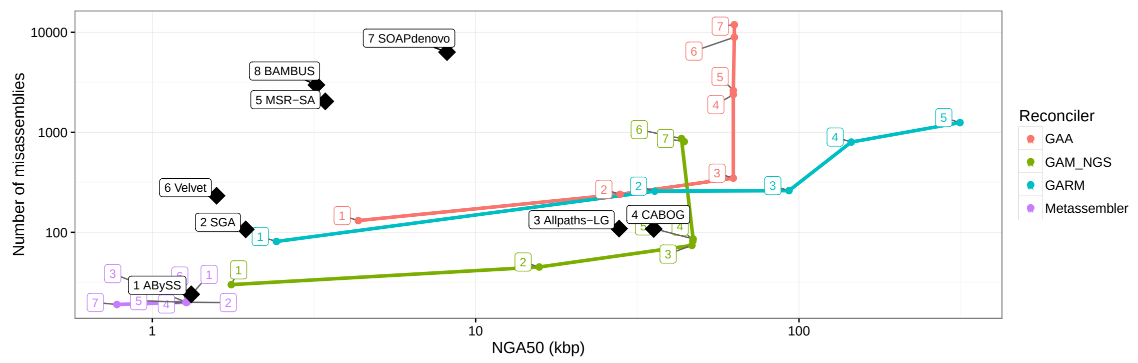


Figure 3.15: Experimental results on merging more than two assemblies (as contigs) – alternative ordering (*Hg_chr14*, genome size 107,349,540 bp). The Figure reports on quality of merged assembly compared to the input assemblies. Tools were ran using default parameters, unless otherwise noted

Figure 3.7, Figure 3.9, and Figure 3.11 show the experimental results for *Staphylococcus aureus*, *Rhodobacter sphaeroides* and *Hg_chr14*, respectively, when inputs are contigs. First observe that in several cases, the process of iterative merging did not complete.

On *Staphylococcus aureus* and *Rhodobacter sphaeroides*, CISA generally improved the contiguity as the number of merged assemblies increased. The number of errors fluctuated over the iterations. GAA did not produce assembly files for the first iteration. Although GAA did not work for this particular ordering it did produce results for the alternative ordering reported in Appendix A Note A.1.7

In *Staphylococcus aureus* and *Rhodobacter sphaeroides*, GAM_NGS's contiguity improved over successive iterations, but the number of misassemblies errors did not decrease (it stayed close to the first master input in all iterations). For *Hg_chr14*, the number of misassemblies was also relatively high. GAM_NGS increased NGA50 by at least 70% compared to CABOG.

In *Staphylococcus aureus*, Metassembler's contiguity improved over successive iterations, but the number of misassemblies also increased. In *Rhodobacter sphaeroides*, Metassembler's assembly did not improve after the forth iteration. Note that NGA50 was lower than BAMBUS2 and SOAPdenovo. Metassembler's assembly had number of misassemblies about the average of the inputs. In *Hg_chr14*, the number of misassembly errors were low and decreased over successive iterations. Contiguity was high, but slightly decreased over successive iterations.

MIX maintained a low number of misassemblies in most iterations but suffered from relatively poor NGA50. Since the genome coverage in most iterations was less than 50% of the reference, no NGA50 was reported for those iteration. On the *Staphylococcus aureus* genome, the coverage was less than 50% in all iterations but it steadily improved with increasing number of inputs. On *Rhodobacter sphaeroides*, the genome coverage was below 50% with four or more inputs.

ZORRO frequently failed to produce results. When it worked, contiguity usually started high, then fluctuated over successive iterations. ZORRO produced relatively high number of misassemblies (somewhat in between the values of the inputs).

We repeated the same experiment but with scaffolds as inputs. Results are reported in Tables A.9, A.10, and A.11 and Figures 3.8, 3.10, and 3.12. CISA's results show that after a certain number of input assemblies, increasing the number of inputs did not affect the results significantly. From that point forward, it generally improved the contiguity and reduced the number of contigs as the number of merged assemblies increased. The number of misassemblies were with the range of input assemblies. CISA reached stability with four inputs on *Staphylococcus aureus* and three inputs on *Rhodobacter sphaeroides*).

MIX on *Staphylococcus aureus*, produced a high number of misassemblies which generally increased as the number of inputs increased. It maintained high genome coverage. It also maintained high contiguity except for the last iteration. On *Rhodobacter sphaeroides*, the number of misassemblies were also relatively high but it fluctuated as the number of inputs increased. It also maintained high contiguity, achieving the best NGA50 for less than five inputs.

ZORRO produced low number of misassemblies on *Staphylococcus aureus* and *Rhodobacter sphaeroides.* Contiguity was poor and generally decreased over successive iterations.

GAM_NGS maintained results very close to the first input throughout all iterations on *Staphylococcus aureus*, *Rhodobacter sphaeroides*, and *Hg_chr14*. In the latter genome, GAM_NGS contiguity generally improved in successive iterations but so did the number of misassemblies.

Metassembler maintained similar quality statistics to CABOG on *Hg_chr14*. On *Rhodobacter sphaeroides*, Metassembler also maintained CABOG's quality statistics with a slight decrease of number of misassemblies, and contiguity as the number of iteration increased. On *Staphylococcus aureus*, Metassembler also maintained quality statistics close but not identical to MSR-CA. In general, as the number of inputs increased, the number of misassemblies slightly decreased and the contiguity slightly improved.

Table 3.2: Experiments on the *Bombus impatiens* assemblies. Notes: all reported statistics are for contigs; tools were ran using default parameters, unless otherwise noted; the E-Size is defined as the expected scaffold size of any arbitrary location in the reference genome.

Reconciliation Tool or Input	Contigs #	N50 (bp)	E-Size (bp)
Input 1 (CABOG)	18,918	23,515	34,227.94
Input 2 (MSR-CA)	18,501	32,431	46,890.24
GAA	Did not produce an assembly file		
GAM_NGS	10,129	52,123	77,240.76
GARM (ctg_scf)	10,572	70,577	98,189.44
Metassembler	17,694	25,317	36,774.11
Input 1 (ABySS)	35,112	14,383	20,904.98
Input 2 (SOAPdenovo)	11,556	57,117	78,228.65
GAA	46,668	63,941	99,133.63
GAM_NGS	Did not produce an assembly file		
GARM (ctg_scf)	9477	64,172	86,881.27
Metassembler	34,149	13,842	20,386.78
Input 1 (SOAPdenovo)	11,556	57,117	78,228.65
Input 2 (ABySS)	35,112	14,383	20,904.98
GAA	46,660	63,941	99,133.42
GAM_NGS	10,971	63,152	87,930
GARM (ctg_scf)	8042	101,115	133,528.41
Metassembler	9349	57,238	78,395.6
Input 1 (MSR-CA)	18,501	32,431	46,890.24
Input 2 (SOAPdenovo)	11,556	57,117	78,228.65
GAA	Did not produce an assembly file		
GAM_NGS	12,559	59,549	89,960.46
GARM (ctg_scf)	5984	117,986	148,549.55
Metassembler	16,234	35,077	50,156.59

Large genomes (GAGE)

To test the ability of these tools to scale to large eukaryotic genomes, we used GAGE's assemblies for *Bombus impatiens*. We selected the two input assemblies where most of the tools were able to complete. A high quality reference genome is unavailable for *Bombus impatiens*, so the statistics we reported were produced by the GAGE script. In addition to the usual assembly statistics, GAGE computes the *e-size*, which is the expected size of a contig (or scaffold). The e-size if computed as $\sum_c L_c^2/G$, where the sum is over all contigs c, G is the expected genome length and L_c is the length of contig c [66].

Results are reported in Table 3.2, in which only contigs and scaffolds of 500bp or longer were considered. Observe that GARM reduced the number of contigs, increased N50 and the e-size for all experiments. GAM_NGS did not work for one of the experiments. In the others, it decreased the number of contigs in all but one experiment. GAM_NGS always improved N50, and increased the e-size in all but one experiment. GAA did not work for two of the experiments. When it worked, it did not reduce the number of contigs, but it increased both N50 and the e-size. Lastly, Metassembler decreased N50 and the e-size in three out of four experiments. Metassembler reduced the number of contigs in half of the experiments.

3.2.6 Parameter tuning

For the results in Appendix A Note A.1, all assembly reconciliation tools were ran with default parameters. Here we explored how other parameter settings affected the experimental results. Each tool has its own set of parameters, as briefly described next.

- CISA has three main parameter namely the minimum contig cutoff, the maximum number of consecutive N's, and the maximum unaligned gap (default values 100bp, 10bp, 0.95 quintile, respectively); we changed the minimum contig cutoff to 200bp and 500bp and the maximum gap size to 100 and 200; we also tried scaffolds as inputs.
- For GAA we focused on two parameters, namely the minimum contig cutoff and the maximum tip size (default values of 100bp and 90 bp, respectively); we changed the contig cutoff size to 200bp and 500 bp and the maximum tip size allowed to 15 bp and 50bp.
- GAM_NGS has three main parameters, namely the minimum number of reads to build a block, the block coverage filtering, and the minimum block length; for these

parameters the authors suggest using 10bp, 0.75, 200bp, respectively for bacterial genomes, and 50bp, 0.75, 500bp for *Hg_chr14*; since there was no option to change the minimum block size, we explored the other two parameters; we used the default values of at least 50 reads per block and 0.75 block coverage filter; we also tried setting the read support to 10 and 30 with 0.75 block coverage filter, as well as read support of 10 and 50 with 0.95 block coverage filter.

- GARM: we explored changing the minimum contig cutoff (default 200bp), minimum overlap (default 200bp) and maximum tip thresholds (default 50bp); in addition to the default values, we tried the following combinations (i) 500bp contig cutoff, 200bp min overlap, and 50bp max tip, (ii) 200bp contig cutoff, 100 bp min overlap, and 50 bp max tip (iii) 200bp contig cutoff, 200bp min overlap, and 100bp max tip, and (iv) 100bp contig cutoff, 50bp min overlap, and 15bp max tip.

- MIX's main parameters are the minimum length of alignment (default 0bp) and minimum contig cutoff (default 500bp); according to the documentation, if these two parameters are not specified MIX is supposed to check thresholds from 0 to 2000 with step of 50; we tried this option, but only got results with default settings; the author of MIX recommend a minimum alignment of 500bp and a minimum contig cutoff of 0bp for bacterial genomes (which is what we used); in addition we tried (i) minAlign=50 and minctg=100, (ii) minAlign=50 and minctg=200, and (iii) minAlign=100 and minctg=500.

- Metassembler has several parameters. We explored the parameters controlling the assembly merge phase, namely minimum read coverage (default 15), minimum overlap (default 60) and identity (default 60); Metassembler accepts identity in base pairs; we tested (i) 60 bp min overlap, 51 bp 85% identity, and 15x coverage (ii) 100bp min overlap, 95% identity, and 15x coverage (iii) 100bp min overlap, 85% identity, and 30x coverage and (iv) 200bp min overlap, 170bp 85% identity, and 30x coverage.

- ZORRO's parameters include the minimum overlap (default 40bp), the maximum tip (default 15), and identity threshold (default 94%); in addition to the default values we tested (i) 40bp min overlap, 100bp max tip, and 94% identity, (ii) 100bp min overlap, 15bp max tip, and 94% identity, (iii) 100bp min overlap, 15bp max tip, and 85% identity, (iv) 100bp min overlap, 50bp max tip, and 85% identity.

Experimental results for all these parameter sets are reported in Appendix A Table A.16, Appendix A Table A.17 and Appendix A Table A.18 for *Staphylococcus aureus*, *Rhodobacter sphaeroides* and *Hg_chr14*, respectively. For most experiments, the variations due to changing parameters were small. Few observations are in order.

Observe that for *Staphylococcus aureus*, Metassembler and GAM_NGS maintained the same statistics for all parameters configurations, with the exception of a slight variation in the size of the assembly. CISA produced changes only when the minimum contig cutoff increased to 500 bp, with contigs as input. In this case, both genome and gene coverage improved but the contiguity decreased with respect to other configurations. With scaffolds as inputs, the contiguity increased but the genome fraction was lower than 50% in most cases. In GARM we observed a small variation in the number of mismatches and indels and an insignificant change in the genome coverage.

3.2.7 Time and Space Analysis

As said, all experiments were performed on a Linux Ubuntu 12.10 server with a 20 cores Intel Xeon CPU E5-2690v2 3GHz and 512GB of RAM. Multithreading was used when available.

First, we measured the usage of computational resources to merge two input assemblies. Graphs in Figure 3.16 illustrate the average (wall clock) run time, the average percentage of processor utilization (where 100% indicates full utilization of one core), and the average memory usage required by each tool to perform each experiments on the four genomes. The average are over all the tested pairs of GAGE assemblies for that genome. Error bars indicate the minimum and maximum.

Second, we measured the usage of computational resources as a function of the number of input assemblies using CISA and MIX, which are the only tools that can merge more than two input assemblies. Graphs in Figure 3.17 shows the (wall clock) run time, processor utilization (where 100% indicates full utilization of one core), and memory usage as the number of input assemblies increases.

3.2.8 Synthetic assemblies

In this set of experiments we tested assembly reconciliation tools on synthetic assemblies generated using RSVSim [8]. We used RSVSim to introduce specific structural variations into the reference genome of *Saccharomyces cerevisiae* [6]. For tools that required reads, we generated synthetic reads using ART [40]. The output of the seven assembly reconciliation tools was fed into Decipher [80]. Decipher detects synteny blocks between a

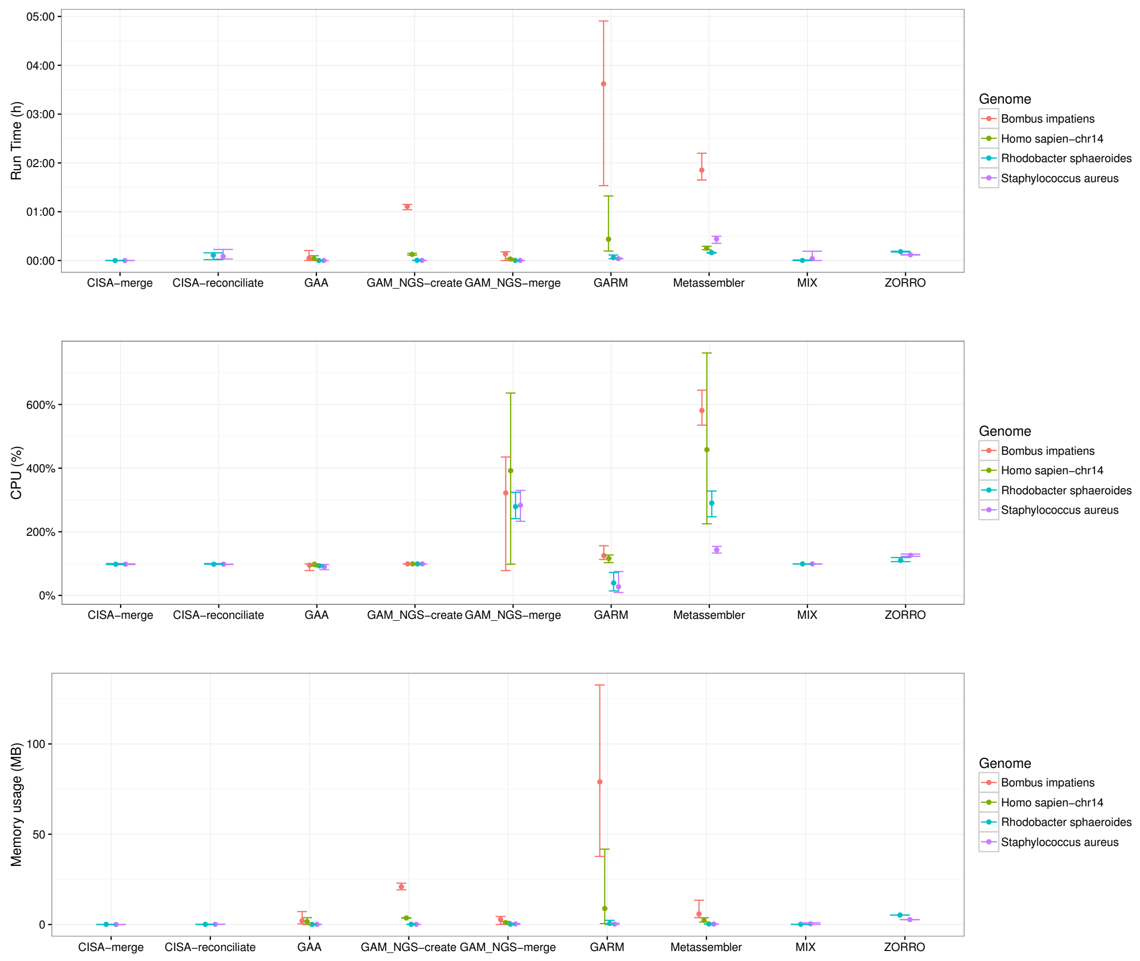


Figure 3.16: Average (wall clock) run time, average percentage of processor utilization (where 100% indicates full utilization of one core), and average memory usage required by each tool to perform each experiments on the four genomes; averages are over all the tested pairs of GAGE assemblies for that genome; error bars indicate the minimum and maximum

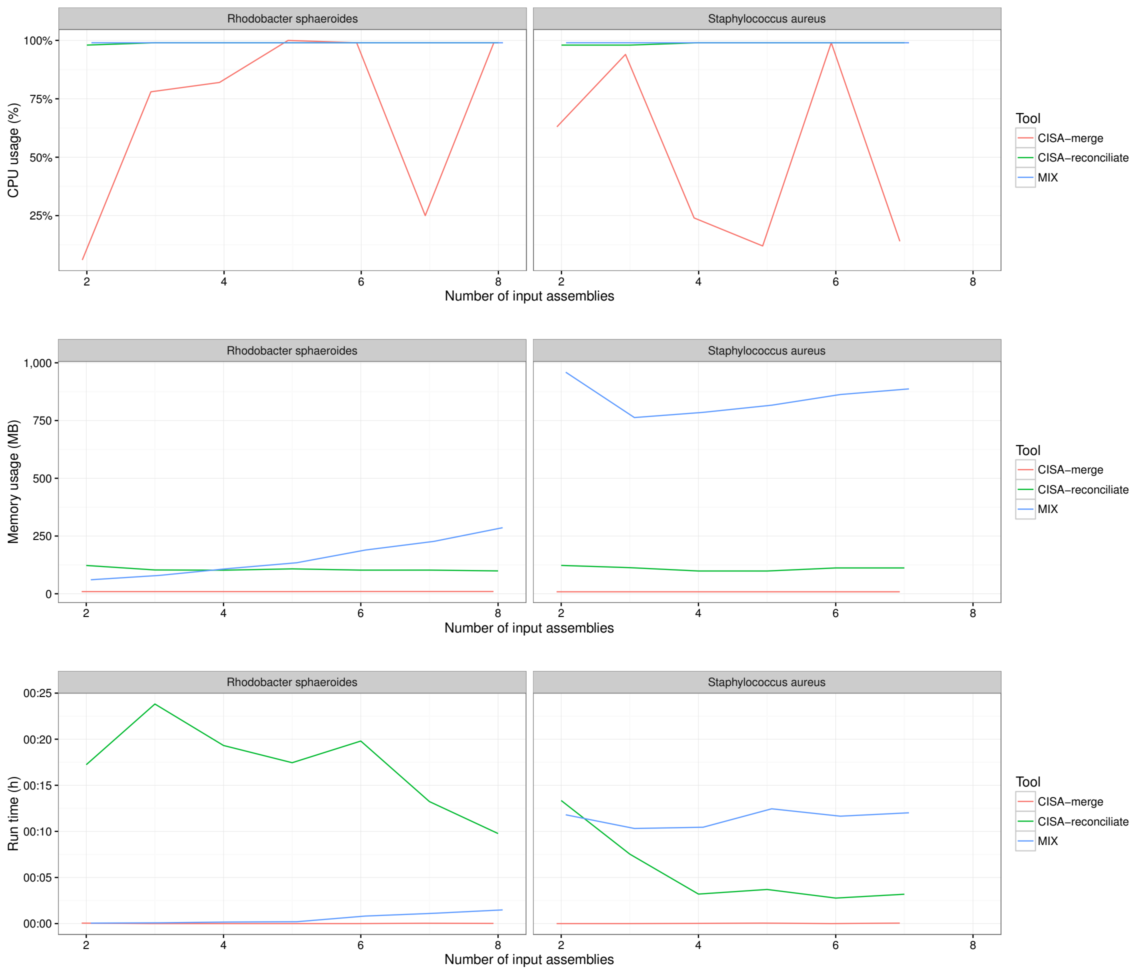


Figure 3.17: Wall clock run time, processor utilization (where 100% indicates full utilization of one core), and memory usage as the number of input assemblies increases (for CISA and MIX)

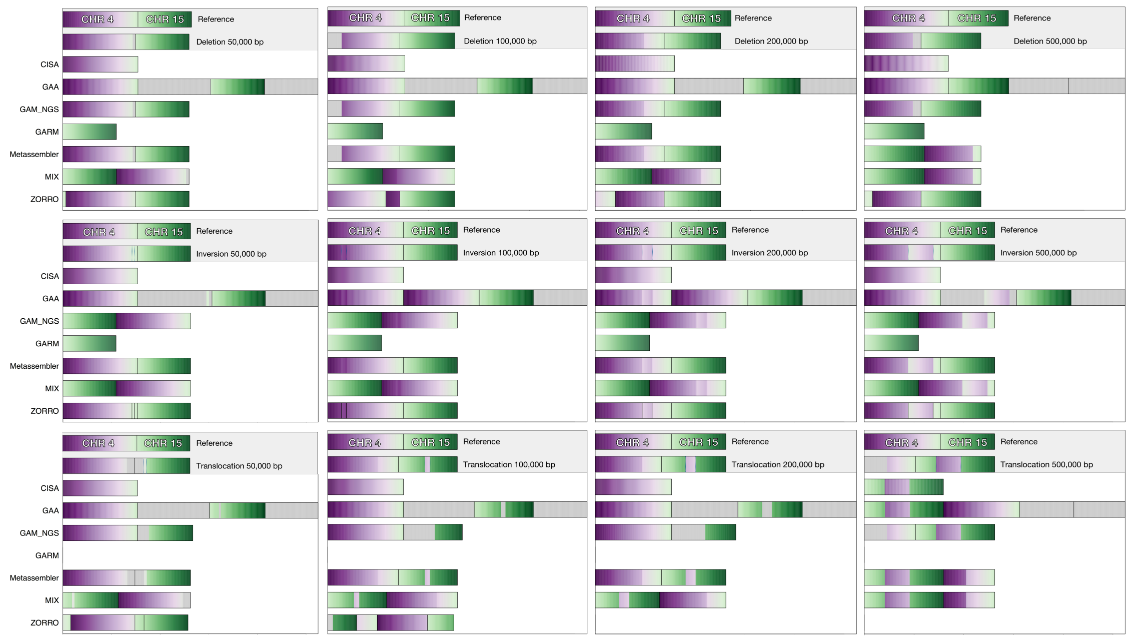


Figure 3.18: The eight assembly reconciliation tools were given in input (A) chromosome 4 and 15 yeast genome and (B) a flawed version of (A) produced by RSVSim containing either a deletion in chromosome 4 (top row), or an inversion in chromosome 4 (middle row) or an translocation from chromosome 4 to chromosome 15 (bottom row); A and B are first two rows in each plot; Decipher was used to detects synteny blocks between the reference and the outputs and to generate synteny plots displayed as gradients: when reference and output disagree, the gradients are interrupted; gray regions indicate blocks that do not match the reference

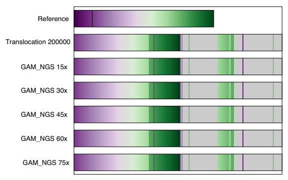
reference and a query sequence, and generates synteny plots displayed as gradients. When reference and query disagree, the gradients are interrupted. Gray regions indicate blocks that do not match the reference.

In each experiment we merged two inputs, namely (1) chromosomes 4 and 15 of the yeast genome and (2) a flawed version of (1) produced by RSVSim containing one structural variation, i.e., either a deletion, an inversion (reversal), or a translocation. RSVSim does not allow *de novo* insertions. For asymmetric tools, the flawed assemblies was used as the master assemblies to model the worst case. We introduced deletions and inversions of various sizes (50 kbp, 100 kbp, 200 kbp, and 500 kbp) into chromosome 4, and generated translocations from chromosome 4 into the chromosome 15 of various sizes (again, 50 kbp, 100 kbp, 200 kbp, and 500kbps).

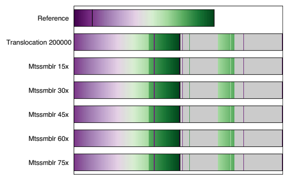
Figure 3.18 (top row) show that CISA resolved the deletion but did not output chromosome 15. GAA also resolved the deletions but it produced two extra sequences that did not align to the reference. GARM did not output chromosome 4. GAM_NGS, Metassembler, and MIX produced assemblies similar to the flawed input assembly. ZORRO broke the assembly at the position of the deletion, produced three individual contigs, and omitted the deleted sequence.

Figure 3.18 (middle row) shows that only CISA resolved the inversion but did not output chromosome 15. GAA did not correct the inversion, and generated a merged assembly that was similar to the flawed input assembly with two additional sequences that did not align to the reference. Again, GAM_NGS, Metassembler, and MIX produced assemblies similar to the flawed assembly. ZORRO broke the inversion by producing three contigs for chromosome 4, and an additional contig representing chromosome 15.

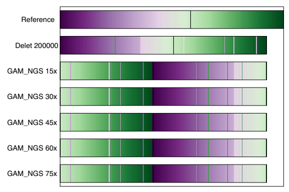
For translocations, the behavior of reconciliation tools depended on the size of the translocation, as shown in Figure 3.18 (bottom row). For translocations of 50, 100 and 200 kbps, CISA, GAA, and GAM_NGS produced the correct version of chromosome 4. CISA did not produce chromosome 15 and GAA and GAM_NGS produced chromosome 15 with an unaligned sequence at the location of the insertion. As before, GAA produced two additional sequences. GARM did produce any merged assembly. Metassembler and MIX's output was similar the flawed input assembly. ZORRO split the assembly over structural



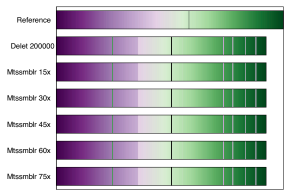
(a) GAM_NGS



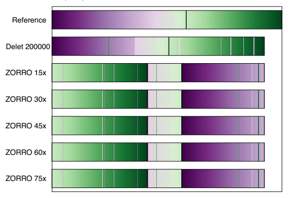
(b) Metassembler



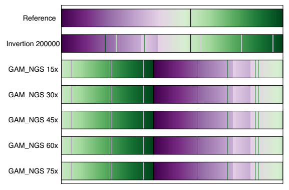
(c) GAM_NGS



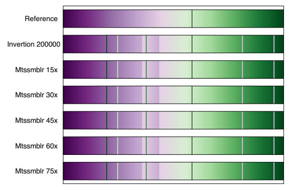
(d) Metassembler



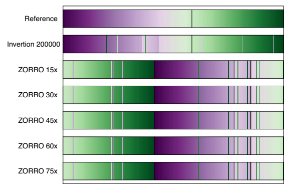
(e) ZORRO



(f) GAM_NGS



(g) Metassembler



(h) ZORRO

Figure 3.19: Assembly reconciliation results for difference choices of read coverage; (a,b) are translocations; (c,d,e) are deletions; (f,g) are inversion

variation breaking points. For 200 and 500 kbps, ZORRO was stopped after allocating more than 350 GB of RAM. None of the tools managed to correct the 500 kbps translocations. CISA and GAA produced the flawed version of chromosome 15. Again, GAA produced the correct version of chromosome 4, but two extraneous sequences. GAM_NGS output was very similar to the input flawed assembly. Metassembler and MIX's produce chromosome 4 without the deleted fragment and a flawed version of chromosome 15.

To test whether read coverage had any impact on the quality of merged assemblies for assembly reconciliation tools that requires reads as input, we ran several experiments on the same synthetic assemblies with increasing reads fold coverage (15x to 75x). Figure 3.19 shows that read coverage did not have any affect on the quality of the results.

3.3 Discussion and Conclusions

Given the practical challenges of *de novo* assembly assembly, the idea of assembly reconciliation is very appealing. One could generate multiple assemblies on the same dataset using various assembly tools and/or parameters, then use an assembly reconciliation tool to merge all the assemblies and obtain a high quality consensus assembly. At the outcome, the expectation is that the quality of the merged assembly should be at least as good as the best assembly in input. In fact, if both input assemblies have some good quality assembly statistics (e.g., one is more contiguous while the other has less misassemblies), one should expect the consensus assembly to inherit the good qualities from both inputs. The reality is that it seems very hard to produce a merged assembly which consistently better than (or at least as good as) both input assemblies. The extensive set of experiments reported in this manuscript showed that none of the tools we evaluated was able to consistently achieve this goal. There were a few cases in which the consensus assembly was better that both inputs, but for the vast majority of the inputs the merged assembly was not.

Despite the inability of these assembly tools to solve the general assembly reconciliation problem, each tool demonstrated some strengths that could lead to algorithmic advances on this problem. For instance, CISA generally was able to correct most structural variations and to ignore duplications in the input assemblies (however, its duplication rate increased as the number of merged assemblies increased); GAA and GARM often improved the contiguity (but often introduced more misassembly errors and increased the duplication ratio); GAM_NGS typically produced consensus assemblies very close to the quality of the reference (but not much better), and it was able to resolve translocations; MIX generally improved the contiguity modestly (but its number of misassemblies was usually close or higher than the most erroneous input, and its genome coverage dropped in some cases); Metassembler often produced consensus assembly with a very low number of misassembly errors, sometimes even lower than both input assemblies (however it did not consistently increase N50); finally, ZORRO generally maintained a high genome coverage (but it did not significantly increase contiguity).

Chapter 4

SequOIA: Sequence Overlap Identification and Assembly

The problem of assembling BAC assemblies to obtain a genome-wide assembly is not new. The public Human Genome Project relied on a tool called GigAssembler [43] to assemble about 30,000 BAC clones using a genome-wide physical map as well as BAC-end sequencing and other genetic markers. GigAssembler used the overlap-layout-consensus approach: it detected prefix-suffix overlaps between BAC contigs to build an overlap graph, it removed transitively-inferable edges, then it found paths in the graph to generate contigs. Unfortunately, GigAssembler has not been maintained since 2001. To the best of our knowledge the only other work in the literature that addresses this problem is MegaWeaver [76], which solves it by computing overlaps between all pairs of BAC assemblies via MegaBlast [86], detects and remove spurious overlaps, then generates a consensus assembly. MegaWeaver is not maintained anymore as well. While most of the genome assemblies follow the whole-shotgun approach, in recent year, there has been renewed interest in the BAC-by-BAC hierarchical sequencing approach (see Introduction for more information about the BAC-by-BAC approach).

Let us first define precisely the BAC assembly problem. We are given in input a set of BAC assemblies $\{\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_n\}$, $n \geq 2$ for a genome G, where each BAC assembly $\mathcal{B}_i$ is a set of unoriented contigs $\{c_1, c_2, \cdots, c_m\}$, $m \geq 1$ (each contig is a string over the DNA alphabet and 'N'). Let c be the fraction of G covered by the contigs in each BAC assemblies. The objective is produce another set of BAC assemblies $\{\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_l\}$, where (i) l is the smallest possible and (ii) the genome coverage of the new set is also c, (iii) the new assemblies do not contain any mis-assembly.

Observe that producing as output the input set is optimal when the BAC assemblies do not overlap. Also observe that producing an empty set as output ($l = 0$) would satisfy (i) and (iii), but not (ii). In order to solve the problem we need to determine BAC overlaps and reduce the redundancy. We propose to use colored positional de Bruijn graph to solve the problem.

4.1 Colored positional de Bruijn graph

Several augmented de Bruijn graph have been introduced in the literature, to address difficulties in finding Eulerian paths in regular de Bruijn graph when dealing with noisy sequencing data for complex, repetitive genome. For instance, sequencing errors the end of reads may result in tips, sequencing errors or mutations towards the middle of reads may introduce in bubbles, and repeats induce a frayed rope structure as shown in Figure 4.1.

Here we consider the *positional* de Bruijn graph and the *colored* de Bruijn graph.

Positional de Bruijn graph. Defined in SEQuel [65] as a variation of the classical de Bruijn graph used in genome assembly, the positional de Bruijn graph is a variation of a de Bruijn graph such that every edge is associated with *kmer* and its inferred positions in contigs. The goal of the tool SEQuel is correct substitutions and small indels smaller than 50 *bps*.

Colored de Bruijn graph. Introduced in [1] to solve the halving problem. In a whole genome duplication evolutionary event, the gene content is duplicated in the offspring and rearranged within the genome. The *halving problem* requires one to reconstruct the pre-duplication ancestral genome. Later the "Cortex" assembler (based on a colored de Bruijn graph graph) was introduced in [41] to assemble multiple genomes simultaneously to detect and genotype genetic variations. Each node in a colored de Bruijn graph is associated with a *kmer* and a list of colors represents all the read sets (i.e., genomes after assembly) containing that *kmer*.

Align Graph was introduced in [7] in order to extend and merge contigs of an existing *de novo* assembly contigs using paired-end reads and guided by a closely related reference genome. The authors align the contigs and paired-end reads to a related reference genome, and exploits the positional information to build a graph that combines reads and contigs. An align graph is a combination of a positional de Bruijn graph and a paired-end de Bruijn graph, where the latter is a generalization of de Bruijn graph that incorporate mate-pair reads distance information.

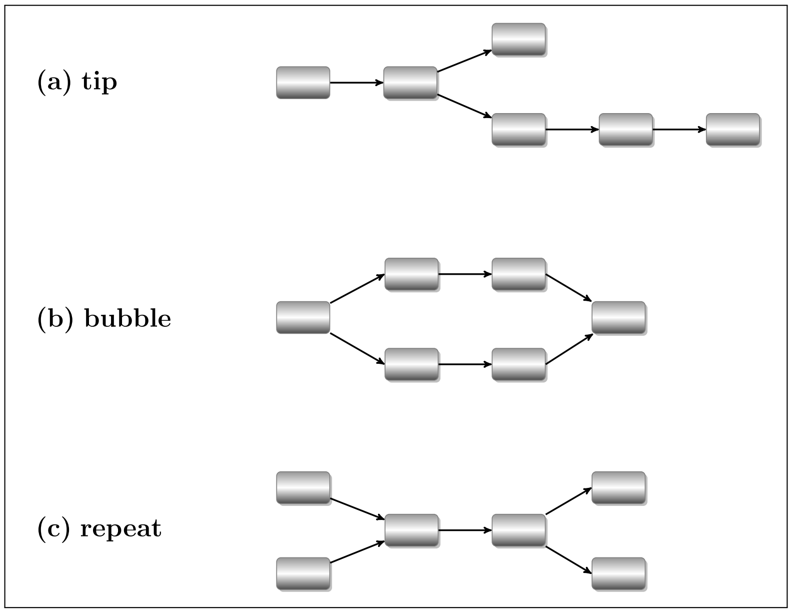


Figure 4.1: de Bruijn graph structure

In our case we are merging assembled BACs to obtain a genome-wide assembly. This problem is an assembly of assemblies, where the input assemblies are expected to have a much lower error rates than reads. Furthermore, the input assemblies are much longer sequences, and their positional information can help to resolve repeats. We propose an algorithm that utilizes positional and color information, and does not require a reference genome of closely related specie.

We assign each input assembly a distinct color. A colored positional de Bruijn graph is an extension of a de Bruijn graph. It is a directed graph where each node p represents a *kmer* and contains a set $\mathcal{L}$ of $(color, pos)$ pairs where *pos* is the starting position of the *kmer* in a sequence uniquely identified by *color*. For any two pairs $(color_i, pos_i) \in \mathcal{L}$ and $(color_j, pos_j) \in \mathcal{L}, color_i \neq color_j$. A labeled directed edge $p \xrightarrow{\alpha} q$ exists in the graph if pair $(color_i, pos_i) \in \mathcal{L}_p$ and pair $(color_j, pos_j) \in \mathcal{L}_q$ are such that $color_i = color_j$ and

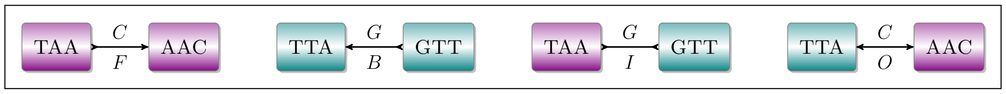


Figure 4.2: Edge orientation and labeling. Each node contains a k-mer ($k = 3$ in this case). The annotation below the edge indicates the assigned direction (Forward, Backward, Innie, Outie), while the annotation on top of the edge is transition nucleotide between the corresponding kmers. Violet denotes *forward* nodes and light blue represent to *backward* nodes.

$|pos_i - pos_j| = 1$. The label $\alpha \in \{A, C, G, T\}$ on the edge and direction $\Delta \in$ {forward, backward, innie, outie} are assigned based on *kmer* orientation of the source and destination nodes. Forward (F) edge connects two forward nodes, backward (B) edge connects two backward nodes, innie (I) edge connects a forward node to backward, and outie (O) connect backward node to forward node. Figure 4.2 illustrates all possible combinations.

4.2 Methods

Our proposed method (called SequOIA) is articulated in four steps: overlap detection, graph construction, graph compaction and graph traversal.

4.2.1 Overlap detection

In the first step, we identify potentially overlapping BAC assemblies. In order to do so, BAC assemblies are clustered into groups based on the Jaccard distance calculated over number of shared *kmer*s between each pair of BACs. The Jaccard distance matrix was first introduced in [14] to cluster webpages. The same approach was also used in locality-sensitive hashing [9], which uses sampling to detect potential overlaps. Our approach generates a k-spectrum for each BAC and calculate a Jaccard-like similarity score for each pair of BACs

(i, j) using the following formula.

$$Jaccard\_score\langle i, j\rangle = max \begin{cases} |i_{<kmers>} \cap j_{<kmers>}| / \min(|i_{<kmers>}|, |j_{<kmers>}|) \\ |i'_{<kmers>} \cap j_{<kmers>}| / \min(|i_{<kmers>}|, |j_{<kmers>}|) \\ |i_{<kmers>} \cap j'_{<kmers>}| / \min(|i_{<kmers>}|, |j_{<kmers>}|) \end{cases}$$

where i' and j' are the DNA reverse complement of i and j, respectively. Note that the BACs is given as a set of unoriented contigs. We do not consider all possible combination of contigs orientation within a BAC, but rather assume that all the contigs in a BAC assembly have the same orientation for the purpose of detecting potential overlaps.

The Jaccard score computes the percentage of shared *kmer*s in relation to size of the smaller BAC. A score above threshold τ indicates potential containment. Score greater than another threshold $\gamma < \tau$ denotes potential overlap. Otherwise, no overlap is reported.

4.2.2 Construction of the colored positional de Bruijn graph

First, we assign each input BAC assembly a unique color. Sequences with the same color are not considered for overlaps (since the belong to the same BAC). We break scaffolds into contigs, then start from an arbitrary assembly from the input. For this arbitrary input assembly, we build the graph by decomposing each contig into *kmer* and creating a node for each *kmer*. We add a forward edge between every two consecutive nodes of the same contig, sorted in ascending positions. For the remaining input assemblies, we process a *kmer* based on following cases

- if $\nexists node\langle kmer\rangle \in G$ or $\forall_{node\langle kmer\rangle} color \in \mathcal{L}_{node}$ then we create a new node
- if $\exists$ only one $node\langle kmer\rangle$ such that $color \notin \mathcal{L}_{node}$ then add $(color, position)$ to $\mathcal{L}_{node}$

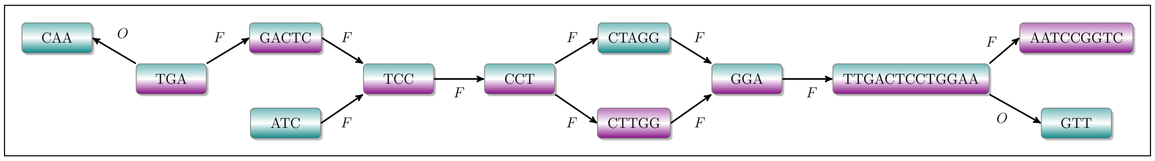


Figure 4.3: An illustration of different kinds of branch nodes

- when multiple nodes for *kmer* exist, we select the best node to merge with based on anchored pairwise alignment between the current sequence and every sequence in $\mathcal{L}_{node}$ of a candidate node (see Algorithm 4.2).

Algorithm 4.1 describes the colored positional de Bruijn graph construction in more details.

4.2.3 Graph compression

We follow the conventional definition of unitigs that a compact node encoding a unitig comprises nodes such that in-degree of all nodes except the first is one and out-degree of all nodes except the last is one. In our algorithm, a normalized confidence score is assigned to each unitig. The score is calculated using the following formula.

$$Confidence\_score = \begin{cases} 0 & \text{if compact node is singleton,} \\ \frac{\sum_{node \in compact\_node} |\mathcal{L}_{node}|}{|unitig|} & \text{otherwise,} \end{cases}$$

Assuming a relatively low number of colors and a large *kmer* size, short unitigs will have lower scores. As the size of the unitigs increases, a higher coverage leads to a higher confidence score.

4.2.4 Graph traversal

Our graph traversal explores alternative paths and produces a string "contig" corresponding to the path with the highest confident score. A node p is considered an *initial* if in-degree$(p) = 0$ or $\forall\ (color, pos) \in \mathcal{L}_p,\ pos = 1$. A *contig* is a path in the graph (or a node in a compact graph). Traversal starts at an initial node and extends to the right and to the left until it generates a contig with *zero* unexplored incoming and outgoing edges.

Given a compact de Bruijn graph, the path extension Algorithm 4.3 solves branches as they appear along the path from the initial node by considering three cases based on the type of the (merge, diverge) and the orientation of alternative branches. We start by solving divergent branches of same orientation. Next, we solve merged branches. Lastly, we solve any branch not addressed by the two previous cases.

Branches with same orientation represent alternative paths in the form of bubbles or tips (Figure 4.5 shows an example). This case has straightforward solution presented in Algorithm 4.4. We simply select a branch with the maximum confident score, and ties are solved by branch length to achieve maximal extension. In the case of divergent branches represent a bubble, selecting which branch to consider for extension depends only of the fragments between endpoints of the bubble. In case of tips, selecting a branch requires solving all subsequent bifurcations in these branches.

Merged branches can be a part of a bubble, for which the solution is to be deferred to the previous case. Merged branches can also be tips, and therefore solved using the same algorithm in the previous case. Otherwise, we merge branches if the merging node has no

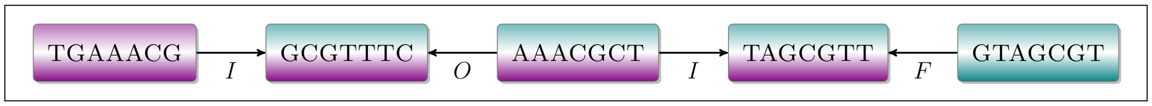


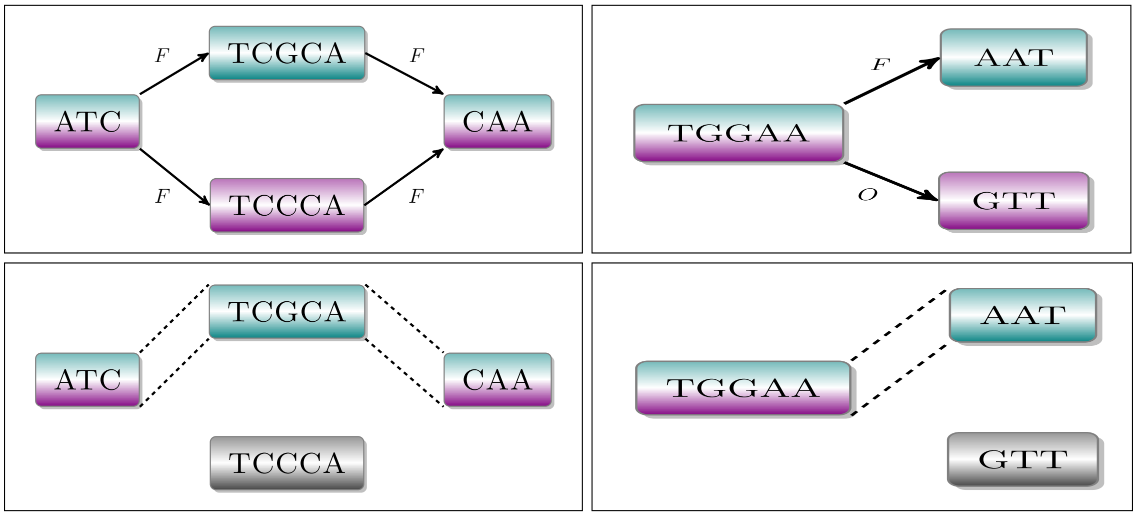
Figure 4.4: Example contains different kinds of SEQ1: TGAAACGCTAC SEQ2: GTAGCGTTTCA, MERGED SEQUENCE: GTAGCGTTTCA branch nodes

outgoing edges. In case the merging node has an outgoing edge, we recursively solve all subsequent bifurcations using the EXTEND algorithm. We now solve for three branches, the merged two branches and the newly acquired potential extension branch. Since two of the three branches must be a part of alternative paths, we select the best alternative and merge the third branch (see Algorithm 4.5).

Branches with different orientation are either extending the source in the same direction (we call these branches *alternative paths*) or extend the source node in different directions (we call these branches *extension paths*). Figures 4.4 and 4.6 illustrate extension paths, while Figure 4.5b illustrates alternative paths. If two branches are parts of alternative paths, we select the most path with higher confidence as explained earlier. If the two branches are extensions, if the branch node is singleton, we merge. Otherwise, the node branches into alternative paths. In this case, we select the path with the highest confidence score and merge afterwards (see Algorithm 4.6 for details).

4.3 Experimental Results

To test SequOIA, we used *Vigna unguiculata* (cowpea) assembled BACs, generated at UC Riverside [59]. The datasets contains 4355 BACs, where each BAC assembly has on



(a) Bubble branch (b) Tip branch

Figure 4.5: Examples illustrate solving bubble and tip branches

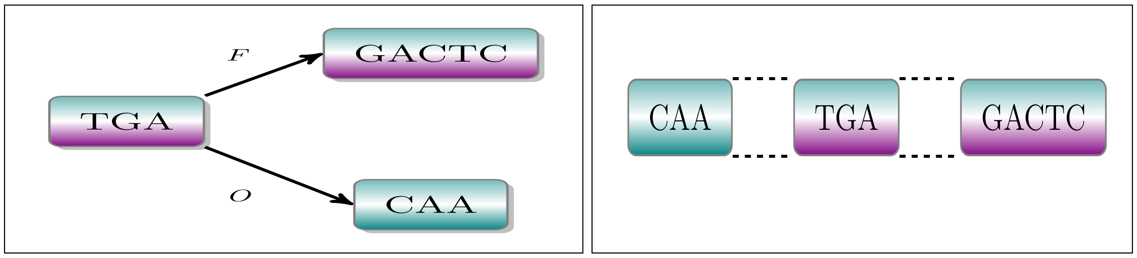


Figure 4.6: Examples illustrate solving extend branches

average of 29 scaffolds. Each BAC assembly has an average N50 of approximately 14 kbp [50]. We excluded 97 BAC assemblies smaller than 5000 nucleotides from the dataset.

We detected overlaps between BACs using the Jaccard-distance matrix method described above. BACs with similarity score greater than 90% typically indicate containment; similarity scores greater than 80% are considered potential overlaps. Our algorithm generated 676 clusters containing on average of 2.67 BACs, with 292 BACs belonging to more than one cluster, and 1444 BACs appearing in at least one cluster.

Each cluster of overlapping BACs was assembled using SequOIA with four *kmer*

Table 4.1: Quality statistics of merged cowpea BACs for SequOIA and CANU. All statistics were generated with QUAST. Statistics below the double lines based on contigs of size $\geq$ 500 bp. The reference assembly is 474,399,596bp.

Assembly	Input BACs	CANU	SequOIA
# contigs ($\geq$ 0 bp)	24,730	7	9550
# contigs ($\geq$ 25,000 bp)	679	1	604
# contigs ($\geq$ 50,000 bp)	105	0	99
Total length ($\geq$ 0 bp) (bp)	94,386,917	60,976	75,504,631
Total length ($\geq$ 25,000 bp) (bp)	26,287,750	31,437	23,429,428
Total length ($\geq$ 50,000 bp) (bp)	6,854,907	0	6,521,556
# contigs	16,069	7	9356
Longest contig (bp)	141,058	31,437	141,104
Total length (bp)	91,591,924	60,976	75,442,096
N50 (bp)	14,602	31,437	16,169
N75 (bp)	6299	6067	7806
L50	1709	1	1318
L75	4086	3	2985
Genome fraction (%)	14.689%	0.011%	13.606%
Duplication ratio	1.297	1.132	1.154
# N's per 100 kbp	23.50	0.00	0.00
Longest alignment (bp)	141,058	17,260	141,104
Total aligned length (bp)	90,135,040	60,974	74,335,415
NA50 (bp)	12,903	14,177	14,177
NA75 (bp)	5413	6067	6677
LA50	1943	2	1515
LA75	4675	4	3454

sizes ranging from $2^8 - 1$ to $2^{11} - 1$. We select the merged assembly that maximizes the decrease in number of contigs compared to the input. The resulting assembly aggregates sets of contigs from all performed merges, in addition to BACs not identified to have potential overlaps. We compared SequOIA to the long read assembler CANU [44]. SequOIA and CANU were run on the same input. Table 4.1 shows the statistics of the input BACs, CANU's output assembly, and SequOIA output assembly. Statistics were generated using QUAST [32].

CANU produced assembly with a higher N50 but only seven contigs that covered less that 1% of the reference genome. The input assemblies covered around 14%. SequOIA produced an assembly that covers 13% of the genome, while containing 62% fewer contigs.

SequOIA also improved 10% the N50 value compared to the input. Both SequOIA and CANU produced equal a similar NA50 value. SequOIA's statistics showed better NA75 and N75 values (please refer to Section 3.1 for more information regarding the definitions of N50 and NA50 values).

4.4 Conclusion

We introduced SequOIA, a new tool for the assembly of BAC assemblies. SequOIA uses a Jaccard-like similarity-matrix clustering approach to detect overlaps between BACs based on the number of shared *kmer*s. To merge overlapping BAC assemblies, SequOIA uses a new version of de Bruijn graph, which combines a colored de Bruijn graph and positional de Bruijn graph. Our new de Bruijn graph utilizes the knowledge of *kmer* position within the sequence to avoids collapsing repeats within the same sequence. Considering that a BACs is an unordered unoriented set of sequences assumed to be non-overlapping, the color information prevents collapsing repeats within a set of sequences sharing the same color. The new data structure also allows one to devise a voting scheme to find the path with the highest confidence. We tested SequOIA assembler on cowpea BAC assemblies produced at UC Riverside. SequOIA successfully increased the contiguity, while producing an assembly containing 62% fewer contigs than the input covering similar genome portion.

Algorithm 4.1 Build colored positional de Bruijn graph

```
1: function BuildGraph(assemblies, $k$: $kmer\_size$)
2:     select an arbitrary assembly and assign unique color
3:     for each $contigs$ in $assembly$ do
4:         $kmer \leftarrow contig[pos : k]$    ▷ $pos = 1$, substring of length k starting at position 1
5:         $node_1 \leftarrow$ new $node\langle kmer, color, pos\rangle$
6:         for $kmer$ in $contig$ do    ▷ excluding the first
7:             $node_2 \leftarrow$ new $node\langle kmer, color, pos\rangle$
8:             AddEdge(Forward, $node_1, node_2$)
9:             $node_1 \leftarrow node_2$

10:     for each $assembly$ do    ▷ excluding the first
11:         $color \leftarrow$ new unique color
12:         parse each $contig \in assembly$
13:         Get $node_1$ following cases in lines 15 to 24.
14:         for $kmer$ in $contig$ do
15:             if $\nexists$ $node\langle kmer\rangle$ then    ▷ If no node represents $kmer$ in Graph
16:                 $node_2 \leftarrow$ new $node\langle kmer, color, pos\rangle$
17:             else if $\exists$ singleton $node\langle kmer\rangle$ then
18:                 if $color \notin node\langle colors\rangle$ then
19:                     add $color$ to $node\langle colors\rangle$
20:                     $node_2 \leftarrow node$
21:                 else $node_2 \leftarrow$ new $node\langle kmer, color, pos\rangle$
22:             else if $node\langle kmer\rangle \in node_1\langle edges\rangle$ and $color \notin node\langle kmer\rangle$ then    ▷ if the previous node points to a node represent $kmer$, reuse that node
23:                 $node_2 \leftarrow node\langle kmer\rangle$
24:             else $node_2 \leftarrow$ Insert_Repeat($kmer, color, pos$)
25:             AddEdge($node_1, node_2$)
26:             $node1 \leftarrow node_2$
27:     end function
```

Algorithm 4.2 Insert repeated sequence to de Bruijn graph

```
function INSERT_REPEAT( $kmer$: sequence, $color$: ID, $pos$: integer)
    declare local $is\_merged \leftarrow false$
    declare local $max\_score \leftarrow 0$
    for each $node\langle kmer\rangle$ in Graph s.t. $color \notin node\langle colors\rangle$ do
        for $color \in node\langle colors\rangle$ do
            $sequence' \leftarrow$ GETSEQUENCE($node, color$)
            $score \leftarrow$ PAIRWISE_ALIGNMENT($sequence, pos, sequence', node\langle pos\rangle$)
            if $score > max\_score$ then
                $max\_node \leftarrow node$
                $is\_merged \leftarrow true$
                $max\_score \leftarrow score$
    if $is\_merged = true$ then $node \leftarrow max\_node$
    else $node \leftarrow$ new $node\langle kmer, color, pos\rangle$
    return $node$
```

Algorithm 4.3 SequOIA de Bruijn graph traversal algorithm

```
function EXTEND( $p$: compact node, $process\_path$: set of compact nodes)
    declare local $in\_edges \leftarrow$ GETINCOMINGEDGES($p$)
    declare local $out\_edges \leftarrow$ GETOUTGOINGEDGES($p$)
    if $p \in process\_path$ then return NULL
    if all $in\_edges$ are explored and all $out\_edges$ are explored then return $p$
    $process\_path \leftarrow$ INSERT($p$)
    if $p \in$ branch nodes then
        for Edge Direction $\in$ {forwards, backwards, innie, outie} do
            if |unexplored $edges$| $\in$ Edge Direction $> 1$ then
                $p \leftarrow$ RESOLVEBUBBLEORTIP($p, process\_path, EdgeDirection$)
    if $p \in$ combine nodes then
        $p \leftarrow$ RESOLVECOMBINE($p, process\_path$)
    if |unexplored $edges$| $> 0$ then
        $p \leftarrow$ RESOLVEBRANCH($p, process\_path$)
    $process\_path \leftarrow$ ERASE($p$)
    return $p$
```

Algorithm 4.4 SequOIA de Bruijn graph traversal algorithm – Solving Bubbles and Tips

```
1: function resolveBubbleOrTip( $p$: compact node, $process\_path$: set of compact nodes, $EdgeDirection$)
2:     declare local $out\_edges \leftarrow$ GetOutgoingEdges($p, EdgeDirection$)
3:     declare local $max\_score \leftarrow 0$
4:     declare local $best\_extension \leftarrow$ Null
5:     for $e \in out\_edges$ do
6:         $q \leftarrow p[e]$
7:         MarkVisited($e$)
8:         $q \leftarrow$ Extend($q, process\_path$)
9:         $score \leftarrow$ Score($p$)
10:        if $score > max\_score$ then
11:            $max\_score \leftarrow score$
12:            $best\_extension \leftarrow q$
13:        if $score = max\_score$ then
14:            if unitigLength($q$) > unitigLength($best\_extension$) then
15:                $best\_extension \leftarrow q$
16:    $p \leftarrow$ Merge($p, best\_extension$)
17:    return $p$
```

Algorithm 4.5 SequOIA de Bruijn graph traversal algorithm – Solving combine

```
1: function resolveCombine( $p$: compact node, $process\_path$: set of compact nodes)
2:     declare local $tip \leftarrow false$
3:     declare local $combined\_nodes \leftarrow$ GetCombineBranches($p$)
4:     declare local $edges \leftarrow$ GetOutgoingEdges($p$)
5:     $process\_path \leftarrow$ Insert($combined\_nodes$)
6:     if $combined\_nodes \subseteq start\_nodes$ or
7:         $combined\_nodes \subseteq end\_nodes$ then $tip \leftarrow true$
   Consider two combined nodes $p$ and $q$
8:     if sameDirection($p, q$) then
9:         if $tip = true$ then                          ▷ If tip, select the return best branch
10:            $p \leftarrow$ resolveBubbleOrTip($p, q$)
11:        return $p$                ▷ if bubble, relegate resolving to the source of the bubble
12:    if $|out\_edges| = 0$ then
13:        $p \leftarrow$ Merge($p, q$)
14:        return $p$
15:    else
16:        $s \leftarrow p[e]$                                              ▷ potential suffix
17:        MarkVisited($e$)
18:        $s \leftarrow$ Extend($s, process\_path$)
19:        if sameDirection($p, s$) then
20:            $s \leftarrow$ resolveBubbleOrTip($p, s$)
21:            $p \leftarrow$ Merge($q, s$)
22:        else if sameDirection($q, s$) then
23:            $s \leftarrow$ resolveBubbleOrTip($q, s$)
24:            $p \leftarrow$ Merge($p, s$)
25:        else
26:            $p \leftarrow$ resolveBubbleOrTip($p, q$)
27:            $p \leftarrow$ Merge($p, s$)
28:    return $p$
```

Algorithm 4.6 SequOIA de Bruijn graph traversal algorithm – Solving branches

```
1: function resolveBranch( $p$: compact node, $process\_path$: set of compact nodes)
2:     declare local $edges \leftarrow$ GetOutgoingEdges($p, EdgeDirection$)
3:     if ($e_{fwd} \in edges$ and $e_{outie} \in edges$ then
4:         $u \leftarrow p[e_{fwd}]$
5:         $v \leftarrow p[e_{outie}]$
6:         MarkVisited($e_{fwd}$)
7:         MarkVisited($e_{outie}$)
8:         $p \leftarrow resolveBubbleOrTip(u, v)$
9:     if ($e_{bck} \in edges$ and $e_{innie} \in edges$ then
10:        $u \leftarrow p[e_{bck}]$
11:        $v \leftarrow p[e_{innie}]$
12:        $p \leftarrow$ resolveBubbleOrTip($u, v$)
13:    for $e \in edges$ do
14:        $q \leftarrow p[e]$
15:        MarkVisited($e$)
16:        $q \leftarrow$ Extend($q, process\_path$)
17:        $p \leftarrow$ Merge($p, q$)
18:    return $p$
```

Chapter 5

Conclusion

In this dissertation we presented novel data structures and computational methods to detect sequence overlaps and assemble overlapped sequences. We introduced Sequence Decision Diagrams which are data structures that can compactly store finite sets of strings and presented algorithms to efficiently perform set operation on them via *memoization* a natural feature of decision diagrams. We also provided an algorithm to solve the all-pair suffix-prefix problem using Sequence Decision Diagrams. In practice, genomic sequences contain many variations due to SNPs, sequencing errors, and misassemblies, among other reasons.

As part of SequOIA, we developed a tool that detects overlaps between sequences based on a Jaccard-like similarity score calculated over the number of shared *kmers* between the two sequences. The use of *kmers* allows for error resilience in detecting potential overlaps. We used this approach to detect potential overlaps between BACs, represented as contigs.

The second component of SequOIA merges overlapping assemblies (represented as sets of contigs) to create longer contigs. The algorithm is based on an augmented de Bruijn graph that we developed. Our de Bruijn graph is a hybrid of the *positional* de Bruijn graph and *colored* de Bruijn graph, that exploits the *a priori* knowledge of the *kmer* positioning within a sequence and to which set that sequence belongs. We showed that our augmented de Bruijn graph can resolve most repeats and produce a de Bruijn graph with fewer path discrepancies. We used SequOIA to merge assembled cowpea BAC clones.

We also presented an extensive comparative analysis of the state-of-the-art assembly reconciliation tools, to better understand the performance of these tools on this hard problem. If assembly reconciliation was solved properly it would very useful. Since it is a common practice to produce multiple assemblies using different assemblers, parameters, or even sequencing technologies, the ability to reconcile multiple assemblies would allow one to leverage the strengths of each assembler/parameters and obtain a higher quality merged assembly.

Bibliography

[1] M.A. Alekseyev and P.A. Pevzner. Colored de Bruijn Graphs and the Genome Halving Problem. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):98–107, January 2007.

[2] Hind Alhakami, Gianfranco Ciardo, and Marek Chrobak. Sequence Decision Diagrams. In *String Processing and Information Retrieval*, pages 149–160. Springer, Cham, October 2014.

[3] Husain Aljazzar, Holger Hermanns, and Stefan Leue. Counterexamples for timed probabilistic reachability. In *Formal Modeling and Analysis of Timed Systems*, number 3829 in Lecture Notes in Computer Science, pages 177–195. Springer Berlin Heidelberg, January 2005.

[4] Husain Aljazzar and Stefan Leue. Extended directed search for probabilistic timed reachability. In *Formal Modeling and Analysis of Timed Systems*, number 4202 in Lecture Notes in Computer Science, pages 33–51. Springer Berlin Heidelberg, January 2006.

[5] H. Aoki, S. Yamashita, and S. Minato. An efficient algorithm for constructing a sequence binary decision diagram representing a set of reversed sequences. In *2011 IEEE International Conference on Granular Computing (GrC)*, pages 54–59, 2011.

[6] Juan Lucas Argueso, Marcelo F. Carazzolle, Piotr A. Mieczkowski, Fabiana M. Duarte, Osmar V.C. Netto, Silvia K. Missawa, Felipe Galzerani, Gustavo G.L. Costa, Ramon O. Vidal, Melline F. Noronha, Margaret Dominska, Maria G.S. Andrietta, Silvio R. Andrietta, Anderson F. Cunha, Luiz H. Gomes, Flavio C.A. Tavares, Andre R. Alcarde, Fred S. Dietrich, John H. McCusker, Thomas D. Petes, and Goncalo A.G. Pereira. Genome structure of a Saccharomyces cerevisiae strain widely used in bioethanol production. *Genome Res*, 19(12):2258–2270, December 2009.

[7] E. Bao, T. Jiang, and T. Girke. AlignGraph: algorithm for secondary de novo genome assembly guided by closely related references. *Bioinformatics*, 30(12):i319–i328, June 2014.

[8] Christoph Bartenhagen and Martin Dugas. RSVSim: an R/Bioconductor package for the simulation of structural variations. *Bioinformatics*, 29(13):1679–1681, July 2013.

[9] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P. Drake, Jane M. Landolin, and Adam M. Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat Biotech*, 33(6):623–630, June 2015.

[10] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell. Building the minimal DFA for the set of all subwords of a word on-line in linear time. In *Automata, Languages and Programming*, number 172 in Lecture Notes in Computer Science, pages 109–118. Springer Berlin Heidelberg, January 1984.

[11] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, January 1985.

[12] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, September 1996.

[13] Keith R. Bradnam, Joseph N. Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, Inan Birol, Sbastien Boisvert, Jarrod A. Chapman, Guillaume Chapuis, Rayan Chikhi, Hamidreza Chitsaz, Wen-Chi Chou, Jacques Corbeil, Cristian Del Fabbro, T. Roderick Docking, Richard Durbin, Dent Earl, Scott Emrich, Pavel Fedotov, Nuno A. Fonseca, Ganeshkumar Ganapathy, Richard A. Gibbs, Sante Gnerre, lnie Godzaridis, Steve Goldstein, Matthias Haimel, Giles Hall, David Haussler, Joseph B. Hiatt, Isaac Y. Ho, Jason Howard, Martin Hunt, Shaun D. Jackman, David B. Jaffe, Erich D. Jarvis, Huaiyang Jiang, Sergey Kazakov, Paul J. Kersey, Jacob O. Kitzman, James R. Knight, Sergey Koren, Tak-Wah Lam, Dominique Lavenier, Franois Laviolette, Yingrui Li, Zhenyu Li, Binghang Liu, Yue Liu, Ruibang Luo, Iain MacCallum, Matthew D. MacManes, Nicolas Maillet, Sergey Melnikov, Delphine Naquin, Zemin Ning, Thomas D. Otto, Benedict Paten, Octvio S. Paulo, Adam M. Phillippy, Francisco Pina-Martins, Michael Place, Dariusz Przybylski, Xiang Qin, Carson Qu, Filipe J. Ribeiro, Stephen Richards, Daniel S. Rokhsar, J. Graham Ruby, Simone Scalabrin, Michael C. Schatz, David C. Schwartz, Alexey Sergushichev, Ted Sharpe, Timothy I. Shaw, Jay Shendure, Yujian Shi, Jared T. Simpson, Henry Song, Fedor Tsarev, Francesco Vezzi, Riccardo Vicedomini, Bruno M. Vieira, Jun Wang, Kim C. Worley, Shuangye Yin, Siu-Ming Yiu, Jianying Yuan, Guojie Zhang, Hao Zhang, Shiguo Zhou, and Ian F. Korf. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaSci*, 2(1):1–31, December 2013.

[14] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the Web. *Computer Networks and ISDN Systems*, 29(813):1157–1166, 1997.

[15] Mark J. Chaisson, Dumitru Brinza, and Pavel A. Pevzner. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome Res.*, 19(2):336–346, February 2009.

[16] Mahul Chakraborty, James G. Baldwin-Brown, Anthony D. Long, and J. J. Emerson. Contiguous and accurate de novo assembly of metazoan genomes with modest long read coverage. *Nucleic Acids Research*, page gkw654, July 2016.

[17] Ye-In Chang, Jiun-Rung Chen, and Yueh-Chi Tsai. Mining subspace clusters from DNA microarray data using large itemset techniques. *Journal of Computational Biology*, 16(5):745–768, May 2009.

[18] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state space generation. In *Proc. TACAS*, pages 328–342, 2001.

[19] James Clarke, Hai-Chen Wu, Lakmal Jayasinghe, Alpesh Patel, Stuart Reid, and Hagan Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nature Nanotechnology*, 4(4):265–270, April 2009.

[20] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *Proc. Formal Description Techniques, FORTE95*, volume 3731 of *LNCS*, pages 443–4572, 2005.

[21] Maxime Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[22] Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. In *Structures in Logic and Computer Science*, number 1261 in Lecture Notes in Computer Science, pages 192–211. Springer Berlin Heidelberg, January 1997.

[23] B. Damman, Tingting Han, and J. Katoen. Regular expressions for PCTL counterexamples. In *Fifth International Conference on Quantitative Evaluation of Systems, 2008. QEST '08*, pages 179–188, 2008.

[24] Shuhei Denzumi, Hiroki Arimura, and Shin-ichi Minato. Substring indices based on bdds. *TCS technical Reports, TCS-TR-A-10*, 42, 2010.

[25] Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura, and Shin ichi Minato. Notes on sequence binary decision diagrams: Relationship to acyclic automata and complexities of binary set operations. In *Proceedings of the Prague Stringology Conference 2011*, pages 147–161, Czech Technical University in Prague, Czech Republic, 2011.

[26] Shuhei Denzumi, Ryo Yoshinaka, Shin-ichi Minato, and Hiroki Arimura. Efficient algorithms on sequence binary decision diagrams for manipulating sets of strings. Technical report, Technical Report, DCS, Hokkaido U., TCS-TR-A-11-53, 2011.

[27] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, Arkadiusz Bibillo, Keith Bjornson, Bidhan Chaudhuri, Frederick Christians, Ronald Cicero, Sonya Clark, Ravindra Dalal, Alex deWinter, John Dixon, Mathieu Foquet, Alfred Gaertner, Paul Hardenbol, Cheryl Heiner, Kevin Hester, David Holden, Gregory Kearns, Xiangxu Kong, Ronald Kuse, Yves Lacroix, Steven Lin, Paul Lundquist, Congcong Ma, Patrick Marks, Mark Maxham, Devon Murphy, Insil Park, Thang Pham, Michael Phillips, Joy Roy, Robert Sebra, Gene Shen, Jon Sorenson, Austin Tomaney, Kevin Travers, Mark Trulson, John Vieceli, Jeffrey Wegener, Dawn Wu, Alicia Yang, Denis Zaccarin, Peter Zhao, Frank

Zhong, Jonas Korlach, and Stephen Turner. Real-Time DNA Sequencing from Single Polymerase Molecules. *Science*, 323(5910):133–138, January 2009.

[28] Adam C. English, Stephen Richards, Yi Han, Min Wang, Vanesa Vee, Jiaxin Qu, Xiang Qin, Donna M. Muzny, Jeffrey G. Reid, Kim C. Worley, and Richard A. Gibbs. Mind the Gap: Upgrading Genomes with Pacific Biosciences RS Long-Read Sequencing Technology. *PLoS One*, 7(11), November 2012.

[29] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J. Ribeiro, Joshua N. Burton, Bruce J. Walker, Ted Sharpe, Giles Hall, Terrance P. Shea, Sean Sykes, Aaron M. Berlin, Daniel Aird, Maura Costello, Riza Daza, Louise Williams, Robert Nicol, Andreas Gnirke, Chad Nusbaum, Eric S. Lander, and David B. Jaffe. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *PNAS*, 108(4):1513–1518, January 2011.

[30] Eric D. Green. Strategies for the systematic sequencing of complex genomes. *Nature Reviews Genetics*, 2(8):573–583, August 2001.

[31] Stuart J. Green, Reigh P. Monreal, Alan T. White, Thomas G. Bayer, Stuart J. Green, Reigh P. Monreal, Alan T. White, Thomas G. Bayer, Yasmin D. Arquiza, Alan T. White, Stuart J. Green, R. Buenaflor, and Jr. Nd Y. D. Arquiza. Phrap documentation, 1999.

[32] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, April 2013.

[33] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[34] Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the All Pairs Suffix-Prefix Problem. *Information Processing Letters*, 41(4):181–185, March 1992.

[35] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques: Concepts and Techniques.* Elsevier, June 2011.

[36] Tingting Han, J. Katoen, and D. Berteun. Counterexample generation in probabilistic model checking. *IEEE Transactions on Software Engineering*, 35(2):241–257, 2009.

[37] Tingting Han and Joost-Pieter Katoen. Counterexamples in probabilistic model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 4424 in Lecture Notes in Computer Science, pages 72–86. Springer Berlin Heidelberg, January 2007.

[38] Bernhard Haubold and Thomas Wiehe. *Introduction to Computational Biology: An Evolutionary Approach.* Springer, January 2006.

[39] David Hernandez, Patrice Franois, Laurent Farinelli, Magne sters, and Jacques Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, May 2008.

[40] Weichun Huang, Leping Li, Jason R. Myers, and Gabor T. Marth. ART: a next-generation sequencing read simulator. *Bioinformatics*, 28(4):593–594, February 2012.

[41] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nat Genet*, 44(2):226–232, February 2012.

[42] William R. Jeck, Josephine A. Reinhardt, David A. Baltrus, Matthew T. Hickenbotham, Vincent Magrini, Elaine R. Mardis, Jeffery L. Dangl, and Corbin D. Jones. Extending assembly of short DNA sequences to handle error. *Bioinformatics (Oxford, England)*, 23(21):2942–2944, November 2007.

[43] W. James Kent and David Haussler. Assembly of the Working Draft of the Human Genome with GigAssembler. *Genome Res.*, 11(9):1541–1548, September 2001.

[44] Sergey Koren, Brian P. Walenz, Konstantin Berlin, Jason R. Miller, Nicholas H. Bergman, and Adam M. Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *bioRxiv*, page 071282, January 2017.

[45] Stefan Kurtz, Adam Phillippy, Arthur L. Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, January 2004.

[46] M. Kwiatkowska. Model checking for probability and time: from theory to practice. In *18th Annual IEEE Symposium on Logic in Computer Science, 2003. Proceedings*, pages 351–360, 2003.

[47] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, Songgang Li, Huanming Yang, Jian Wang, and Jun Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, 20(2):265–272, February 2010.

[48] Shin-Hung Lin and Yu-Chieh Liao. CISA: Contig Integrator for Sequence Assembly of Bacterial Genomes. *PLoS ONE*, 8(3):e60843, 2013.

[49] Elsa Loekito, James Bailey, and Jian Pei. A binary decision diagram based approach for mining frequent subsequences. *Knowledge and Information Systems*, 24(2):235–268, August 2010.

[50] Stefano Lonardi, Hamid Mirebrahim, Steve Wanamaker, Matthew Alpert, Gianfranco Ciardo, Denisa Duma, and Timothy J. Close. When less is more: 'slicing' sequencing data improves read decoding accuracy and de novo assembly quality. *Bioinformatics*, 31(18):2972–2980, September 2015.

[51] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, Jingbo Tang, Gengxiong Wu, Hao Zhang, Yujian Shi, Yong Liu, Chang Yu, Bo Wang, Yao Lu, Changlei Han, David W

Cheung, Siu-Ming Yiu, Shaoliang Peng, Zhu Xiaoqian, Guangming Liu, Xiangke Liao, Yingrui Li, Huanming Yang, Jian Wang, Tak-Wah Lam, and Jun Wang. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *Gigascience*, 1:18, December 2012.

[52] Tanja Magoc, Stephan Pabinger, Stefan Canzar, Xinyue Liu, Qi Su, Daniela Puiu, Luke J. Tallon, and Steven L. Salzberg. GAGE-B: an evaluation of genome assemblers for bacterial organisms. *Bioinformatics*, 29(14):1718–1725, July 2013.

[53] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948, October 1993.

[54] Marcel Margulies, Michael Egholm, William E. Altman, Said Attiya, Joel S. Bader, Lisa A. Bemben, Jan Berka, Michael S. Braverman, Yi-Ju Chen, Zhoutao Chen, Scott B. Dewell, Lei Du, Joseph M. Fierro, Xavier V. Gomes, Brian C. Goodwin, Wen He, Scott Helgesen, Chun He Ho, Gerard P. Irzyk, Szilveszter C. Jando, Maria L.I. Alenquer, Thomas P. Jarvie, Kshama B. Jirage, Jong-Bum Kim, James R. Knight, Janna R. Lanza, John H. Leamon, Steven M. Lefkowitz, Ming Lei, Jing Li, Kenton L. Lohman, Hong Lu, Vinod B. Makhijani, Keith E. McDade, Michael P. McKenna, Eugene W. Myers, Elizabeth Nickerson, John R. Nobile, Ramona Plant, Bernard P. Puc, Michael T. Ronan, George T. Roth, Gary J. Sarkis, Jan Fredrik Simons, John W. Simpson, Maithreyan Srinivasan, Karrie R. Tartaro, Alexander Tomasz, Kari A. Vogt, Greg A. Volkmer, Shally H. Wang, Yong Wang, Michael P. Weiner, Pengguang Yu, Richard F. Begley, and Jonathan M. Rothberg. Genome Sequencing in Open Microfabricated High Density Picoliter Reactors. *Nature*, 437(7057):376–380, September 2005.

[55] Luz Mayela Soto-Jimenez, Karel Estrada, and Alejandro Sanchez-Flores. GARM: Genome Assembly, Reconciliation and Merging Pipeline. *Current Topics in Medicinal Chemistry*, 14(3):418–424, February 2014.

[56] Edward M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, 23(2):262–272, April 1976.

[57] Jason R. Miller, Arthur L. Delcher, Sergey Koren, Eli Venter, Brian P. Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, December 2008.

[58] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *30th Conference on Design Automation, 1993*, pages 272–277, 1993.

[59] Mara Muoz-Amatrian, Hamid Mirebrahim, Pei Xu, Steve I. Wanamaker, MingCheng Luo, Hind Alhakami, Matthew Alpert, Ibrahim Atokple, Benoit J. Batieno, Ousmane Boukar, Serdar Bozdag, Ndiaga Cisse, Issa Drabo, Jeffrey D. Ehlers, Andrew Farmer, Christian Fatokun, Yong Q. Gu, Yi-Ning Guo, Bao-Lam Huynh, Scott A. Jackson, Francis Kusi, Cynthia T. Lawley, Mitchell R. Lucas, Yaqin Ma, Michael P. Timko,

Jiajie Wu, Frank You, Noelle A. Barkley, Philip A. Roberts, Stefano Lonardi, and Timothy J. Close. Genome resources for climate-resilient cowpea, an essential crop for food security. *The Plant Journal*, 89(5):1042–1054, March 2017.

[60] Eugene W. Myers, Granger G. Sutton, Art L. Delcher, Ian M. Dew, Dan P. Fasulo, Michael J. Flanigan, Saul A. Kravitz, Clark M. Mobarry, Knut H. J. Reinert, Karin A. Remington, Eric L. Anson, Randall A. Bolanos, Hui-Hsien Chou, Catherine M. Jordan, Aaron L. Halpern, Stefano Lonardi, Ellen M. Beasley, Rhonda C. Brandon, Lin Chen, Patrick J. Dunn, Zhongwu Lai, Yong Liang, Deborah R. Nusskern, Ming Zhan, Qing Zhang, Xiangqun Zheng, Gerald M. Rubin, Mark D. Adams, and J. Craig Venter. A Whole-Genome Assembly of Drosophila. *Science*, 287(5461):2196–2204, March 2000.

[61] Giuseppe Narzisi and Bud Mishra. Comparing De Novo Genome Assembly: The Long and Short of It. *PLOS ONE*, 6(4):e19175, April 2011.

[62] Jurgen Nijkamp, Wynand Winterbach, Marcel van den Broek, Jean-Marc Daran, Marcel Reinders, and Dick de Ridder. Integrating genome assemblies with MAIA. *Bioinformatics*, 26(18):i433–i439, September 2010.

[63] Mihai Pop, Daniel S. Kosack, and Steven L. Salzberg. Hierarchical Scaffolding With Bambus. *Genome Res.*, 14(1):149–159, January 2004.

[64] José Ignacio Requeno and José Manuel Colom. Compact representation of biological sequences using set decision diagrams. In *6th International Conference on Practical Applications of Computational Biology & Bioinformatics*, number 154 in Advances in Intelligent and Soft Computing, pages 231–239. Springer Berlin Heidelberg, January 2012.

[65] Roy Ronen, Christina Boucher, Hamidreza Chitsaz, and Pavel Pevzner. SEQuel: improving the accuracy of genome assemblies. *Bioinformatics*, 28(12):i188–i196, June 2012.

[66] Steven L. Salzberg, Adam M. Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J. Treangen, Michael C. Schatz, Arthur L. Delcher, Michael Roberts, Guillaume Marais, Mihai Pop, and James A. Yorke. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Res.*, 22(3):557–567, March 2012.

[67] Kyriakos N. Sgarbas, Nikos D. Fakotakis, and George K. Kokkinakis. Optimal insertion in deterministic DAWGs. *Theoretical Computer Science*, 301:103–117, May 2003.

[68] Detlef Sieling and Ingo Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, 48(3):139–144, November 1993.

[69] Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.*, 22(3):549–556, March 2012.

[70] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J. M. Jones, and nan Birol. ABySS: A parallel assembler for short read sequence data. *Genome Res.*, 19(6):1117–1123, June 2009.

[71] Daniel D. Sommer, Arthur L. Delcher, Steven L. Salzberg, and Mihai Pop. Minimus: a fast, lightweight genome assembler. *BMC Bioinformatics*, 8(1):64, February 2007.

[72] Hayssam Soueidan, Florence Maurier, Alexis Groppi, Pascal Sirand-Pugnet, Florence Tardy, Christine Citti, Virginie Dupuy, and Macha Nikolski. Finishing bacterial genome assemblies with Mix. *BMC Bioinformatics*, 14(15):1–11, October 2013.

[73] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.

[74] Francesco Vezzi, Federica Cattonaro, and Alberto Policriti. e-RGA: enhanced Reference Guided Assembly of Complex Genomes. *EMBnet.journal*, 17(1):pp. 46–54, August 2011.

[75] Riccardo Vicedomini, Francesco Vezzi, Simone Scalabrin, Lars Arvestad, and Alberto Policriti. GAM-NGS: genomic assemblies merger for next generation sequencing. *BMC Bioinformatics*, 14(Suppl 7):S6, April 2013.

[76] Daolong Wang, Mario Lauria, Bo Yuan, and Fred A. Wright. Mega Weaver: A Simple Iterative Approach for BAC Consensus Assembly. In *Proceedings of the Second Conference on Asia-Pacific Bioinformatics - Volume 29*, APBC '04, pages 145–153, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[77] Ren L. Warren, Granger G. Sutton, Steven J. M. Jones, and Robert A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, February 2007.

[78] Alejandro H. Wences and Michael C. Schatz. Metassembler: merging and optimizing de novo genome assemblies. *Genome Biology*, 16(1):207, September 2015.

[79] H.E. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1):63–78, 2002.

[80] Erik S. Wright. DECIPHER: harnessing local sequence context to improve protein multiple sequence alignment. *BMC Bioinformatics*, 16:322, 2015.

[81] Guohui Yao, Liang Ye, Hongyu Gao, Patrick Minx, Wesley C. Warren, and George M. Weinstock. Graph accordance of next-generation sequence assemblies. *Bioinformatics*, 28(1):13–16, January 2012.

[82] Sungroh Yoon and G. De Micheli. An application of zero-suppressed binary decision diagrams to clustering analysis of DNA microarray data. In *26th Annual International Conference of the IEEE Engineering in Medicine & Biology Society, 2004. IEMBS '04*, volume 2, pages 2925–2928, 2004.

[83] Sungroh Yoon, Christine Nardini, Luca Benini, and Giovanni De Micheli. Discovering coherent biclusters from gene expression data using zero-suppressed binary decision diagrams. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 2(4):339–354, 2005.

[84] Daniel R. Zerbino. Using the Velvet de novo assembler for short-read sequencing technologies. *Curr Protoc Bioinformatics*, CHAPTER:Unit–11.5, September 2010.

[85] Daniel R. Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18(5):821–829, May 2008.

[86] Zheng Zhang, Scott Schwartz, Lukas Wagner, and Webb Miller. A Greedy Algorithm for Aligning DNA Sequences. *Journal of Computational Biology*, 7(1-2):203–214, February 2000.

[87] Aleksey V. Zimin, Guillaume Marais, Daniela Puiu, Michael Roberts, Steven L. Salzberg, and James A. Yorke. The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677, November 2013.

[88] Aleksey V. Zimin, Douglas R. Smith, Granger Sutton, and James A. Yorke. Assembly reconciliation. *Bioinformatics*, 24(1):42–45, January 2008.

Appendix A

A Comparative Evaluation of Assembly Reconciliation Tools: Supplementary Material

A.1 Experimental results on GAGE assemblies: comments on Tables A.1-A.15

A.1.1 High contiguity, high correctness inputs (GAGE)

In the first set of experiments, the objective was to explore the contiguity/correctness tradeoff. Specifically, we wanted to test the ability of reconciliation tools to take advantage of the contiguity of the first input assembly and the correctness of the second in order to create a merged assembly with a number of misassemblies comparable to the second assembly and a contiguity comparable to the first assembly. The two input assemblies to be merged were chosen so that one has high N50 value (but possibly a relatively high number of misassembly errors) and the other has few misassembly errors (and possibly a lower N50).

Table A.1 reports the results of merging the SOAPdenovo assembly (high N50) with the ABySS assembly (low misassembly errors) for the three chosen genomes. Since the assembly produced by ABySS on the *Rhodobacter sphaeroides* genome has more misassembly errors than the assembly generated by SOAPdenovo we also considered the results on *Rhodobacter sphaeroides* reported in Table A.5 where the input assemblies were produced by ALLPATHS-LG and SGA. The SOAPdenovo assembly was used as the "master" assembly in all tools that distinguish the assembly inputs.

Observe in Table A.1 that on the *Staphylococcus aureus* genome, all tools increase the contiguity by less than 3%, although the number of contigs decreased by $7 - 30\%$ (except for GAA). While none of the tools was able to improve assembly errors compared to the ABySS assembly, GAA and MIX produced more errors than SOAPdenovo. CISA produced the lowest number of misassemblies (13% less than SOAPdenovo) at the cost of a 4% decrease in genome and gene coverage. Otherwise, GAM_NGS and Metassembler maintained quality statistics close to that of SOAPdenovo.

In this and the rest of the experiments below, GAA consistently produced assemblies with predictable statistics. In the vast majority of the cases, GAA created a merged assembly in which the number of contigs, the size of the resulting assembly, and the number of misassemblies were very close to the sum of those statistics for the input assemblies. GAA's gene coverage was typically low in *Staphylococcus aureus* and *Rhodobacter sphaeroides* (not as much on *Hg_chr14*, where the gene coverage was generally high in comparison to other merged assemblies), while the percentage of covered genome was relatively high. While GAA's N50 was low, in terms of NGA50 the contiguity was at least as good as the most contiguous input assembly. In fact for *Hg_chr14*, GAA increased NGA50 by $19 - 123\%$ except for one case in which the increase was negligible.

When the input was composed of scaffolds, all tools improved contiguity by less than 5%, and reduced the number of scaffolds by $12 - 92\%$, with GARM reporting the highest decrease. GARM was the only tool that significantly increased N50 and produced the lowest number of misassemblies; however, GARM's merged assembly covered less than 40% of the reference sequence and less than one third of the genes. In contrast, MIX's merged assembly covered 94% of the genes despite (i) including only about 44% of the reference genome and (ii) decreasing the contiguity by 48%. If we exclude the number of contigs and NGA50, all the other assembly statistics for GAM_NGS and Metassembler are very similar to SOAPdenovo. None of the tools was able to reduce the number of misassembly errors compared to ABySS; in fact, CISA and MIX produced more errors than SOAPdenovo.

Despite the fact that ABySS's assembly for *Rhodobacter sphaeroides* had a higher number of misassembly errors than SOAPdenovo, none of the merged assemblies improved on the number of misassemblies compared to SOAPdenovo. Except for GAA, the number of misassembly errors produced by all tools were closer to the master (SOAPdenovo). As expected, tools that rely on the master assembly had a lower number of misassemblies than those that did not rank the inputs. With scaffolds as inputs, changes in NGA50 were negligible for all tools except for CISA. With contigs as inputs, GAM_NGS improved the contiguity by at most 11%, Metassembler and MIX increased it by 2%, and CISA dropped it by 85%. CISA also increased the number of contigs by 18%, and decreased genome and gene coverage by about 45%. GAM_NGS's assembly covered less than one quarter of the genome and about one fifth of the genes sequences, but its output had quality statistics similar to SOAPdenovo (with a 5% decrease in scaffolds). MIX and Metassembler decreased the number of scaffolds by 30% and 39%, respectively; otherwise, they maintained contiguity and coverage statistics within 1% of SOAPdenovo. GARM significantly improved the contiguity in terms of N50 but maintained the same NGA50 as SOAPdenovo. GARM decreased genome and gene coverage by 11%.

With contigs as inputs, GAM_NGS maintained the same genome and gene coverage as SOAPdenovo. MIX and Metassembler produced comparable results, namely (i) they both reduced the number of contigs by nearly one quarter, (ii) increased N50 by 10%, (iii) maintained the same genome coverage, and (iv) decreased gene coverage by less than 2%.

In the majority of the cases, experimental results obtained with ALLPATHS-LG (high N50) and SGA (low misassembly errors) on the *Rhodobacter sphaeroides* genome (reported in Table A.5) followed similar patterns to the ones we observed in Table A.1. CISA increased the number of contigs, but decreased the contiguity, genome and gene coverage (although the reduction was far less this time). GAA followed the same general pattern mentioned earlier. GAM_NGS did not increase contiguity but rather maintained it close to that of the master assembly. Metassembler and MIX also did not increase contiguity, but they reduced the number of contigs, as well as genome and gene coverage. ZORRO worked for this experiment: it increased the number of contigs, decreased contiguity by 10%, but retained genome and gene coverage of ALLPATHS-LG. ZORRO's merged assembly is the only one that achieved a smaller number of misassembly errors than ALLPATHS-LG (but still higher that SGA).

With scaffolds as input assemblies, CISA again reduced the number of contigs and produced an assembly with low genome and gene coverage. GAM_NGS reduced the number of contigs slightly but retained the quality statistics of the master assembly. Observe in Table A.1 that GARM improved N50 by 57% although it retained NGA50 close to SOAPdenovo (the master assembly). Observe in Table A.5 that GARM maintained ALLPATHS-LG's contiguity statistics. In both experiments GARM decreased genome and gene coverage; on the positive side, the consensus assembly had about 85% less scaffolds compared to the master.

Experimental results on the *Hg_chr14* with contigs as input assemblies (Table A.1), show that (i) GAM_NGS slightly improved contiguity, (ii) Metassembler maintained contiguity with fewer contigs, (iii) GAA crashed, (iv) number of misassemblies were closer to SOAPdenovo. With scaffolds as inputs, GARM drastically reduced the number of contigs, but also decreased the genome coverage by 7%. GAM_NGS and Metassembler produced assemblies with quality statistics close to SOAPdenovo except for a 26% decrease in the number of contigs for Metassembler.

A.1.2 Reordering the inputs (GAGE)

As mentioned above, some of the assembly reconciliation tools assume that the first input assembly is the master assembly, and should be "trusted" more (we call these tools *asymmetric*). The goal of this set of experiments is determine how the quality of the merged assembly depends on the specific order of the inputs.

To determine how the ranking affected the results, we repeated the same experiments reported in the previous section but switched the order of the inputs. A comparative analysis of the results in Table A.1 and Table A.2 prompts a few observations. First, we note that CISA, MIX, and GARM are *symmetric* (i.e., they do not require users to rank the inputs, see Table 3.1), hence they are expected to be unaffected by the reordering. Experimental results confirm that CISA and GARM are indeed unaffected. The reordering however affected MIX results, albeit only slightly.

For *Staphylococcus aureus*, MIX's contiguity statistics (N50 and NGA50) and genome coverage were not affected by the reordering of the inputs. However, we observed (i) a 2% decrease in gene coverage, (ii) a small difference in the number of contigs (± 1), and (iii) a small change in the number of misassemblies, although still higher than SOAPdenovo in both cases.

On the *Rhodobacter sphaeroides* genome, all statistics remained unchanged except for the number of misassemblies that increased after reordering. In addition, with contigs as inputs we did not observe an increase in NGA50 after the reordering.

Despite the fact that GAA requires input ranking, the results for *Staphylococcus aureus* and *Rhodobacter sphaeroides* were similar. The output statistics of GAA followed the general pattern mentioned in the previous section. For *Hg_chr14*, GAA crashed in one ordering but not on the other. For all three genome, GAM_NGS and Metassembler produced consensus assemblies with quality statistics close to the master assembly.

Note that the merged assemblies have higher contiguity in Table A.1, in which the master has higher N50. In contrast, the number of misassemblies were lower in Table A.2 for both *Staphylococcus aureus* and *Hg_chr14* in which the master had lower errors (with the exception of MIX). Merged assemblies for *Rhodobacter sphaeroides* had higher contiguity and lower number of misassemblies, in which the master had higher N50 and lower number of misassemblies (see Table A.1).

A.1.3 High-quality inputs (GAGE)

In the third set of experiments we tested the ability of the reconciliation tools to merge two high quality assemblies. We selected two highly contiguous assemblies (i.e., small number of contigs and scaffolds, high N50 values) and low number of misassembly errors. Table A.3 show the result of merging assemblies produced by ALLPATHS-LG as first input and either MSR-CA, SOAPdenovo, or CABOG as the second assembly.

Observe that for *Staphylococcus aureus* with contigs as inputs, GAM_NGS produced an improved assembly that (i) had no misassemblies, (ii) was 66% more contiguous, and (iii) covered the same portions of the genome and the genes. The next best assembly was by Metassembler with a 107% increase in contiguity and a 51% decrease in the number of contigs, but it had a slight increase in the number of misassemblies compared to ALLPATHS-LG. MIX also improved the contiguity by 107% (N50), but due to the high number of misassemblies (higher than MSR-CA) the increase in contiguity dropped to 4% when aligned to the reference. MIX's gene coverage also dropped by 37%. CISA improved contiguity by 11%, and reduced the number of contigs by nearly a half, but it produced a number of errors higher than ALLPATHS-LG. CISA also decreased genome and gene coverage. ZORRO decreased contiguity by 30% and increased the number of contigs by 22%, although it maintained genome and gene coverage.

With scaffolds as inputs, ALLPATHS-LG has no misassemblies, a lower N50 than MSR-CA but higher NGA50. In general, asymmetric tools produced a lower number of misassemblies and decreased the N50. For instance, GAM_NGS maintained quality statistics of ALLPATHS-LG. Although ZORRO is asymmetric it decreased contiguity by more than 90%. On the other hand, symmetric tools had a higher number of misassemblies. GARM achieved the highest increase of NGA50 (16%).

The contiguity of the merged assemblies improved $11\% - 108\%$ with the exception of ZORRO, which decreased the contiguity by 30%. GARM increased contiguity the most (108%) at the expense of (i) an additional 12% duplication rate, (ii) a number of misassemblies close to MSR-CA, and (iii) a 10% decrease in gene coverage. MIX introduced no misassemblies, but covered only 25% of the genome and gene sequences. Notably, both GAM_NGS and Metassembler (i) improved contiguity by 66.5%, (ii) reduced the number of contigs, (iii) introduced no misassemblies, (iv) and maintained gene coverage. These are two rare examples in which we observed an unquestionable improvement in the merged assembly.

On the *Rhodobacter sphaeroides* genome, the two input assemblies had almost the same number of misassemblies but the assembly produced by SOAPdenovo was much less fragmented. Only MIX, Metassembler and GARM increased N50 by 37%, 43%, and 69%, respectively (with only Metassembler increasing NGA50 significantly). All other tools decreased the contiguity. In terms of correctness, ZORRO and CISA (using scaffolds as inputs) reduced the number of misassemblies but also decreased the contiguity by 99% and 60%, respectively. Other tools produced merged assemblies with a number of misassemblies not better than the inputs.

GARM improved the contiguity by 38% while CISA increased it by less than 2%. GARM, CISA, and MIX reduced the number of contigs by 48%, 51%, and 60%, respectively, but also decreased genome and gene coverage. MIX is the only tool that reduced the number of misassemblies, but again its assembly only covered about half of the genome. None of the tools improved both contiguity and the number of misassemblies.

In *Hg_chr14*, GAA decreased the contiguity by 8%, but it improved the NGA50 by 76%, and increased the gene coverage by 13%. Nevertheless, it had a 198% inflation rate and produced a number of misassemblies equal to the sum of the number of misassemblies in the two inputs. GAM_NGS reduced the number of contigs by 10%, improved the contiguity (39% increase in N50, 28% increase in NGA50), slightly reduced the number of misassemblies, but decreased the gene coverage by 11%. Metassembler produced quality statistics that are very close to ALLPATHS-LG.

With scaffolds as inputs, GAM_NGS and Metassembler maintained similar quality statistics to ALLPATHS-LG, with the exception of the number of contigs (Metassembler decreased it by 33%) and gene coverage (GAM_NGS and Metassembler decreased by 18% and 51%, respectively). GARM improved N50 but decreased NGA50 by 9%. It also increased the number of misassemblies and decreased genome and gene coverage.

GARM improved the contiguity by 128% and reduced the number of contigs in half at the cost of 14% inflation and about 41% increase in the number of misassemblies.

GAA and GAM_NGS improved the contiguity by 76% and 28%, but only GAA increased the gene coverage.

A.1.4 Highly-fragmented inputs (GAGE)

The goal of this set of experiments was to evaluate the performance of assembly reconciliation tools when provided with two highly fragmented input assemblies. Input assemblies were selected to have a high percentage of contigs shorter than 200 bps, a high number of contigs and scaffolds, and low N50.

Table A.4 shows the results of merging ABySS assembly and SGA assembly. Observe that when we used contigs as inputs, ABySS had a higher contiguity than SGA (except in *Hg_chr14*). The opposite, however, was observed when scaffolds were provided in input. In *Staphylococcus aureus* and *Rhodobacter sphaeroides* with contigs as inputs, all tools increased N50 except for GAA. In terms of NGA50, only asymmetric tools maintained or improved over NGA50 of the better input assembly (in *Staphylococcus aureus* we observed up to 8% increase, and up to 17% in *Rhodobacter sphaeroides*). However, in *Hg_chr14* (with contigs as inputs) only GAM_NGS improved the N50. In terms of NGA50, GAA produced a 123% increase over SGA, while GAM_NGS did not improve it over SGA, but it increased it 33% over ABySS.

With scaffolds as inputs, we observed a decrease in N50 except for MIX and GARM (when SGA inputs are scaffolds). MIX, GARM, and CISA are symmetric tools, hence they are expected to perform better than other tools when the non-master input has better quality. CISA, however, produced inferior results with scaffolds as inputs in most experiments. It turns out that CISA with default parameters break scaffolds into contigs when a scaffold contains more than ten consecutive occurrences of Ns. MIX and GARM enhanced or maintained N50 of SGA. In terms of NGA50, MIX maintained it, while GARM slightly decreased it compared to SGA (yet still higher than ABySS). The number of contigs decreased although it remained relatively high in the majority of the cases. CISA had more than 80% decrease in the number of contigs with scaffolds as inputs, but the genome coverage was poor. GARM reduced the number of contigs by $74 - 91\%$, regardless of the genome coverage.

A.1.5 De Bruijn vs. string graph assembly (GAGE)

Here we tested the effect of merging assemblies generated using different assembly strategies. Specifically, we merged an assembly generated by an assembler that uses a de Bruijn graphs with an assembly produced by an assembler based on the string graph. Table A.5 shows the result of merging an assembly produced by ALLPATHS-LG (based on the de Bruijn graph) with an assembly produced by SGA (based on the string graph). Overall, GAM_NGS, Metassembler, and MIX maintained similar assembly statistics as ALLPATHS-LG.

Note that *Staphylococcus aureus* input assemblies (as contigs) had only one misassembly. The merged assemblies also have one misassembly, with the exception of GAA (two) and ZORRO (none). ZORRO corrected the assembly error without affecting N50 but at the price of a 17% increase in the number of contigs. CISA also increased the number of contigs, decreased NGA50 by 49%, and decreased the gene coverage by 15%. With scaffolds as inputs, ALLPATHS-LG's assembly has no assembly errors. In fact, observe that all merged assemblies did not have any misassemblies. GARM produced only 3 scaffolds and increased N50 by 31% but kept NGA50 close to ALLPATHS-LG, while decreasing less than 6% of genome and gene coverage. CISA covered less than 40% of the genome, while ZORRO decreased the contiguity by 99%.

On *Rhodobacter sphaeroides* with contigs as inputs, CISA and ZORRO decreased the contiguity by 34% and 10%, respectively. CISA decreased genome and gene coverage by 8%, while ZORRO maintained ALLPATHS-LG's coverage. GAM_NGS and Metassembler slightly reduced the number of contigs; otherwise they maintained ALLPATHS-LG's quality statistics. All tools produced a relatively high number of misassemblies (similar to ALLPATHS-LG). With scaffolds as inputs, CISA, ZORRO, and GARM's assembly statistics followed the same of statistics of *Staphylococcus aureus*. All assemblies, with the exception of CISA and ZORRO, had a number of misassemblies closer to ALLPATHS-LG. CISA again covered less than one fifth of the genome and ZORRO decreased the contiguity by 99%. GARM produced only four contigs but decreased the genome coverage by less than 5%. GAM_NGS, Metassembler, and MIX produced consensus assemblies with quality statistics comparable to ALLPATHS-LG.

In *Hg_chr14* (with contigs as inputs) GAM_NGS and Metassembler reduced the number of contigs by 4% and 2%, respectively. GAM_NGS increased NGA50 by 2%. With scaffolds as inputs, GAM_NGS and Metassembler maintained assembly statistics close to ALLPATHS-LG except for the fact that Metassembler reduced the number of contigs by 21%. GARM reduced the number of contigs by 83%, maintained genome and gene coverage but increased the number of misassemblies by 9% (compared to ALLPATHS-LG) and decreased NGA50 by 9%.

GARM increased contiguity by 58%, while other tools improved it by less than 3%. GAM_NGS and Metassembler produced about the same number of misassembly errors as the higher of the two inputs. GARM improved NGA50 the most, but also increased the number of misassemblies by 42% and had 31% inflation rate.

A.1.6 Multiple inputs (GAGE)

In this set of experiments we tested the ability of the tools to merge more than two assemblies. When an assembly reconciliation tool allowed no more than two assemblies in input (see Table 3.1 for a list), we merged them in an iterative fashion. For instance, to merge three assemblies, we first merged two assemblies, then merged the result to the third assemblies. Metassembler uses a similar strategy: when the user provides multiple assemblies the tool iteratively performs pairwise reconciliation, where the output of one iteration is the input of the next. The ordering of the input assemblies was chosen based on *feature response* curve (FR curve), which is an assembly quality metric proposed in [61]. The FR curve represents the dependency between contigs that contains no more than τ features and the corresponding genome coverage. The x-axis represents τ and the y-axis represent genome coverage: the “steeper” is the curve, the better is the assembly. We used the FR curves in [75] to determine the merging order of the GAGE assemblies, starting with the assemblies with highest quality. Results for an alternative ordering is discussed in the next section. For tools that allowed to merge more than two assemblies (e.g., CISA and MIX), the merging was done in one step from the original assemblies. Here we were interested in measuring the contiguity and correctness of the resulting assemblies as the number of input assemblies increases.

Tables A.6, A.7, and A.8; in addition to Figure 3.7, and Figures 3.9 and 3.11, show the experimental results for *Staphylococcus aureus*, *Rhodobacter sphaeroides* and *Hg_chr14*, respectively, when inputs are contigs. First observe that in several cases, the process of iterative merging did not complete.

On *Staphylococcus aureus* and *Rhodobacter sphaeroides*, CISA generally improved the contiguity and decreased the number of contigs as the number of merged assemblies increased. The number of errors and the percentage of genome covered fluctuated over the iterations. As the number of merged assemblies increased, CISA increased the duplication rate and decreased the percentage of covered genes. GAA did not produce assembly files for the first iteration. Although GAA did not work for this particular ordering it did produce results for the alternative ordering reported in the next section.

In *Staphylococcus aureus* and *Rhodobacter sphaeroides*, GAM_NGS's contiguity improved over successive iterations, but the number of misassemblies errors did not decrease (it stayed close to the first master input in all iterations). On the positive side, (i) the number of contigs was relatively small and (ii) the percentage of genome covered was relatively high, and (iii) gene coverage was relatively high, although slightly lower than the best gene coverage in the input assemblies. In contrast, the percentage of gene coverage decreased for *Hg_chr14*. Although the genome coverage and contiguity were high, the number of misassemblies was also relatively high. GAM_NGS increased NGA50 by at least 70% compared to CABOG.

In *Staphylococcus aureus*, Metassembler's contiguity improved and the number of contigs decreased over successive iterations, but the number of misassemblies also increased.

Metassembler maintained high genome and gene coverage, although slightly lower than the best gene coverage in the input assemblies. In *Rhodobacter sphaeroides*, Metassembler's assembly did not improve after the forth iteration. Note that NGA50 was lower than BAMBUS2 and SOAPdenovo. Metassembler's assembly had low genome and gene coverage and number of misassemblies was about the average of the inputs. In *Hg_chr14*, the number of contigs and misassembly errors were low and decreased over successive iterations. Contiguity, genome and gene coverage were high, but slightly decreased over successive iterations.

MIX maintained a low number of misassemblies in most iterations but suffered from low genome and gene coverage. Also, NGA50 was relatively poor. Since the genome coverage in most iterations was less than 50% of the reference, no NGA50 was reported for those iteration. On the *Staphylococcus aureus* genome, the coverage was less than 50% in all iterations but it steadily improved with increasing number of inputs. On *Rhodobacter sphaeroides*, the genome coverage was below 50% with four or more inputs.

ZORRO frequently failed to produce results. When it worked, it increased genome and gene coverage. Contiguity usually started high, then fluctuated over iterations. ZORRO produced relatively high number of contigs and misassemblies (somewhat in between the values of the inputs).

We repeated the same experiment but with scaffolds as inputs. Results are reported in Tables A.9, A.10, and A.11 and Figures 3.8, 3.10, and 3.12. CISA's results show that after a certain number of input assemblies, increasing the number of inputs did not affect the results significantly. From that point forward, it generally improved the contiguity and reduced the number of contigs as the number of merged assemblies increased, at the cost of decreased genome and gene coverage and about 25% inflation rate. The number of misassemblies were with the range of input assemblies. CISA reached stability with four inputs on *Staphylococcus aureus* and three inputs on *Rhodobacter sphaeroides*).

MIX maintained a low number of contigs albeit this number fluctuated in *Rhodobacter sphaeroides* with increasing number of inputs. MIX also produced a high duplication ratio. On *Staphylococcus aureus*, MIX produced a high number of misassemblies which generally increased as the number of inputs increased. It maintained high genome coverage but gene coverage was poor in comparison to the inputs. It also maintained high contiguity except for the last iteration. On *Rhodobacter sphaeroides*, the number of misassemblies were also relatively high but it fluctuated as the number of inputs increased. Genome coverage increased steadily but gene coverage decreased. It also maintained high contiguity, achieving the best NGA50 for less than five inputs.

ZORRO produced a high number of contigs and a low number of misassemblies on *Staphylococcus aureus* and *Rhodobacter sphaeroides.* It maintained a high genome coverage but it slightly decreased gene coverage. Contiguity was poor and generally decreased over successive iterations.

GAM_NGS maintained results very close to the first input throughout all iterations on *Staphylococcus aureus*, *Rhodobacter sphaeroides*, and *Hg_chr14*. In the latter genome, GAM_NGS contiguity generally improved in successive iterations but so did the number of misassemblies.

Metassembler maintained similar quality statistics to CABOG on *Hg_chr14*, although the number of contigs slightly decreased over successive iteration. On *Rhodobacter* *sphaeroides*, Metassembler also maintained CABOG's quality statistics with a slight decrease of (i) number of contigs, (ii) number of misassemblies, (iii) genome and gene coverage, and (iv) contiguity, as the number of iteration increased. On *Staphylococcus aureus*, Metassembler also maintained quality statistics close but not identical to MSR-CA. In general, Metassembler produced a small number of contigs. Also, as the number of inputs increased, the number of misassemblies slightly decreased and the contiguity slightly improved.

A.1.7 Multiple inputs (alternative ordering)

In this set of experiments we tested the ability of the tools to merge more than two assemblies on an alternative ordering to the FR curves used in the main Text. Recall that when an assembly reconciliation tool allowed no more than two assemblies in input (see Table 1 in the main text for a list), we merged them in an iterative fashion starting from the most contiguous assemblies (see main Text for more details)

Tables A.12, A.13, A.14, and A.15 show the experimental results for *Staphylococcus aureus*, *Rhodobacter sphaeroides* (two tables) and *Hg_chr14*, respectively on this alternative ordering. Figures 3.8 – 3.12 summerize the results with respect to contiguity and correctness.

First observe that similar to what we observed for the ordering based on FR curves, in many instances the process of iterative merging did not complete.

On *Staphylococcus aureus* and *Rhodobacter sphaeroides*, CISA generally increased the contiguity and decreased the number of contigs as the number of merged assemblies increased. The number of errors and the percentage of genome covered fluctuated over the iterations. While the percentage of covered genes peaked with three input assemblies, CISA increased the duplication rate as the number of merged assemblies increased. GAA instead increased contiguity, number of errors, and duplication rate and the percentage of covered genome fraction, as the number of merged assemblies increased.

In *Staphylococcus aureus*, *Rhodobacter sphaeroides*, and *Hg_chr14*, GAA produced a monotonic increase in duplication rate at successive iterations, while misassemblies seemed to be the union of those present in the input assemblies. GAA's contiguity did not increase over successive iterations, but the genome coverage was relatively high, while gene coverage which was very low in both *Staphylococcus aureus* and *Rhodobacter sphaeroides.*

GAM_NGS's contiguity increased over successive iterations, but the number of misassemblies did not decrease. On the positive side, the number of misassemblies was small and the percentage of genome covered was high. In *Staphylococcus aureus* and *Rhodobacter sphaeroides*, gene coverage was high, although slightly lower than the best gene coverage in the input assemblies. In contrast, the percentage of gene coverage decreased for *Hg_chr14*.

GARM increased the contiguity over successive iterations but also inflated the resulting assembly. The number of misassemblies and the genome/gene coverage fluctuated. The percentage of gene coverage decreased in *Hg_chr14*. In *Rhodobacter sphaeroides*, GARM crashed after the third iteration. Note that in the second iteration of *Staphylococcus aureus* only 26 contigs covered nearly 93% of the genome with 91% gene coverage, no misassemblies, and no inflation. In *Staphylococcus aureus*, Metassembler maintained a low error rate and NGA50 (with the exception of *Hg_chr14*) over successive iterations (although NGA50 was consistently low). In *Hg_chr14*, NGA50 was low and also decreasing over iterations.

In *Rhodobacter sphaeroides*, genome and gene coverage for Metassembler was low with respect to input assemblies.

MIX maintained a low number of misassemblies in most iterations but suffered from low genome and gene coverage. Its NGA50 fluctuated over successive iteration, but it was relatively poor. Since the genome coverage in some iterations is less than 50% of the reference, no NGA50 was reported for those iteration.

ZORRO frequently failed to produce results. When it worked, it increased the percentage of genome coverage and gene coverage and it did not increased duplication.

Table A.1: Contiguity-correctness experimental results. Assembly reconciliation tools are given in input two assemblies to merge, in which the first has high contiguity, the second has high correctness. The table reports on quality of merged assembly compared to the two input assemblies. Notes: (c) indicates that the assembly is composed of contigs, (s) indicates that the assembly is composed of scaffolds; all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Staphylococcus aureus (genome size 2,903,081 bp)													
SOAPdenovo (c)	70	518,710	2,897,432	288,184	31	2,027,905	23.00	2.45	0.07	98.55	1.01	150,794	96.42
ABySS (c)	247	125,049	3,631,245	25,084	5	22,399	12.39	0.85	7.79	97.27	1.29	29,198	77.53
CISA	49	483,164	2,794,831	212,755	27	1,804,073	22.37	2.65	7.55	94.72	1.02	154,503	92.36
GAA	317	518,710	6,528,677	48,845	36	2,050,304	27.14	2.33	4.37	98.88	2.28	150,794	4.32
GAM_NGS	65	545,577	2,901,216	288,184	31	2,053,193	26.28	2.59	0.07	98.58	1.02	152,795	96.40
MIX	56	540,299	2,944,556	294,927	36	2,195,378	24.66	2.62	0.07	98.61	1.03	152,795	94.57
Metassembler	50	528,561	2,893,344	288,184	31	2,080,336	22.41	2.24	1.11	98.52	1.01	154,116	96.00
SOAPdenovo (s)	64	518,710	2,902,967	331,598	32	2,348,756	24.23	2.76	167.31	98.55	1.01	172,575	96.39
ABySS (s)	206	130,192	3,692,703	27,695	10	178,901	12.22	1.09	1520.97	97.54	1.30	31,703	77.56
CISA	51	526,000	3,030,347	331,595	37	2,502,090	26.03	2.68	489.71	97.78	1.07	172,574	90.87
GAM_NGS	56	526,202	2,904,792	335,060	32	2,358,582	25.86	2.76	174.16	98.61	1.01	181,779	96.47
GARM (ctg_scf)	41	1,055,685	7,562,761	335,611	105	7,125,530	28.46	2.34	18,510.37	72.15	3.61	217,484	5.07
GARM (scf_ctg)	5	423,440	1,485,622	410,622	14	1,478,652	27.49	2.34	12,403.76	38.32	1.33	NA	33.28
MIX	45	542,391	2,974,002	356,570	35	2,442,687	27.93	3.11	191.93	98.69	1.04	177,880	93.92
Metassembler	43	529,535	2,902,261	331,598	32	2,358,551	23.29	2.59	193.95	98.56	1.01	181,779	96.01
Rhodobacter sphaeroides (genome size 4,603,060 bp)													
SOAPdenovo (c)	114	376,585	4,569,340	131,681	11	633,163	21.28	9.51	0.00	98.72	1.01	129,613	92.24
ABySS (c)	1509	54,734	4,830,769	5562	85	866,218	22.76	5.84	2.32	93.75	1.12	5303	76.12
CISA	135	157,113	2,635,836	60,566	23	429,493	45.18	11.06	0.57	55.40	1.03	19,060	49.52
GAA	1604	376,585	9,367,810	23,577	96	1,499,381	27.78	10.64	1.20	99.41	2.05	129,613	22.27
GAM_NGS	110	376,585	4,569,928	151,404	11	612,573	22.11	9.59	0.04	98.74	1.01	144,469	92.19
MIX	87	376,585	4,601,257	144,469	21	951,069	25.57	10.12	0.41	98.96	1.01	131,601	90.69
Metassembler	87	376,585	4,564,073	144,469	12	668,770	21.75	9.61	4.01	98.79	1.00	131,601	91.50
SOAPdenovo (s)	76	1,154,134	4,579,801	660,164	13	1,927,959	21.30	9.81	228.42	98.73	1.01	539,770	92.58
ABySS (s)	1352	87,855	4,968,921	8036	85	1,012,703	23.38	8.92	2302.47	94.21	1.14	7136	76.92
CISA	43	118,526	1,271,328	45,199	25	584,067	72.32	12.85	1964.87	23.67	1.17	NA	19.02
GAM_NGS	72	1,154,134	4,580,383	660,164	13	1,927,959	21.34	9.81	222.06	98.75	1.01	539,770	92.55
GARM (ctg_scf)	82	885,089	14,770,552	340,523	121	13,355,236	41.34	30.74	14,620.98	81.61	3.93	300,232	18.83
GARM (scf_ctg)	11	1,160,202	4,086,623	1,035,290	17	3,379,068	41.14	31.14	208.88	87.97	1.01	540,088	83.34
MIX	53	2,189,067	4,622,653	665,695	20	3,370,373	23.05	9.93	263.61	98.86	1.02	539,770	91.39
Metassembler	46	1,158,092	4,580,689	665,622	15	1,961,085	21.67	9.95	280.55	98.73	1.01	538,783	91.76
Homo sapiens, chromosome 14 (genome size 107,349,540 bp)													
SOAPdenovo (c)	15,028	147,494	90,398,734	16,179	6329	43,713,769	152.34	24.26	0.02	77.30	1.09	8155	64.03
ABySS (c)	32,050	30,053	67,074,140	3182	24	128,244	84.48	9.20	1.31	61.54	1.01	1319	83.85
GAA	Did not produce output files												
GAM_NGS	14,755	147,494	90,399,807	16,649	6281	44,141,293	152.57	24.43	0.09	77.36	1.09	8345	63.60
Metassembler	12,331	147,494	87,578,779	16,797	6128	43,411,953	148.18	23.98	0.02	76.38	1.07	8155	63.30
SOAPdenovo (s)	7264	1,849,511	100,880,746	381,286	8171	89,396,625	152.68	24.52	10,166.39	77.44	1.20	17,851	31.73
ABySS (s)	31,582	30,053	67,724,594	3355	37	237,738	84.67	9.37	859.44	61.61	1.02	1339	83.10
GAM_NGS	7166	1,849,511	100,347,539	368,318	8019	88,844,125	152.81	24.64	10,151.99	76.89	1.20	17,903	29.85
GARM (ctg_scf)	435	139,253	7,402,130	27,562	521	5,362,430	191.29	31.20	8979.09	5.40	1.25	NA	11.18
GARM (scf_ctg)	949	1,853,709	92,980,399	427,952	7958	89,041,639	159.13	25.85	10,863.86	72.25	1.18	17,783	22.95
Metassembler	5358	1,849,511	98,820,840	387,994	8116	89,280,078	152.75	24.55	10,227.11	76.98	1.18	17,851	28.80

Table A.2: Contiguity-correctness experimental results. Assembly reconciliation tools are given in input the same two assemblies in Table A.1, but the order is swapped. The table reports on quality of merged assembly compared to the two input assemblies. Notes: (c) indicates that the assembly is composed of contigs, (s) indicates that the assembly is composed of scaffolds; all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Staphylococcus aureus (genome size 2,903,081 bp)													
ABySS (c)	247	125,049	3,631,245	25,084	5	22,399	12.39	0.85	7.79	97.27	1.29	29,198	77
SOAPdenovo (c)	70	518,710	2,897,432	288,184	31	2,027,905	23.00	2.45	0.07	98.55	1.01	150,794	96
CISA	49	483,164	2,794,831	212,755	27	1,804,073	22.37	2.65	7.55	94.72	1.02	154,503	92
GAA	316	518,710	6,524,766	48,845	36	2,050,304	27.21	2.33	4.37	98.88	2.27	150,794	4
GAM_NGS	171	280,341	3,696,361	39,384	20	739,372	15.26	0.99	6.41	97.73	1.30	69,239	78
MIX	57	522,777	2,893,314	294,927	33	2,086,001	23.65	2.59	0.10	98.59	1.01	152,795	96
Metassembler	51	524,335	2,945,348	179,402	28	1,642,287	22.90	2.35	25.09	98.23	1.03	154,503	94
ABySS (s)	206	130,192	3,692,703	27,695	10	178,901	12.22	1.09	1520.97	97.54	1.30	31,703	77
SOAPdenovo (s)	64	518,710	2,902,967	331,598	32	2,348,756	24.23	2.76	167.31	98.55	1.01	172,575	96
CISA	51	526,000	3,030,347	331,595	37	2,502,090	26.03	2.68	489.71	97.78	1.07	172,574	90
GAM_NGS	90	474,493	3,626,574	130,192	28	1,841,289	21.65	2.14	1416.24	98.18	1.27	123,783	78
GARM (ctg_scf)	41	1,055,685	7,562,761	335,611	105	7,125,530	28.46	2.34	18,510.37	72.15	3.61	217,484	5
GARM (scf_ctg)	5	410,638	1,445,727	383,625	14	1,438,757	23.97	2.43	9991.24	38.32	1.30	NA	33
MIX	44	524,869	2,923,103	356,570	37	2,405,808	26.96	2.93	209.50	98.66	1.02	177,880	95
Metassembler	63	410,149	3,395,040	172,462	28	1,443,578	24.03	2.52	951.56	98.34	1.19	155,925	84
Rhodobacter sphaeroides (genome size 4,603,060 bp)													
ABySS (c)	1509	54,734	4,830,769	5562	85	866,218	22.76	5.84	2.32	93.75	1.12	5303	76
SOAPdenovo (c)	114	376,585	4,569,340	131,681	11	633,163	21.28	9.51	0.00	98.72	1.01	129,613	92
CISA	135	157,113	2,635,836	60,566	23	429,493	45.18	11.06	0.57	55.40	1.03	19,060	49
GAA	1622	376,585	9,398,024	23,115	96	1,499,381	27.78	10.64	1.19	99.41	2.05	129,613	22
GAM_NGS	704	158,412	4,953,882	22,244	86	991,638	28.82	10.10	2.28	96.57	1.11	22,244	80
MIX	87	382,294	4,606,966	144,469	26	1,500,909	26.77	10.04	0.41	98.86	1.01	129,613	90
Metassembler	196	190,287	4,841,862	65,553	77	2,242,683	28.67	12.22	42.63	98.27	1.07	58,769	88
ABySS (s)	1352	87,855	4,968,921	8036	85	1,012,703	23.38	8.92	2302.47	94.21	1.14	7136	76
SOAPdenovo (s)	76	1,154,134	4,579,801	660,164	13	1,927,959	21.30	9.81	228.42	98.73	1.01	539,770	92
CISA	43	118,526	1,271,328	45,199	25	584,067	72.32	12.85	1964.87	23.67	1.17	NA	19
GAM_NGS	1328	87,855	4,982,719	8240	85	1,022,301	22.80	9.49	2326.56	94.48	1.14	7266	76
GARM (scf_ctg)	82	885,089	14,770,552	340,523	121	13,355,236	41.34	30.74	14,620.98	81.61	3.93	300,232	18
GARM (ctg_scf)	11	1,160,145	4,086,020	1,035,309	17	3,378,435	38.95	31.10	207.46	87.95	1.01	540,075	83
MIX	53	2,194,776	4,628,362	665,695	22	4,041,777	22.83	10.06	263.29	98.86	1.02	539,765	91
Metassembler	183	206,656	5,025,412	84,139	86	2,832,228	25.66	15.28	1664.02	98.22	1.11	82,101	86
Homo sapiens, chromosome 14 (genome size 107,349,540 bp)													
ABySS (c)	32,050	30,053	67,074,140	3182	24	128,244	84.48	9.20	1.31	61.54	1.01	1319	83
SOAPdenovo (c)	15,028	147,494	90,398,734	16,179	6329	43,713,769	152.34	24.26	0.02	77.30	1.09	8155	64
GAA	41,007	147,494	150,670,123	7865	6329	43,803,415	155.06	24.61	0.60	76.10	1.84	9733	79
GAM_NGS	17,579	146,388	76,617,304	11,218	2853	24,315,403	110.27	16.87	1.01	68.83	1.04	4216	67
Metassembler	31,909	30,053	66,662,156	3168	22	124,338	84.37	9.20	1.28	61.52	1.01	1291	77
ABySS (s)	31,582	30,053	67,724,594	3355	37	237,738	84.67	9.37	859.44	61.61	1.02	1339	83
SOAPdenovo (s)	7264	1,849,511	100,880,746	381,286	8171	89,396,625	152.68	24.52	10,166.39	77.44	1.20	17,851	31
GAM_NGS	24,913	183,079	76,225,840	5205	1425	11,646,550	102.35	13.54	4323.03	65.94	1.06	2278	76
GARM (ctg_scf)	435	139,253	7,402,130	27,562	521	5,362,430	191.29	31.20	8979.09	5.40	1.25	NA	11
Metassembler	31,467	30,053	67,494,412	3359	36	232,366	84.49	9.36	659.65	61.59	1.02	1334	76

Table A.3: Experimental results on merging high-quality assemblies. The table reports on quality of merged assembly compared to the two input assemblies. Notes: (c) indicates that the assembly is composed of contigs, (s) indicates that the assembly is composed of scaffolds; all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Staphylococcus aureus (genome size 2,903,081 bp)													
ALLPATHS-LG (c)	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	95.72
MSR-CA (c)	89	139,438	2,860,132	59,152	20	641,173	20.67	1.82	0.00	98.17	1.00	55,068	95.16
CISA	31	480,850	2,901,690	107,125	7	887,865	9.45	1.09	1.17	97.71	1.02	107,125	93.10
GAA	147	234,488	5,728,434	69,048	20	729,528	11.26	1.66	0.75	99.41	1.99	109,325	3.62
GAM_NGS	45	474,996	2,871,133	161,072	0	0	2.65	0.94	1.43	98.91	1.00	161,072	95.91
MIX	37	593,709	3,110,176	200,247	22	1,910,115	8.27	1.67	1.22	99.08	1.08	101,091	59.86
Metassembler	29	481,089	2,853,430	200,247	2	34,427	3.05	0.91	4.66	98.32	1.00	200,247	95.38
ZORRO	72	201,529	2,885,456	70,361	4	121,296	5.63	1.39	0.42	99.13	1.00	68,006	96.19
ALLPATHS-LG (s)	11	1,435,559	2,879,481	1,091,731	0	0	3.97	2.51	345.31	98.86	1.00	1,082,616	96.59
MSR-CA (s)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96.65
CISA	10	2,411,914	2,865,036	2,411,914	47	2,857,360	20.87	3.71	308.93	98.39	1.00	220,020	96.82
GAM_NGS	11	1,435,559	2,879,481	1,091,731	0	0	3.97	2.51	345.31	98.86	1.00	1,082,616	96.59
GARM (ctg_scf)	4	2,612,640	3,154,342	2,612,640	44	3,154,342	13.57	2.78	168.66	96.49	1.13	255,111	91.99
GARM (scf_ctg)	3	1,531,987	2,931,536	1,531,987	12	2,931,536	14.95	4.15	118.47	90.58	1.12	356,937	89.19
MIX	10	3,844,359	5,680,169	3,844,359	54	5,596,862	15.08	4.57	336.91	99.39	1.97	1,077,568	88.08
Metassembler	7	1,435,836	2,860,386	1,435,836	2	1,435,836	3.40	1.61	395.02	98.15	1.00	1,082,089	96.02
ZORRO	124	178,381	2,943,696	57,402	6	48,675	3.83	1.78	478.82	98.95	1.02	60,015	94.26
Rhodobacter sphaeroides (genome size 4,603,060 bp)													
ALLPATHS-LG (c)	203	106,467	4,587,354	42,455	10	404,185	6.33	4.77	2.79	99.20	1.00	41,487	92.55
SOAPdenovo (c)	114	376,585	4,569,340	131,681	11	633,163	21.28	9.51	0.00	98.72	1.01	129,613	92.24
CISA	56	423,736	4,826,390	131,681	13	824,841	15.78	8.86	0.79	97.86	1.07	131,601	84.89
GAA	316	376,585	9,152,896	67,208	21	1,037,348	14.43	7.03	1.40	99.53	2.00	129,613	6.01
GAM_NGS	147	259,855	4,589,116	65,618	14	515,689	12.13	5.73	2.68	99.26	1.00	60,502	92.61
MIX	79	400,670	4,962,938	180,222	29	2,386,110	24.31	9.21	0.60	98.86	1.09	131,601	85.54
Metassembler	65	446,066	4,588,876	187,667	17	1,358,099	11.75	6.78	47.24	99.32	1.00	187,667	93.03
ZORRO	291	93,996	4,603,056	27,399	12	183,417	11.31	4.63	1.02	99.09	1.01	27,233	91.83
ALLPATHS-LG (s)	33	3,192,334	4,608,763	3,192,334	15	4,447,871	5.91	6.79	467.31	99.24	1.01	928,821	95.02
SOAPdenovo (s)	76	1,154,134	4,579,801	660,164	13	1,927,959	21.30	9.81	228.42	98.73	1.01	539,770	92.58
CISA	32	1,331,947	1,824,146	1,331,947	12	1,660,662	23.74	6.44	238.36	39.44	1.01	NA	37.97
GAM_NGS	29	3,192,334	4,609,305	3,192,334	15	4,449,040	5.91	6.83	467.14	99.26	1.01	928,821	95.13
GARM (ctg_scf)	11	1,164,408	4,508,978	1,115,528	23	4,344,826	21.86	11.59	196.65	89.25	1.10	248,291	82.83
GARM (scf_ctg)	4	4,425,980	6,151,505	4,425,980	34	6,151,505	12.66	8.34	2128.16	94.53	1.41	358,682	91.36
MIX	28	3,192,334	4,720,129	3,192,334	14	4,550,518	6.08	6.87	456.15	99.27	1.03	928,821	95.01
Metassembler	30	3,191,637	4,606,469	3,191,637	16	4,446,954	6.48	6.83	421.10	99.29	1.01	929,418	95.04
ZORRO	272	117,853	4,626,121	32,447	10	134,705	9.08	5.86	434.79	99.05	1.01	32,447	92.30
Homo sapiens, chromosome 14 (genome size 107,349,540 bp)													
ALLPATHS-LG (c)	4469	240,773	84,416,102	38,359	109	1,384,277	67.71	21.79	54.60	78.48	1.00	27,772	62.80
CABOG (c)	3233	296,904	86,189,919	46,699	108	3,694,326	101.52	23.29	0.00	79.94	1.00	35,539	59.37
GAA	7687	296,904	170,593,729	42,872	217	5,078,603	94.38	23.82	27.02	80.36	1.98	62,647	70.82
GAM_NGS	2916	483,603	84,926,958	65,003	107	2,715,562	70.71	22.69	52.70	78.99	1.00	45,596	55.98
Metassembler	4362	240,773	84,241,180	38,473	109	1,432,983	67.61	21.80	54.53	78.37	1.00	27,830	62.73
ALLPATHS-LG (s)	174	81,646,936	87,646,728	81,646,936	455	87,219,645	66.85	22.87	3734.63	78.51	1.04	397,351	14.00
CABOG (s)	474	2,260,562	86,481,568	401,279	269	43,205,905	101.10	24.68	267.20	80.01	1.01	215,699	29.56
GAM_NGS	171	81,646,936	87,651,696	81,646,936	455	87,219,645	66.89	22.88	3734.42	78.51	1.04	397,351	21.43
GARM (ctg_scf)	297	2,558,470	96,444,763	501,662	544	77,192,101	91.38	24.79	3207.84	76.15	1.18	195,601	27.05
GARM (scf_ctg)	23	200,905,823	207,730,622	200,905,823	106	6,711,411	164.08	37.12	55,599.20	78.67	1.09	74,709	18.48
Metassembler	117	81,646,936	87,530,459	81,646,936	454	87,213,926	66.80	22.86	3731.03	78.46	1.04	397,351	14.37

Table A.4: Experimental results on merging highly fragmented assemblies. The table reports on quality of merged assembly compared to the two input assemblies. Notes: (c) indicates that the assembly is composed of contigs, (s) indicates that the assembly is composed of scaffolds; all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Staphylococcus aureus (genome size 2,903,081 bp)													
ABySS (c)	247	125,049	3,631,245	25,084	5	22,399	12.39	0.85	7.79	97.27	1.29	29,198	77.53
SGA (c)	985	16,870	2,748,664	4178	1	2431	1.02	0.11	0.00	94.44	1.00	4005	81.58
CISA	183	125,049	2,775,035	25,585	6	44,257	14.86	1.23	4.58	95.07	1.00	23,610	92.52
GAA	1225	125,049	6,361,589	10,544	6	24,830	11.82	0.77	4.45	97.93	2.24	29,198	27.27
GAM_NGS	223	125,383	3,634,133	27,931	5	22,399	13.19	0.92	7.79	97.38	1.28	31,393	77.45
MIX	176	125,049	2,922,239	27,931	14	217,753	12.57	1.60	9.68	97.03	1.04	27,666	92.69
Metassembler	203	125,049	3,480,529	28,148	4	19,126	11.55	0.82	11.58	97.19	1.23	31,180	77.72
ABySS (s)	206	130,192	3,692,703	27,695	10	178,901	12.22	1.09	1520.97	97.54	1.30	31,703	77.56
SGA (s)	299	286,534	3,051,005	149,421	3	113,622	1.09	11.47	9852.72	94.87	1.11	134,849	82.67
CISA	31	130,192	1,937,031	81,084	4	192,275	6.30	7.68	6307.75	54.72	1.22	42,752	45.91
GAM_NGS	203	130,192	3,667,521	27,917	10	178,901	12.22	1.09	1518.22	97.56	1.29	31,703	77.53
GARM (ctg_scf)	24	348,780	3,286,103	230,715	26	2,287,325	10.48	2.52	5161.68	93.00	1.15	124,186	86.50
GARM (scf_ctg)	53	248,986	1,603,258	46,033	25	736,257	32.17	6.48	4270.49	30.86	1.78	7132	19.50
MIX	278	316,106	3,167,023	149,421	10	801,229	3.55	11.31	9401.92	95.06	1.15	134,849	80.70
Metassembler	116	233,465	3,613,534	63,433	10	195,410	11.69	1.91	1587.67	97.28	1.28	87,846	76.71
Rhodobacter sphaeroides (genome size 4,603,060 bp)													
ABySS (c)	1509	54,734	4,830,769	5562	85	866,218	22.76	5.84	2.32	93.75	1.12	5303	76.12
SGA (c)	2173	29,520	4,188,432	2530	1	4048	5.70	2.47	0.00	90.70	1.00	2280	77.99
CISA	1343	54,734	4,586,141	5631	62	655,971	23.40	5.79	1.98	94.51	1.05	5082	80.07
GAA	3652	54,734	8,975,227	3515	86	870,266	22.41	5.41	1.25	95.96	2.03	6712	33.05
GAM_NGS	1394	54,734	4,843,138	5823	82	870,277	22.86	5.98	2.31	94.08	1.12	5626	77.03
MIX	1442	54,734	4,670,901	5638	111	964,749	19.87	7.28	2.21	93.72	1.08	4845	77.37
Metassembler	1466	54,734	4,792,900	5705	81	876,669	22.88	5.84	2.34	93.70	1.11	5435	75.43
ABySS (s)	1352	87,855	4,968,921	8036	85	1,012,703	23.38	8.92	2302.47	94.21	1.14	7136	76.92
SGA (s)	1208	148,756	5,328,387	44,205	3	31,764	5.86	7.22	21,499.94	90.77	1.26	17,695	77.90
CISA	154	96,941	3,878,079	28,615	28	535,930	17.65	11.64	19,413.30	59.62	1.41	15,332	49.80
GAM_NGS	1227	87,855	5,088,762	9076	85	1,029,168	22.28	9.29	4248.87	94.46	1.16	8029	77.09
GARM (ctg_scf)	106	166,599	4,632,346	68,824	186	4,110,632	20.05	12.48	18,414.95	73.91	1.36	17,133	65.89
GARM (scf_ctg)	178	87,870	1,912,210	15,350	52	634,591	30.78	17.47	5630.66	34.12	1.21	NA	29.65
MIX	1136	148,756	5,568,950	50,549	29	801,324	6.98	8.57	20,654.38	91.09	1.31	21,096	74.61
Metassembler	945	87,855	5,247,377	13,289	80	1,160,977	22.92	10.08	8078.86	94.37	1.20	11,657	77.55
Homo sapiens, chromosome 14 (genome size 107,349,540 bp)													
ABySS (c)	32,050	30,053	67,074,140	3182	24	128,244	84.48	9.20	1.31	61.54	1.01	1319	83.85
SGA (c)	33,695	30,350	75,492,807	3317	107	249,973	87.51	12.57	0.00	69.89	1.01	1945	88.96
GAA	65,737	30,350	142,552,348	3255	131	378,217	90.36	13.71	0.62	70.52	1.88	4336	83.60
GAM_NGS	28,291	52,608	69,320,650	3757	29	161,851	85.25	9.96	1.18	63.74	1.01	1753	82.70
Metassembler	31,853	30,053	66,646,160	3174	22	124,338	84.13	9.18	1.28	61.48	1.01	1293	76.87
ABySS (s)	31,582	30,053	67,724,594	3355	37	237,738	84.67	9.37	859.44	61.61	1.02	1339	83.10
SGA (s)	9586	551,622	88,557,645	82,616	170	8,811,457	87.77	18.34	14,499.38	70.47	1.16	35,317	34.97
GAM_NGS	16,084	168,748	80,212,474	15,869	115	1,945,058	86.43	14.31	10,009.71	66.58	1.11	4213	57.79
GARM (ctg_scf)	1751	555,426	82,819,623	91,759	277	18,242,778	88.50	17.64	15,041.66	64.98	1.17	34,725	26.63
GARM (scf_ctg)	627	30,127	4,114,216	8195	53	471,071	126.00	21.41	12,030.12	3.09	1.18	NA	NA
Metassembler	31,401	30,053	67,410,722	3359	34	209,172	84.30	9.34	649.81	61.55	1.02	1331	76.12

Table A.5: Experimental results on merging assemblies produced by assemblers based on the de Bruijn graph compared to string graph. The table reports on quality of merged assembly compared to the two input assemblies. Notes: (c) indicates that the assembly is composed of contigs, (s) indicates that the assembly is composed of scaffolds; all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Staphylococcus aureus (genome size 2,903,081 bp)													
ALLPATHS-LG (c)	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	95
SGA (c)	985	16,870	2,748,664	4178	1	2431	1.02	0.11	0.00	94.44	1.00	4005	81
CISA	65	208,210	2,438,736	69,908	1	30,003	3.19	0.78	1.44	84.14	1.00	49,357	81
GAA	1037	234,488	5,599,935	16,131	2	92,065	1.63	0.73	0.77	99.11	1.95	96,740	26
GAM_NGS	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	95
MIX	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	95
Metassembler	59	234,488	2,869,780	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	95
ZORRO	69	234,548	2,873,546	96,657	0	0	1.99	0.59	0.90	98.90	1.00	96,657	96
ALLPATHS-LG (s)	11	1,435,559	2,879,481	1,091,731	0	0	3.97	2.51	345.31	98.86	1.00	1,082,616	96
SGA (s)	299	286,534	3,051,005	149,421	3	113,622	1.09	11.47	9852.72	94.87	1.11	134,849	82
CISA	4	461,617	1,286,214	286,534	0	0	0.98	8.46	7240.24	38.69	1.15	NA	35
GAM_NGS	11	1,435,559	2,879,481	1,091,731	0	0	3.97	2.51	345.31	98.86	1.00	1,082,616	96
GARM (ctg_scf)	19	657,195	4,255,326	246,153	13	2,640,794	3.58	1.52	12,687.18	97.25	1.51	231,221	64
GARM (scf_ctg)	3	1,435,569	2,707,066	1,435,569	0	0	3.60	2.85	366.04	92.92	1.00	1,082,874	91
MIX	11	1,435,559	2,879,481	1,091,731	0	0	3.97	2.51	345.31	98.86	1.00	1,082,616	96
Metassembler	11	1,435,619	2,879,843	1,092,093	0	0	3.59	1.92	503.22	98.71	1.00	1,080,404	96
ZORRO	759	48,229	3,116,392	10,351	0	0	1.06	0.70	8684.11	97.73	1.00	10,768	87
Rhodobacter sphaeroides (genome size 4,603,060 bp)													
ALLPATHS-LG (c)	203	106,467	4,587,354	42,455	10	404,185	6.33	4.77	2.79	99.20	1.00	41,487	92
SGA (c)	2173	29,520	4,188,432	2530	1	4048	5.70	2.47	0.00	90.70	1.00	2280	77
CISA	233	106,467	4,239,822	31,549	10	373,275	6.61	4.79	2.81	91.67	1.00	27,262	85
GAA	2348	106,467	8,730,704	10,451	11	408,233	6.74	4.79	1.47	99.32	1.91	41,487	29
GAM_NGS	201	106,467	4,588,158	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	92
MIX	202	106,467	4,592,497	42,455	12	464,515	6.66	4.80	2.74	99.22	1.01	41,487	92
Metassembler	200	106,467	4,587,010	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	92
ZORRO	231	105,315	4,596,344	37,312	8	215,902	6.96	4.68	1.46	99.29	1.01	37,195	92
ALLPATHS-LG (s)	33	3,192,334	4,608,763	3,192,334	15	4,447,871	5.91	6.79	467.31	99.24	1.01	928,821	95
SGA (s)	1208	148,756	5,328,387	44,205	3	31,764	5.86	7.22	21,499.94	90.77	1.26	17,695	77
CISA	10	148,922	1,021,674	113,539	2	227,007	3.45	9.28	17,854.33	18.27	1.22	NA	17
GAM_NGS	31	3,192,334	4,609,567	3,192,334	15	4,452,322	6.19	6.78	467.22	99.26	1.01	928,821	95
GARM (ctg_scf)	65	238,957	6,690,621	134,426	71	5,275,835	9.44	7.40	16,050.80	86.27	1.69	73,449	51
GARM(scf_ctg)	4	3,192,749	4,371,411	3,192,749	12	4,371,411	8.48	8.87	479.98	94.29	1.01	929,045	91
MIX	32	3,192,334	4,783,485	3,192,334	16	4,496,001	6.30	6.81	1383.89	99.26	1.05	928,821	92
Metassembler	30	3,192,172	4,608,043	3,192,172	15	4,451,946	6.05	6.88	547.37	99.17	1.01	928,722	94
ZORRO	1970	105,319	5,706,683	3275	7	95,930	6.39	5.05	19,859.89	98.61	1.01	4944	82
Homo sapiens, chromosome 14 (genome size 107,349,540 bp)													
ALLPATHS-LG (c)	4469	240,773	84,416,102	38,359	109	1,384,277	67.71	21.79	54.60	78.48	1.00	27,772	62
SGA (c)	33,695	30,350	75,492,807	3317	107	249,973	87.51	12.57	0.00	69.89	1.01	1945	88
GAA	38,155	240,773	159,891,549	10,679	216	1,634,250	71.63	22.02	28.83	79.17	1.88	27,895	81
GAM_NGS	4283	240,773	84,466,433	39,449	110	1,663,046	68.05	21.79	54.51	78.53	1.00	28,458	62
Metassembler	4359	240,773	84,259,313	38,473	109	1,384,277	67.54	21.81	54.50	78.36	1.00	27,772	62
ALLPATHS-LG (s)	174	81,646,936	87,646,728	81,646,936	455	87,219,645	66.85	22.87	3734.63	78.51	1.04	397,351	14
SGA (s)	9586	551,622	88,557,645	82,616	170	8,811,457	87.77	18.34	14,499.38	70.47	1.16	35,317	34
GAM_NGS	174	81,646,936	87,646,728	81,646,936	455	87,219,645	66.85	22.87	3734.63	78.51	1.04	397,351	14
GARM (ctg_scf)	1186	671,738	123,685,540	148,610	1160	86,820,549	92.01	31.12	13,901.57	75.55	1.52	84,526	37
GARM (scf_ctg)	30	81,832,553	87,591,308	81,832,553	496	87,461,370	91.83	31.93	3459.35	78.17	1.04	359,840	NA
Metassembler	137	81,646,936	87,584,806	81,646,936	455	87,219,645	66.85	22.87	3737.20	78.50	1.04	397,351	14

Table A.6: Experimental results on merging more than two assemblies ordered by the FRCurve score (*Staphylococcus aureus*, genome size: 2,903,081 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation	Contigs	Largest	Size	N50	Misassembly	Misassembly	Mismatches	Indels	N's	Genome	Duplication	NGA50	Genes
Tool or Input	(#)	(bp)	(bp)	(bp)	(#)	Length (bp)	(#)	(#)	(#)	covered (%)	ratio	(bp)	(%)
Input 1 (MSR-CA)	89	139,438	2,860,132	59,152	20	641,173	20.67	1.82	0.00	98.17	1.00	55,068	95
Input 2 (ALLPATHS-LG)	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
Input 3 (BAMBUS2)	106	158,330	2,832,623	50,192	1	9411	1.41	6.95	0.35	97.66	1.00	50,192	95
Input 4 (SGA)	985	16,870	2,748,664	4178	1	2431	1.02	0.11	0.00	94.44	1.00	4005	82
Input 5 (Velvet)	128	169,214	2,837,036	52,792	9	284,379	13.51	1.69	0.00	97.73	1.00	48,149	97
Input 6 (SOAPdenovo)	70	518,710	2,897,432	288,184	31	2,027,905	23.00	2.45	0.07	98.55	1.01	150,794	96
Input 7 (ABySS)	247	125,049	3,631,245	25,084	5	22,399	12.39	0.85	7.79	97.27	1.29	29,198	78
CISA (1+2)	31	480,850	2,901,690	107,125	7	887,865	9.45	1.09	1.17	97.71	1.02	107,125	93
CISA (1+2+3)	26	384,401	2,888,221	139,438	9	1,106,426	6.22	2.12	1.45	92.43	1.08	139,438	84
CISA (1+2+3+4)	25	384,489	2,898,535	139,438	10	1,200,735	6.30	2.16	1.45	92.43	1.08	139,438	83
CISA (1+2+...+5)	24	390,086	2,869,762	171,871	8	976,622	5.26	1.87	1.43	90.39	1.09	149,829	81
CISA (1+2+...+6)	17	521,399	3,094,516	288,204	25	2,175,309	17.21	2.68	0.65	92.46	1.15	179,344	77
CISA (1+2+...+7)	17	521,399	3,094,993	288,204	27	2,286,203	17.25	2.72	0.87	92.48	1.15	179,344	77
GAA (1+2)						Did not produce an assembly file							
GAM_NGS (1+2)	57	254,779	2,862,666	83,138	19	816,715	20.33	1.82	0.21	98.27	1.00	77,039	95
GAM_NGS ((1+2)+3)	52	360,903	2,861,820	90,479	19	1,177,440	20.20	1.93	0.21	98.24	1.00	82,101	96
GAM_NGS (((1+2)+3)+4)	52	360,903	2,861,820	90,479	19	1,177,440	20.20	1.93	0.21	98.24	1.00	82,101	96
GAM_NGS (((1+2)+...)+5)	49	360,903	2,861,801	139,061	17	1,163,920	20.20	1.93	0.21	98.24	1.00	100,381	96
GAM_NGS (((1+2)+...)+6)	45	360,903	2,861,888	139,061	18	1,284,451	20.55	1.93	0.21	98.25	1.00	110,995	96
GAM_NGS (((1+2)+...)+7)	44	360,903	2,861,951	168,739	19	1,538,807	20.41	1.93	0.21	98.25	1.00	110,995	96
GARM (1+2)	16	833,732	3,469,519	625,562	7	2,559,729	14.53	3.69	0.09	93.38	1.28	325,955	65
GARM ((1+2)+3)	16	833,738	3,554,698	625,563	8	2,524,523	11.80	3.92	0.23	93.11	1.30	330,720	64
GARM (((1+2)+3)+4)	1	634,995	634,995	634,995	4	634,995	14.10	4.59	1.42	21.74	1.01	NA	23
GARM (((1+2)+...)+5)	2	625,576	634,987	625,576	3	634,987	17.90	5.07	0.00	21.74	1.01	NA	22
GARM (((1+2)+...)+6)	11	316,522	1,117,006	152,795	10	794,721	29.53	2.44	0.00	38.15	1.01	NA	39
GARM (((1+2)+...)+7)	79	316,549	2,768,200	239,944	24	1,756,442	31.13	1.72	4.84	43.93	2.17	152,817	16
Metassembler (1+2)	33	315,232	2,855,905	172,800	12	1,096,894	20.30	1.47	0.28	98.25	1.00	139,438	95
Metassembler ((1+2)+3)	29	356,601	2,856,200	186,644	13	1,207,911	19.99	1.54	0.28	98.24	1.00	149,559	95
Metassembler (((1+2)+3)+4)	29	356,601	2,856,200	186,644	13	1,207,911	19.99	1.54	0.28	98.24	1.00	149,559	95
Metassembler (((1+2)+...)+5)	29	356,601	2,856,454	186,644	13	1,208,165	19.99	1.54	0.28	98.24	1.00	149,559	95
Metassembler (((1+2)+...)+6)	19	356,601	2,862,874	263,604	17	1,846,745	22.70	1.86	0.35	98.33	1.00	203,830	95
Metassembler (((1+2)+...)+7)	19	356,601	2,868,370	263,604	17	1,846,745	22.70	1.86	0.35	98.33	1.01	203,830	95
MIX (1+2)	23	200,247	766,204	66,230	1	89,634	3.52	1.17	1.70	26.39	1.00	NA	26
MIX (1+2+3)	16	200,247	545,364	80,813	0	0	3.85	0.37	2.02	18.78	1.00	NA	20
MIX (1+2+3+4)	18	200,247	560,310	80,813	0	0	3.75	0.89	1.96	19.29	1.00	NA	20
MIX (1+2+...+5)	12	200,247	412,282	69,391	0	0	1.94	1.21	2.67	14.20	1.00	NA	15
MIX (1+2+...+6)	25	518,710	1,292,513	316,522	17	1,019,107	26.74	3.11	0.00	44.31	1.01	NA	44
MIX (1+2)+...+7)	22	518,710	1,280,637	316,522	16	1,014,066	21.77	2.66	0.00	43.98	1.00	NA	44
ZORRO (1+2)	71	201,636	2,885,411	70,361	4	121,394	11.92	2.12	0.49	99.14	1.00	68,006	96
ZORRO ((1+2)+3)	114	96,442	2,881,965	43,462	4	37,282	10.47	2.19	0.35	99.01	1.00	43,461	96
ZORRO (((1+2)+3)+4)	109	136,629	2,887,028	46,362	4	37,271	10.85	1.81	0.59	99.03	1.00	46,362	96
ZORRO (((1+2)+...)+5)						Produced an empty assembly file							

Table A.7: Experimental results on merging more than two assemblies (contigs) ordered by the FRCurve score (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Input 1 (MSR-CA)	377	83,726	4,458,952	23,575	18	151,790	23.43	4.90	0.00	96.14	1.01	21,236	89
Input 2 (ALLPATHS-LG)	203	106,467	4,587,354	42,455	10	404,185	6.33	4.77	2.79	99.20	1.00	41,487	93
Input 3 (CABOG)	318	88,519	4,236,663	22,044	11	276,929	29.07	5.48	0.00	91.91	1.00	19,076	87
Input 4 (BAMBUS2)	170	279,125	4,369,357	97,331	4	123,417	5.82	5.84	0.00	94.89	1.00	93,198	90
Input 5 (SOAPdenovo)	114	376,585	4,569,340	131,681	11	633,163	21.28	9.51	0.00	98.72	1.01	129,613	92
Input 6 (Velvet)	482	60,714	4,470,215	16,033	6	127,247	8.58	4.06	0.00	96.94	1.00	15,439	92
Input 7 (SGA)	2173	29,520	4,188,432	2530	1	4048	5.70	2.47	0.00	90.70	1.00	2280	78
Input 8 (ABySS)	1509	54,734	4,830,769	5562	85	866,218	22.76	5.84	2.32	93.75	1.12	5303	76
CISA (1+2)	178	119,461	4,613,681	48,392	10	299,921	12.50	5.52	2.08	97.69	1.03	47,689	90
CISA ((1+2)+3)	147	119,461	4,677,881	50,889	16	408,918	22.74	5.45	1.73	97.33	1.05	48,389	89
CISA (((1+2)+3)+4)	68	279,125	4,997,864	97,454	16	588,100	29.68	7.58	0.56	92.31	1.18	104,509	73
CISA (((1+2)+...)+5)	39	377,514	5,072,379	154,477	18	1,224,227	32.58	10.63	0.30	89.95	1.23	154,477	67
CISA (((1+2)+...)+6)	38	377,514	5,008,389	154,477	18	1,224,227	32.48	10.72	0.30	89.95	1.21	154,477	68
CISA (((1+2)+...)+7)	38	377,567	5,008,474	154,509	18	1,224,227	32.48	10.72	0.30	89.95	1.21	154,509	68
CISA (((1+2)+...)+8)	38	377,567	5,009,192	154,509	18	1,224,227	32.48	10.72	0.30	89.96	1.21	154,509	68
GAA (1+2)	Did not produce an assembly file												
GAM_NGS (1+2)	187	201,120	4,466,857	49,612	20	266,620	23.33	6.54	0.18	96.27	1.01	44,822	91
GAM_NGS ((1+2)+3)	182	201,120	4,466,712	49,612	20	266,620	23.27	6.54	0.18	96.27	1.01	44,822	90
GAM_NGS (((1+2)+3)+4)	177	201,120	4,470,093	49,612	21	315,092	23.31	6.52	0.18	96.36	1.01	47,869	91
GAM_NGS (((1+2)+...)+5)	135	201,120	4,470,366	74,545	21	318,137	23.47	7.06	0.18	96.36	1.01	73,441	91
GAM_NGS (((1+2)+...)+6)	132	201,120	4,471,653	81,570	21	362,068	24.07	7.14	0.18	96.39	1.01	77,229	90
GAM_NGS (((1+2)+...)+7)	132	201,120	4,471,957	81,570	21	362,068	24.07	7.12	0.18	96.40	1.01	77,229	90
GAM_NGS (((1+2)+...)+8)	131	201,120	4,471,960	81,570	21	390,907	24.16	7.12	0.18	96.40	1.01	75,898	90
GARM (1+2)	63	287,164	4,330,697	110,067	12	754,372	10.33	7.41	1.18	90.85	1.02	104,539	86
GARM ((1+2)+3)	6	222,843	682,218	191,001	2	191,001	9.26	6.83	3.52	14.31	1.04	NA	14
Metassembler (1+2)	158	201,129	4,456,882	50,989	14	251,647	24.31	5.60	0.56	96.59	1.00	48,911	91
Metassembler ((1+2)+3)	148	201,129	4,296,041	50,989	10	196,260	23.00	5.65	0.33	93.12	1.00	45,415	88
Metassembler (((1+2)+3)+4)	139	201,129	4,292,624	53,825	10	196,260	22.69	5.70	0.33	93.05	1.00	47,869	88
Metassembler (((1+2)+...)+5)	94	213,406	4,318,507	74,545	11	265,268	22.26	6.29	0.32	93.58	1.00	67,149	89
Metassembler (((1+2)+...)+6)	94	213,406	4,318,507	74,545	11	265,268	22.26	6.29	0.32	93.58	1.00	67,149	89
Metassembler (((1+2)+...)+7)	94	213,406	4,318,507	74,545	11	265,268	22.26	6.29	0.32	93.58	1.00	67,149	89
Metassembler (((1+2)+...)+8)	94	213,307	4,317,739	74,545	11	265,268	22.27	6.29	0.32	93.56	1.00	67,149	89
MIX (1+2)	150	106,467	3,059,487	41,925	7	286,990	7.10	5.04	2.55	66.42	1.00	20,550	64
MIX ((1+2)+3)	133	106,467	2,823,064	41,925	7	286,990	7.37	5.25	2.27	61.29	1.00	18,408	59
MIX (((1+2)+3)+4)	92	167,490	2,083,309	93,198	2	31,469	5.14	5.19	0.00	45.24	1.00	NA	43
MIX (((1+2)+...)+5)	38	376,585	1,727,015	162,015	4	176,145	18.10	7.98	0.00	37.57	1.00	NA	36
MIX (((1+2)+...)+6)	43	376,585	1,612,424	131,681	4	176,145	24.32	8.19	0.00	35.01	1.00	NA	34
MIX (((1+2)+...)+7)	44	376,585	1,813,898	162,015	4	176,145	21.89	7.94	0.00	39.40	1.00	NA	38
MIX (((1+2)+...)+8)	40	376,585	1,870,882	162,015	3	154,488	21.65	8.18	0.00	40.64	1.00	NA	39
ZORRO (1+2)	202	201,129	4,647,602	47,869	15	277,329	21.61	5.81	1.12	99.52	1.01	43,630	93
ZORRO ((1+2)+3)	197	164,959	4,647,457	44,633	15	277,032	22.25	5.87	1.20	99.50	1.01	42,476	93
ZORRO (((1+2)+3)+4)	401	105,289	4,575,721	18,953	13	221,587	20.84	5.38	0.46	97.77	1.02	18,719	90
ZORRO (((1+2)+...)+5)	390	88,474	4,716,386	21,369	15	258,783	22.95	5.77	0.81	99.11	1.03	21,369	89
ZORRO (((1+2)+...)+6)	Produced an empty assembly file												

Table A.8: Experimental results on merging more than two assemblies (contigs) ordered by the FRCurve score (*Hg_chr14*, genome size 107,349,540 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Input 1 (CABOG)	3233	296,904	86,189,919	46,699	108	3,694,326	101.52	23.29	0.00	79.94	1.00	35,539	59
Input 2 (ALLPATHS-LG)	4469	240,773	84,416,102	38,359	109	1,384,277	67.71	21.79	54.60	78.48	1.00	27,772	63
Input 3 (ABySS)	32,050	30,053	67,074,140	3182	24	128,244	84.48	9.20	1.31	61.54	1.01	1319	84
Input 4 (SOAPdenovo)	15,028	147,494	90,398,734	16,179	6329	43,713,769	152.34	24.26	0.02	77.30	1.09	8155	64
Input 5 (MSR-CA)	25,022	53,925	81,485,144	5470	2038	6,589,712	220.87	24.66	0.00	74.21	1.02	3427	83
Input 6 (BAMBUS)	12,396	736,657	67,814,016	8500	2973	12,211,265	104.23	22.18	0.01	62.52	1.01	3218	62
Input 7 (SGA)	33,695	30,350	75,492,807	3317	107	249,973	87.51	12.57	0.00	69.89	1.01	1945	89
Input 8 (Velvet)	32,842	27,872	70,575,215	3087	232	602,910	104.51	21.41	0.00	65.33	1.00	1580	82
GAA (1+2)						Did not produce an assembly file							
GAM_NGS (1+2)	1828	483,622	86,048,162	85,886	114	5,037,243	97.81	24.07	7.99	80.04	1.00	64,276	50
GAM_NGS ((1+2)+3)	1807	483,622	86,137,425	86,940	113	5,369,578	98.02	24.10	8.00	80.04	1.00	66,262	50
GAM_NGS (((1+2)+3)+4)	1762	484,018	87,263,515	89,705	651	20,748,279	100.92	24.45	7.46	79.62	1.02	60,249	47
GAM_NGS (((1+2)+...)+5)	1755	484,018	87,236,255	90,198	620	20,456,345	101.32	24.51	7.46	79.62	1.02	60,573	47
GAM_NGS (((1+2)+...)+6)	1732	484,018	87,152,297	91,223	591	20,589,809	101.31	24.63	7.42	79.57	1.02	61,643	47
GAM_NGS (((1+2)+...)+7)	1731	484,018	87,147,035	91,259	589	20,770,121	101.80	24.68	7.40	79.58	1.02	61,184	47
GAM_NGS (((1+2)+...)+8)	1742	484,018	87,276,638	91,162	577	20,890,649	101.86	25.18	7.41	79.56	1.02	60,968	47
GARM (1+2)	1528	675,898	91,934,931	104,633	267	15,806,190	97.48	27.97	1.37	78.98	1.08	83,375	46
GARM ((1+2)+3)	11	264,708	1,042,179	144,055	2	209,647	78.68	22.18	3.74	0.93	1.05	NA	2
GARM (((1+2)+3)+4)	6	264,725	587,151	145,239	1	72,910	120.13	29.78	5.28	0.55	1.00	NA	1
GARM (((1+2)+...)+5)	5	265,528	506,174	265,528	2	338,439	132.87	30.41	3.56	0.47	1.00	NA	1
GARM (((1+2)+...)+6)	4	61,336	95,535	61,336	1	11,608	145.65	44.01	0.00	0.09	1.00	NA	1
GARM (((1+2)+...)+7)	4	61,340	95,539	61,340	1	11,608	146.70	44.01	0.00	0.09	1.00	NA	1
GARM (((1+2)+...)+8)	10	61,349	358,528	61,349	2	23,218	175.32	59.48	0.00	0.09	3.75	NA	1
Metassembler (1+2)	3148	296,904	86,092,149	46,780	107	3,693,384	101.12	23.24	0.00	79.87	1.00	35,539	59
Metassembler ((1+2)+3)	3107	296,904	86,035,088	46,780	105	3,687,158	101.04	23.22	0.00	79.82	1.00	35,539	59
Metassembler(((1+2)+...)+4)	3107	296,904	86,035,088	46,780	105	3,687,158	101.04	23.22	0.00	79.82	1.00	35,539	59
Metassembler(((1+2)+...)+5)	3106	296,904	86,032,998	46,780	106	3,726,354	101.04	23.22	0.00	79.82	1.00	35,506	59
Metassembler (((1+2)+...)+6)	2849	296,904	83,919,773	48,129	86	3,655,931	101.27	22.81	0.00	77.89	1.00	35,299	57
Metassembler(((1+2)+...)+7)	2849	296,904	83,919,773	48,129	86	3,655,931	101.27	22.81	0.00	77.89	1.00	35,299	57
Metassembler(((1+2)+...)+8)	2833	296,904	83,887,139	48,129	86	3,655,931	101.24	22.81	0.00	77.86	1.00	35,299	57

Table A.9: Experimental results on merging more than two assemblies (scaffolds) ordered by the FRCurve score (*Staphylococcus aureus*, genome size 2,903,081 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: all reported statistics are for scaffolds; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Input 1 (MSR-CA)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96
Input 2 (ALLPATHS-LG)	11	1,435,559	2,879,481	1,091,731	0	0	3.97	2.51	345.31	98.86	1.00	1,082,616	96
Input 3 (BAMBUS2)	16	1,426,293	2,862,465	1,083,792	5	2,682,283	1.48	7.86	1020.27	97.72	1.01	675,931	96
Input 4 (SGA)	299	286,534	3,051,005	149,421	3	113,622	1.09	11.47	9852.72	94.87	1.11	134,849	82
Input 5 (Velvet)	26	989,718	2,860,883	762,333	38	2,518,079	14.78	2.92	618.27	97.90	1.01	142,399	96
Input 6 (SOAPdenovo)	64	518,710	2,902,967	331,598	32	2,348,756	24.23	2.76	167.31	98.55	1.01	172,575	96
Input 7 (ABySS)	206	130,192	3,692,703	27,695	10	178,901	12.22	1.09	1520.97	97.54	1.30	31,703	77
CISA (1+2)	10	2,411,914	2,865,036	2,411,914	47	2,857,360	20.87	3.71	308.93	98.39	1.00	220,020	96
CISA (1+2+3)	14	1,796,233	2,884,041	1,796,233	47	2,856,673	20.72	3.79	283.08	99.07	1.00	220,020	96
CISA (1+2+3+4)	5	2,411,914	3,005,759	2,411,914	33	2,411,914	22.26	3.16	2186.34	82.94	1.25	220,020	64
CISA (1+2+…+5)	5	2,411,914	3,005,759	2,411,914	33	2,411,914	22.26	3.16	2186.34	82.94	1.25	220,020	64
CISA (1+2+…+6)	5	2,411,914	3,005,759	2,411,914	33	2,411,914	22.26	3.16	2186.34	82.94	1.25	220,020	64
CISA (1+2+…+7)	5	2,411,914	3,005,759	2,411,914	33	2,411,914	22.26	3.16	2186.34	82.94	1.25	220,020	64
GAA (1+2)	Did not produce an assembly file												
GAM_NGS(1+2)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96
GAM_NGS((1+2)+3)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96
GAM_NGS(((1+2)+3)+4)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96
GAM_NGS(((1+2)+…)+5)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96
GAM_NGS(((1+2)+…)+6)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96
GAM_NGS(((1+2)+…)+7)	13	2,411,914	2,871,405	2,411,914	49	2,806,230	20.90	3.75	360.56	98.23	1.01	220,020	96
GARM (1+2)	16	833,732	3,469,519	625,562	7	2,559,729	14.53	3.69	0.09	93.38	1.28	325,955	65
GARM ((1+2)+3)	16	833,738	3,554,698	625,563	8	2,524,523	11.80	3.92	0.23	93.11	1.30	330,720	64
GARM (((1+2)+3)+4)	1	634,995	634,995	634,995	4	634,995	14.10	4.59	1.42	21.74	1.01	NA	23
GARM (((1+2)+…)+5)	2	625,576	634,987	625,576	3	634,987	17.90	5.07	0.00	21.74	1.01	NA	22
GARM (((1+2)+…)+6)	11	316,522	1,117,006	152,795	10	794,721	29.53	2.44	0.00	38.15	1.01	NA	39
GARM (((1+2)+…)+7)	79	316,549	2,768,200	239,944	24	1,756,442	31.13	1.72	4.84	43.93	2.17	152,817	16
Metassembler(1+2)	6	2,411,900	2,864,554	2,411,900	41	2,747,562	21.29	4.03	326.75	98.19	1.01	254,304	96
Metassembler((1+2)+3)	5	2,411,900	2,863,848	2,411,900	41	2,747,562	21.12	4.04	326.83	98.17	1.01	254,304	96
Metassembler(((1+2)+3)+4)	5	2,412,269	2,864,113	2,412,269	38	2,747,776	20.93	4.04	428.40	98.08	1.01	254,235	96
Metassembler(((1+2)+…)+5)	5	2,412,191	2,864,011	2,412,191	38	2,747,674	21.07	4.07	413.83	98.10	1.01	254,235	96
Metassembler(((1+2)+…)+6)	5	2,412,371	2,864,191	2,412,371	38	2,747,854	21.34	4.11	385.34	98.13	1.01	254,869	96
Metassembler(((1+2)+…)+7)	5	2,412,557	2,869,873	2,412,557	37	2,748,040	21.34	4.11	383.64	98.14	1.01	257,812	96
MIX (1+2)	8	3,844,359	5,658,879	3,844,359	54	5,596,862	14.81	3.91	338.18	98.60	1.98	1,077,439	59
MIX (1+2+3)	8	3,844,359	6,742,126	3,844,359	55	6,680,654	14.70	4.61	498.33	98.66	2.36	1,077,439	59
MIX (1+2+3+4)	8	3,844,359	6,798,280	3,844,359	56	6,795,960	14.77	4.61	597.20	98.66	2.38	1,077,439	57
MIX (1+2+…+5)	8	3,844,359	6,856,232	3,844,359	57	6,853,912	14.59	4.68	592.54	98.66	2.40	1,077,568	57
MIX (1+2+…+6)	8	3,846,299	6,858,172	3,846,299	59	6,855,852	14.77	4.61	592.34	98.67	2.40	1,077,568	57
MIX (1+2+…+7)	10	2,740,531	4,120,225	2,740,531	50	4,060,521	19.68	5.51	713.87	96.97	1.47	393,556	52
ZORRO (1+2)	124	178,394	2,999,234	57,412	4	35,916	8.77	3.66	490.53	98.93	1.04	60,028	92
ZORRO ((1+2)+3)	180	140,437	2,958,837	37,759	7	69,094	6.15	2.73	1241.13	98.56	1.02	37,759	93
ZORRO (((1+2)+3)+4)	799	48,229	3,196,025	10,028	4	11,211	4.59	1.94	9394.80	97.53	1.02	11,005	86
ZORRO (((1+2)+…)+5)	427	142,720	3,249,732	36,869	6	7464	13.11	2.24	9421.15	98.28	1.03	41,143	93
ZORRO (((1+2)+…)+6)	375	212,768	3,428,465	54,406	21	540,955	26.36	3.76	8971.77	99.14	1.08	62,734	90

Table A.10: Experimental results on merging more than two assemblies (scaffolds) ordered by the FRCurve score (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: all reported statistics are for scaffolds; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation Tool or Input	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Input 1 (MSR-CA)	36	2,975,504	4,495,726	2,975,504	29	3,765,235	20.10	11.36	725.76	96.22	1.01	535,459	92
Input 2 (ALLPATHS-LG)	33	3,192,334	4,608,763	3,192,334	15	4,447,871	5.91	6.79	467.31	99.24	1.01	928,821	95
Input 3 (CABOG)	130	1,352,519	4,259,679	245,073	20	2,427,350	28.19	7.91	505.84	91.95	1.01	55,312	88
Input 4 (BAMBUS2)	92	2,438,508	4,428,612	2,438,508	11	2,971,543	5.95	6.20	1288.01	94.94	1.01	343,187	90
Input 5 (SOAPdenovo)	76	1,154,134	4,579,801	660,164	13	1,927,959	21.30	9.81	228.42	98.73	1.01	539,770	92
Input 6 (Velvet)	115	770,958	4,572,546	353,027	41	1,840,832	8.64	10.36	1898.40	97.43	1.01	223,489	93
Input 7 (SGA)	1208	148,756	5,328,387	44,205	3	31,764	5.86	7.22	21,499.94	90.77	1.26	17,695	77
Input 8 (ABySS)	1352	87,855	4,968,921	8036	85	1,012,703	23.38	8.92	2302.47	94.21	1.14	7136	76
CISA (1+2)	8	3,192,334	4,962,823	3,192,334	20	4,818,520	10.09	7.26	559.52	95.79	1.13	928,821	82
CISA (1+2+3)	5	2,952,884	5,053,724	2,952,884	12	3,946,023	8.23	7.40	397.67	88.68	1.24	865,102	65
CISA (1+2+3+4)	5	2,952,884	5,053,724	2,952,884	12	3,946,023	8.23	7.40	397.67	88.68	1.24	865,102	65
CISA (1+2+...+5)	5	2,952,884	5,053,724	2,952,884	12	3,946,023	8.23	7.40	397.67	88.68	1.24	865,102	65
CISA (1+2+...+6)	5	2,952,991	5,054,054	2,952,991	13	4,184,063	8.23	7.37	397.64	88.68	1.24	865,102	65
CISA (1+2+...+7)	5	2,952,991	5,054,054	2,952,991	13	4,184,063	8.23	7.37	397.64	88.68	1.24	865,102	65
CISA (1+2+...+8)	5	2,953,019	5,054,305	2,953,019	13	4,184,091	8.23	7.40	397.62	88.68	1.24	865,325	65
GAA (1+2)						Did not produce an assembly file							
GAM_NGS(1+2)	36	2,975,504	4,495,726	2,975,504	29	3,765,235	20.10	11.36	725.76	96.22	1.01	535,459	92
GAM_NGS(1+2+3)	36	2,975,504	4,495,726	2,975,504	29	3,765,235	20.10	11.36	725.76	96.22	1.01	535,459	92
GAM_NGS(((1+2)+3)+4)	36	2,975,504	4,495,885	2,975,504	29	3,765,235	19.46	11.24	725.73	96.22	1.01	535,459	92
GAM_NGS(((1+2)+...)+5)	35	2,975,504	4,495,773	2,975,504	29	3,765,235	19.53	11.24	725.75	96.22	1.01	535,459	92
GAM_NGS(((1+2)+...)+6)	35	2,975,504	4,495,773	2,975,504	29	3,765,235	19.53	11.24	725.75	96.22	1.01	535,459	92
GAM_NGS(((1+2)+...)+7)	35	2,975,504	4,495,773	2,975,504	29	3,765,235	19.53	11.24	725.75	96.22	1.01	535,459	92
GAM_NGS(((1+2)+...)+8)	35	2,975,504	4,495,773	2,975,504	29	3,765,235	19.53	11.24	725.75	96.22	1.01	535,459	92
GARM (1+2)	63	287,164	4,330,697	110,067	12	754,372	10.33	7.41	1.18	90.85	1.02	104,539	86
GARM ((1+2)+3)	6	222,843	682,218	191,001	2	191,001	9.26	6.83	3.52	14.31	1.04	NA	14
Metassembler(1+2)	18	2,975,287	4,485,906	2,975,287	23	3,767,604	19.45	10.94	623.84	96.53	1.01	542,318	92
Metassembler(1+2+3)	12	2,975,287	4,330,089	2,975,287	18	3,610,551	18.22	11.16	598.37	93.23	1.01	542,318	89
Metassembler(((1+2)+3)+4)	11	2,975,287	4,329,474	2,975,287	18	3,610,551	18.22	11.16	598.46	93.22	1.01	542,318	89
Metassembler(((1+2)+...)+5)	11	2,974,171	4,345,683	2,974,171	19	3,627,111	17.67	11.17	589.73	93.56	1.01	542,141	90
Metassembler(((1+2)+...)+6)	11	2,974,143	4,345,655	2,974,143	19	3,627,083	17.67	11.10	588.10	93.56	1.01	542,141	90
Metassembler(((1+2)+...)+7)	11	2,974,043	4,345,553	2,974,043	19	3,627,064	18.32	11.57	849.21	93.32	1.01	541,086	90
Metassembler(((1+2)+...)+8)	11	2,973,698	4,344,414	2,973,698	17	3,626,594	18.28	11.55	848.51	93.29	1.01	541,086	90
MIX (1+2)	17	10,738,254	11,354,250	10,738,254	46	11,324,751	11.33	5.84	550.11	96.65	2.55	1,443,594	91
MIX (1+2+3)	13	10,738,254	11,886,482	10,738,254	47	11,832,525	19.84	6.14	568.05	96.92	2.67	1,443,594	78
MIX (1+2+3+4)	16	10,738,254	12,576,249	10,738,254	55	12,334,005	20.32	6.35	566.99	97.50	2.80	1,443,594	72
MIX (1+2+...+5)	20	6,575,227	9,499,456	6,575,227	46	9,190,236	19.93	6.41	621.73	97.21	2.12	1,208,391	70
MIX (1+2+...+6)	24	6,575,227	10,129,718	6,575,227	56	9,100,171	19.91	6.62	825.45	97.42	2.26	1,208,391	59
MIX (1+2+...+7)	27	6,575,227	10,240,091	6,575,227	60	9,181,273	20.12	6.68	1035.00	97.60	2.28	1,208,391	58
MIX (1+2+...+8)	31	6,575,227	10,345,176	6,575,227	76	9,214,371	20.75	6.81	1891.69	98.84	2.28	1,208,391	56
ZORRO (1+2)	213	223,172	4,709,234	60,544	12	281,886	19.37	11.67	868.36	99.27	1.02	60,489	92
ZORRO (1+2+3)	209	159,412	4,760,417	62,222	15	425,102	22.94	11.10	1204.83	99.26	1.03	60,687	92
ZORRO (((1+2)+3)+4)	543	90,880	4,735,152	17,051	15	256,065	18.12	7.99	2213.93	97.38	1.04	17,579	88
ZORRO (((1+2)+...)+5)	529	88,610	4,844,535	18,041	21	252,097	25.08	8.57	2262.12	98.90	1.04	18,736	88
ZORRO (((1+2)+...)+6)	493	158,620	4,904,571	26,231	19	237,325	21.13	10.03	3836.77	98.80	1.04	26,811	90
ZORRO (((1+2)+...)+7)	2207	87,934	5,979,889	3125	14	141,079	20.06	6.03	21,777.66	97.91	1.04	4501	80

Table A.11: Experimental results on merging more than two assemblies (scaffolds) ordered by the FRCurve score (*Hg_chr14*, genome size 107,349,540 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: all reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation	Contigs	Largest	Size	N50	Misassembly	Misassembly	Mismatches	Indels	N's	Genome	Duplication	NGA50	Genes
Tool or Input	(#)	(bp)	(bp)	(bp)	(#)	Length (bp)	(#)	(#)	(#)	covered (%)	ratio	(bp)	(%)
Input 1 (CABOG)	474	2,260,562	86,481,568	401,279	269	43,205,905	101.10	24.68	267.20	80.01	1.01	215,699	29
Input 2 (ALLPATHS-LG)	174	81,646,936	87,646,728	81,646,936	455	87,219,645	66.85	22.87	3734.63	78.51	1.04	397,351	14
Input 3 (ABySS)	31,582	30,053	67,724,594	3355	37	237,738	84.67	9.37	859.44	61.61	1.02	1339	83
Input 4 (SOAPdenovo)	7264	1,849,511	100,880,746	381,286	8171	89,396,625	152.68	24.52	10,166.39	77.44	1.20	17,851	31
Input 5 (MSR-CA)	1056	4,208,965	89,531,681	893,428	6938	86,537,566	226.67	41.77	6810.92	75.68	1.10	31,378	23
Input 6 (BAMBUS)	847	2,671,981	78,283,056	372,757	5010	74,197,764	102.69	21.95	13,247.26	62.59	1.16	13,131	19
Input 7 (SGA)	9586	551,622	88,557,645	82,616	170	8,811,457	87.77	18.34	14,499.38	70.47	1.16	35,317	34
Input 8 (Velvet)	1463	4,628,722	138,771,192	854,836	12,836	99,579,307	107.32	28.47	45,797.46	68.55	1.57	3334	28
GAA (1+2)	Did not produce an assembly file												
GAM_NGS (1+2)	473	2,260,562	86,479,699	401,279	269	43,205,905	101.10	24.68	267.22	80.01	1.01	215,699	29
GAM_NGS ((1+2)+3)	466	2,260,562	87,363,451	409,989	271	43,745,348	101.36	24.71	272.57	80.01	1.02	226,183	NA
GAM_NGS (((1+2)+3)+4)	462	2,260,562	87,370,724	409,989	276	43,994,460	101.39	24.67	284.02	80.00	1.02	226,183	29
GAM_NGS (((1+2)+...)+5)	473	2,260,562	86,479,699	401,279	269	43,205,905	101.10	24.68	267.22	80.01	1.01	215,699	29
GAM_NGS (((1+2)+...)+6)	455	2,260,562	87,576,196	409,989	331	47,081,846	101.52	24.68	734.37	79.81	1.02	223,685	28
GAM_NGS (((1+2)+...)+7)	455	2,260,562	87,577,295	409,989	331	47,081,846	101.51	24.68	737.15	79.81	1.02	223,685	28
GARM (1+2)	1528	675,898	91,934,931	104,633	267	15,806,190	97.48	27.97	1.37	78.98	1.08	83,375	0
GARM ((1+2)+3)	11	264,708	1,042,179	144,055	2	209,647	78.68	22.18	3.74	0.93	1.05	NA	0
GARM (((1+2)+3)+4)	6	264,725	587,151	145,239	1	72,910	120.13	29.78	5.28	0.55	1.00	NA	0
GARM (((1+2)+...)+5)	5	265,528	506,174	265,528	2	338,439	132.87	30.41	3.56	0.47	1.00	NA	0
GARM (((1+2)+...)+6)	4	61,336	95,535	61,336	1	11,608	145.65	44.01	0.00	0.09	1.00	NA	0
GARM (((1+2)+...)+7)	4	61,340	95,539	61,340	1	11,608	146.70	44.01	0.00	0.09	1.00	NA	0
GARM (((1+2)+...)+8)	10	61,349	358,528	61,349	2	23,218	175.32	59.48	0.00	0.09	3.75	NA	0
Metassembler (1+2)	465	2,260,562	86,454,843	401,279	269	43,205,905	100.93	24.64	267.28	79.99	1.01	215,699	29
Metassembler ((1+2)+3)	461	2,260,562	86,446,782	401,279	269	43,205,905	100.92	24.64	267.31	79.99	1.01	215,699	29
Metassembler(((1+2)+...)+4)	461	2,260,562	86,446,782	401,279	269	43,205,905	100.92	24.64	267.31	79.99	1.01	215,699	29
Metassembler(((1+2)+...)+5)	461	2,260,562	86,446,782	401,279	269	43,205,905	100.92	24.64	267.31	79.99	1.01	215,699	29
Metassembler (((1+2)+...)+6)	457	2,260,562	86,427,089	401,279	269	43,205,905	100.93	24.64	266.36	79.97	1.01	215,699	29
Metassembler(((1+2)+...)+7)	457	2,260,562	86,427,089	401,279	269	43,205,905	100.93	24.64	266.36	79.97	1.01	215,699	29
Metassembler(((1+2)+...)+8)	456	2,260,562	86,419,639	401,279	269	43,205,905	100.93	24.64	266.38	79.97	1.01	215,699	29

Table A.12: Experimental results on merging more than two assemblies (as contigs) with an alternative ordering (*Staphylococcus aureus*, genome size 2,903,081 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: Statistics reported are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted

Reconciliation Tool	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Input 1 (SGA)	985	16,870	2,748,664	4178	1	2431	1.02	0.11	0.00	94.44	1.00	4005	82
Input 2 (ABySS)	247	125,049	3,631,245	25,084	5	22,399	12.39	0.85	7.79	97.27	1.29	29,198	78
Input 3 (ALLPATHS-LG)	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
Input 4 (MSR-CA)	89	139,438	2,860,132	59,152	20	641,173	20.67	1.82	0.00	98.17	1.00	55,068	95
Input 5 (Velvet)	128	169,214	2,837,036	52,792	9	284,379	13.51	1.69	0.00	97.73	1.00	48,149	97
Input 6 (SOAPdenovo)	70	518,710	2,897,432	288,184	31	2,027,905	23.00	2.45	0.07	98.55	1.01	150,794	96
Input 7 (BAMBUS2)	106	158,330	2,832,623	50,192	1	9411	1.41	6.95	0.35	97.66	1.00	50,192	95
CISA (1+2)	183	125,049	2,775,035	25,585	6	44,257	14.86	1.23	4.58	95.07	1.00	23,610	93
CISA (1+2+3)	52	245,875	2,877,782	131,781	2	251,080	5.15	1.08	4.66	98.97	1.00	131,781	96
CISA (1+2+3+4)	27	481,008	2,867,994	114,938	10	1,204,080	8.73	1.09	1.26	95.04	1.04	114,873	90
CISA (1+2+. . . +5)	22	390,086	2,730,406	201,618	8	1,101,155	6.87	1.02	1.28	90.78	1.04	114,873	86
CISA (1+2+. . . +6)	18	521,399	3,033,607	288,204	28	2,329,572	16.70	2.18	0.86	94.85	1.10	179,344	84
CISA (1+2+. . . +7)	18	521,399	3,033,607	288,204	28	2,329,572	16.70	2.18	0.86	94.85	1.10	179,344	84
GAA (1+2)	1232	125,049	6,379,909	10,535	6	24,830	11.04	0.70	4.44	97.93	2.24	29,198	27
GAA ((1+2)+3)	1290	234,488	9,246,594	24,526	7	114,464	5.06	0.87	3.53	99.30	3.21	107,125	27
GAA (((1+2)+3)+4)	1378	234,488	12,105,447	32,180	26	754,358	11.24	1.66	2.69	99.63	4.19	123,414	27
GAA ((((1+2)+3)+4)+5)	1502	234,488	14,931,998	35,275	35	1,038,737	14.23	2.63	2.18	99.73	5.16	132,320	27
GAA ((((1+2)+. . .)+5)+6)	1570	518,710	17,823,454	46,565	66	3,066,642	18.36	2.59	1.84	99.80	6.16	200,247	27
GAA ((((1+2)+. . .)+6)+7)	1665	518,710	20,622,604	48,304	67	3,076,053	18.02	2.80	1.64	99.80	7.12	200,247	27
GAM_NGS (1+2)	545	77,430	2,784,898	10,328	1	2431	2.19	0.36	0.36	95.78	1.00	9371	89
GAM_NGS ((1+2)+3)	193	234,571	2,835,523	69,769	1	2431	2.51	0.60	1.23	97.61	1.00	68,648	92
GAM_NGS (((1+2)+3)+4)	120	386,769	2,846,597	97,655	2	386,651	4.43	0.60	1.23	98.02	1.00	97,655	93
GAM_NGS ((((1+2)+3)+4)+5)	89	386,769	2,852,151	98,193	3	484,844	6.14	0.70	1.23	98.22	1.00	97,655	94
GAM_NGS ((((1+2)+. . .)+5)+6)	65	475,528	2,857,733	172,603	3	576,152	6.97	0.74	1.22	98.37	1.00	149,870	94
GAM_NGS ((((1+2)+. . .)+6)+7)	50	493,282	2,858,546	216,471	3	576,152	7.04	0.88	1.29	98.41	1.00	172,596	94
GARM (1+2)	96	125,477	1,486,962	28,882	6	32,684	24.19	4.73	8.14	30.61	1.67	3115	20
GARM ((1+2)+3)	26	751,197	2,697,219	234,533	0	0	8.08	3.30	0.15	92.91	1.00	200,248	91
GARM (((1+2)+3)+4)	27	833,735	3,132,585	430,275	14	1,960,878	19.30	5.00	0.00	98.54	1.10	331,026	88
GARM ((((1+2)+3)+4)+5)	28	885,261	5,439,174	480,862	21	4,289,841	19.85	5.19	0.00	98.03	1.91	833,528	19
GARM ((((1+2)+. . .)+5)+6)	12	885,261	2,983,960	885,261	18	2,810,529	24.05	3.26	0.00	56.00	1.84	833,529	22
GARM ((((1+2)+. . .)+6)+7)	53	885,273	3,346,358	158,330	14	1,715,106	14.38	5.51	0.09	95.59	1.21	195,100	75
Metassembler (1+2)	401	77,430	2,825,605	19,008	3	11,572	5.42	0.54	1.91	96.54	1.01	18,208	91
Metassembler ((1+2)+3)	139	172,102	2,855,739	48,865	3	36,237	4.19	0.92	2.77	97.86	1.01	48,862	94
Metassembler (((1+2)+3)+4)	115	218,147	2,847,058	56,518	4	61,375	7.20	1.20	2.70	97.58	1.01	53,149	94
Metassembler ((((1+2)+3)+4)+5)	114	218,147	2,847,492	56,518	5	167,844	8.12	1.24	2.70	97.59	1.01	53,149	94
Metassembler ((((1+2)+. . .)+5)+6)	102	222,905	2,857,903	64,023	12	641,767	13.13	1.44	2.76	97.86	1.01	60,945	94
Metassembler ((((1+2)+. . .)+6)+7)	101	222,905	2,857,921	64,023	12	641,767	13.13	1.48	2.76	97.86	1.01	60,945	94
MIX (1+2)	156	125,049	2,400,202	27,695	3	12,322	15.91	1.11	6.83	80.34	1.03	23,082	79
MIX (1+2+3)	44	234,488	1,946,016	97,697	1	89,634	2.00	0.82	1.90	67.03	1.00	48,304	64
MIX (1+2+3+4)	19	200,247	658,945	69,391	0	0	3.34	0.91	1.97	22.69	1.00	NA	23
MIX (1+2+. . . +5)	18	200,247	685,973	66,230	0	0	3.06	0.73	1.75	23.62	1.00	NA	24
MIX (1+2+. . . +6)	22	518,710	1,614,188	331,598	24	1,345,664	25.45	2.92	0.00	55.35	1.01	46,390	55
MIX (1+2+. . . +7)	22	518,710	1,283,250	316,522	16	1,014,066	26.86	3.13	0.00	43.99	1.01	NA	44
ZORRO (1+2)	Produced an empty assembly file												

Table A.13: Experimental results on merging more than two assemblies (as contigs) with an alternative ordering (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: Statistics reported are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation	Contigs	Largest	Size	N50	Misassembly	Misassembly	Mismatches	Indels	N's	Genome	Duplication	NGA50	Genes
Tool	(#)	(bp)	(bp)	(bp)	(#)	Length (bp)	(#)	(#)	(#)	covered (%)	ratio	(bp)	(%)
Input 1 (SGA)	2173	29,520	4,188,432	2530	1	4048	5.70	2.47	0.00	90.70	1.00	2280	78
Input 2 (CABOG)	318	88,519	4,236,663	22,044	11	276,929	29.07	5.48	0.00	91.91	1.00	19,076	87
Input 3 (Velvet)	482	60,714	4,470,215	16,033	6	127,247	8.58	4.06	0.00	96.94	1.00	15,439	92
Input 4 (MSR-CA)	377	83,726	4,458,952	23,575	18	151,790	23.43	4.90	0.00	96.14	1.01	21,236	89
Input 5 (ALLPATHS-LG)	203	106,467	4,587,354	42,455	10	404,185	6.33	4.77	2.79	99.20	1.00	41,487	93
Input 6 (ABySS)	1509	54,734	4,830,769	5562	85	866,218	22.76	5.84	2.32	93.75	1.12	5303	76
Input 7 (BAMBUS2)	170	279,125	4,369,357	97,331	4	123,417	5.82	5.84	0.00	94.89	1.00	93,198	90
Input 8 (SOAPdenovo)	114	376,585	4,569,340	131,681	11	633,163	21.28	9.51	0.00	98.72	1.01	129,613	92
CISA (1+2)	361	88,509	4,227,470	20,303	11	286,366	29.00	5.27	0.00	91.84	1.00	17,590	86
CISA (1+2+3)	351	88,516	4,550,294	22,683	12	311,963	29.42	5.28	0.00	98.28	1.01	21,435	92
CISA (1+2+3+4)	207	88,519	4,632,650	32,548	13	277,288	35.31	6.57	0.00	97.21	1.04	31,768	88
CISA (1+2+...+5)	145	119,461	4,716,607	50,937	20	563,586	23.50	5.56	1.78	98.08	1.05	47,859	89
CISA (1+2+...+6)	120	119,461	4,623,276	51,710	24	714,575	23.88	6.39	1.93	95.61	1.05	50,889	87
CISA (1+2+...+7)	66	279,125	4,983,880	101,270	20	780,427	29.25	8.45	0.72	92.03	1.18	105,293	73
CISA (1+2+...+8)	38	377,567	5,009,192	154,509	18	1,224,227	32.48	10.72	0.30	89.96	1.21	154,509	68
GAA (1+2)	2489	88,519	8,422,953	6871	12	280,977	28.51	5.16	0.00	98.44	1.86	19,278	33
GAA ((1+2)+3)	2971	88,519	12,893,168	11,295	18	408,224	29.41	5.26	0.00	98.69	2.84	26,927	29
GAA (((1+2)+3)+4)	3348	88,519	17,352,120	13,897	36	560,014	35.68	5.66	0.00	99.48	3.79	36,899	30
GAA ((((1+2)+3)+4)+5)	3542	106,467	21,929,269	18,074	46	964,199	23.32	5.54	0.58	99.69	4.78	58,687	30
GAA ((((1+2)+...)+5)+6)	5032	106,467	26,727,756	14,534	131	1,830,417	24.71	6.14	0.90	99.79	5.82	58,687	33
GAA ((((1+2)+...)+6)+7)	5126	279,125	30,814,257	18,186	134	1,948,517	24.73	6.31	0.78	99.79	6.71	97,360	34
GAA (((((1+2)+...)+6)+7)+8)	5239	376,585	35,379,799	22,511	145	2,581,680	27.66	7.92	0.68	99.84	7.70	162,986	34
GAM_NGS (1+2)	772	88,314	4,386,840	15,268	8	177,696	32.62	5.36	0.00	95.30	1.00	13,419	85
GAM_NGS ((1+2)+3)	538	88,314	4,425,712	20,210	8	177,803	32.65	5.81	0.00	96.16	1.00	18,619	86
GAM_NGS (((1+2)+3)+4)	412	103,668	4,468,894	26,294	8	152,011	35.55	6.22	0.00	97.04	1.00	24,420	86
GAM_NGS ((((1+2)+3)+4)+5)	314	129,231	4,496,017	39,565	12	311,939	36.81	6.57	0.24	97.56	1.00	35,767	87
GAM_NGS ((((1+2)+...)+5)+6)	304	136,773	4,498,185	40,397	12	410,768	37.48	6.57	0.24	97.60	1.00	37,591	87
GAM_NGS ((((1+2)+...)+6)+7)	281	136,773	4,502,388	40,839	12	410,811	37.54	6.65	0.24	97.69	1.00	40,397	87
GAM_NGS ((((1+2)+...)+7)+8)	220	227,331	4,517,030	51,976	13	493,518	42.32	7.40	0.24	98.00	1.00	47,344	87
GARM (1+2)	226	88,521	2,972,274	22,754	7	218,125	39.83	7.72	0.13	64.47	1.00	11,605	62
GARM ((1+2)+3)	317	88,521	4,439,643	23,404	15	449,942	29.70	6.36	0.59	88.15	1.10	21,768	78
GARM (((1+2)+3)+4)	196	102,179	4,703,725	41,550	21	468,001	35.75	7.08	0.40	93.95	1.09	40,660	84
GARM ((((1+2)+3)+4)+5)						Did not produce assembly files							

Table A.14: Experimental results on merging more than two assemblies (as contigs) with an alternative ordering (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: Statistics reported are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation Tool	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Input 1 (SGA)	2173	29,520	4,188,432	2530	1	4048	5.70	2.47	0.00	90.70	1.00	2280	78
Input 2 (CABOG)	318	88,519	4,236,663	22,044	11	276,929	29.07	5.48	0.00	91.91	1.00	19,076	87
Input 3 (Velvet)	482	60,714	4,470,215	16,033	6	127,247	8.58	4.06	0.00	96.94	1.00	15,439	92
Input 4 (MSR-CA)	377	83,726	4,458,952	23,575	18	151,790	23.43	4.90	0.00	96.14	1.01	21,236	89
Input 5 (ALLPATHS-LG)	203	106,467	4,587,354	42,455	10	404,185	6.33	4.77	2.79	99.20	1.00	41,487	93
Input 6 (ABySS)	1509	54,734	4,830,769	5562	85	866,218	22.76	5.84	2.32	93.75	1.12	5303	76
Input 7 (BAMBUS2)	170	279,125	4,369,357	97,331	4	123,417	5.82	5.84	0.00	94.89	1.00	93,198	90
Input 8 (SOAPdenovo)	114	376,585	4,569,340	131,681	11	633,163	21.28	9.51	0.00	98.72	1.01	129,613	92
Metassembler (1+2)	654	85,765	4,104,684	14,248	3	98,343	17.15	4.51	0.00	89.07	1.00	13,072	84
Metassembler ((1+2)+3)	611	85,765	4,110,564	15,622	2	46,073	17.10	4.72	0.00	89.21	1.00	13,960	84
Metassembler (((1+2)+3)+4)	447	101,356	4,127,410	20,614	3	69,841	21.91	5.65	0.00	89.62	1.00	18,750	85
Metassembler ((((1+2)+3)+4)+5)	375	125,375	4,161,595	25,381	3	69,841	22.74	6.30	0.17	90.36	1.00	22,037	86
Metassembler ((((1+2)+...)+5)+6)	373	125,375	4,170,384	25,536	3	69,841	22.70	6.32	0.17	90.44	1.00	22,046	86
Metassembler ((((1+2)+...)+6)+7)	244	125,375	4,218,214	37,701	5	102,128	22.87	7.50	0.17	91.50	1.00	33,004	86
Metassembler ((((1+2)+...)+7)+8)	375	125,375	4,161,595	25,381	3	69,841	22.74	6.30	0.17	90.36	1.00	22,037	86
MIX (1+2)	311	88,519	4,130,025	21,782	9	220,445	12.80	4.68	0.00	89.61	1.00	18,331	85
MIX (1+2+3)	299	88,519	3,942,979	22,024	9	183,272	30.24	5.60	0.00	85.71	1.00	17,868	81
MIX (1+2+3+4)	190	88,519	2,425,032	22,024	7	126,788	16.66	4.41	0.00	52.67	1.00	3780	52
MIX (1+2+...+5)	146	106,467	3,028,711	40,555	4	235,868	5.28	4.79	2.05	65.78	1.00	20,380	63
MIX (1+2+...+6)	142	106,467	2,873,498	38,162	4	235,868	5.08	4.32	1.95	62.41	1.00	18,081	60
MIX (1+2+...+7)	78	279,125	2,668,747	107,963	2	31,469	6.74	7.01	0.00	57.98	1.00	28,260	56
MIX (1+2+...+8)	42	376,585	1,905,119	162,015	4	176,145	20.32	7.88	0.00	41.38	1.00	NA	40
ZORRO (1+2)	419	70,517	4,534,309	19,161	10	199,853	37.47	5.67	0.04	98.39	1.00	18,028	92
ZORRO ((1+2)+3)						Produced an empty assembly file							

Table A.15: Experimental results on merging more than two assemblies (as contigs) with an alternative ordering (*Hg_chr14*, genome size 107,349,540 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: Reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps; tools were ran using default parameters, unless otherwise noted; (1+2)+3 means that assembly 1 and 2 were merged first, the result of which was then merged to assembly 3

Reconciliation Tool	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Input 1 (ABySS)	32,050	30,053	67,074,140	3182	24	128,244	84.48	9.20	1.31	61.54	1.01	1319	84
Input 2 (SGA)	33,695	30,350	75,492,807	3317	107	249,973	87.51	12.57	0.00	69.89	1.01	1945	89
Input 3 (ALLPATHS-LG)	4469	240,773	84,416,102	38,359	109	1,384,277	67.71	21.79	54.60	78.48	1.00	27,772	63
Input 4 (CABOG)	3233	296,904	86,189,919	46,699	108	3,694,326	101.52	23.29	0.00	79.94	1.00	35,539	59
Input 5 (MSR-CA)	25,022	53,925	81,485,144	5470	2038	6,589,712	220.87	24.66	0.00	74.21	1.02	3427	83
Input 6 (Velvet)	32,842	27,872	70,575,215	3087	232	602,910	104.51	21.41	0.00	65.33	1.00	1580	82
Input 7 (SOAPdenovo)	15,028	147,494	90,398,734	16,179	6329	43,713,769	152.34	24.26	0.02	77.30	1.09	8155	64
Input 8 (BAMBUS)	12,396	736,657	67,814,016	8500	2973	12,211,265	104.23	22.18	0.01	62.52	1.01	3218	62
GAA (1+2)	65,737	30,350	142,552,348	3255	131	378,217	90.38	13.70	0.62	70.52	1.88	4336	84
GAA ((1+2)+3)	70,204	240,773	226,966,664	6119	240	1,762,494	72.47	22.05	20.70	79.33	2.67	27,960	79
GAA (((1+2)+3)+4)	73,416	296,904	313,139,760	12,780	348	5,456,915	95.13	23.86	15.00	80.66	3.62	62,647	78
GAA ((((1+2)+3)+4)+5)	98,421	296,904	394,599,185	9054	2383	12,043,368	99.30	24.45	11.90	80.80	4.55	62,647	81
GAA ((((1+2)+...)+5)+6)	131,262	296,904	465,173,620	6993	2615	12,646,278	99.66	24.47	10.10	80.87	5.35	62,647	81
GAA ((((1+2)+...)+6)+7)	140,219	296,904	548,769,603	8483	8920	56,321,449	106.13	25.07	8.56	80.98	6.31	63,107	83
GAA ((((1+2)+...)+7)+8)	152,492	736,657	616,390,960	8492	11,893	68,532,714	105.99	25.07	7.63	81.00	7.08	63,107	84
GAM_NGS (1+2)	28,297	52,608	69,334,740	3755	30	163,762	85.20	9.95	1.18	63.74	1.01	1755	83
GAM_NGS ((1+2)+3)	9736	215,503	79,581,806	34,673	45	744,030	85.23	18.38	10.52	73.52	1.01	15,730	60
GAM_NGS (((1+2)+3)+4)	4481	397,751	83,219,962	69,951	74	3,263,338	92.18	21.68	9.94	77.00	1.01	46,651	52
GAM_NGS ((((1+2)+3)+4)+5)	4192	397,751	83,368,507	70,604	82	3,549,998	94.03	22.02	9.90	77.17	1.01	46,964	51
GAM_NGS ((((1+2)+...)+5)+6)	4138	397,751	83,403,680	70,931	86	3,889,938	94.63	22.48	9.86	77.19	1.01	47,047	51
GAM_NGS ((((1+2)+...)+6)+7)	3697	397,751	84,176,751	75,581	866	19,610,702	99.41	23.13	9.31	77.42	1.01	43,297	47
GAM_NGS ((((1+2)+...)+7)+8)	3554	397,751	84,227,364	77,356	807	19,477,192	99.48	23.26	9.26	77.45	1.01	44,168	47
GARM (1+2)	20,866	55,072	70,026,320	5197	81	340,195	88.71	13.87	0.19	64.59	1.00	2420	77
GARM ((1+2)+3)	3352	430,290	84,628,192	51,097	258	6,861,404	90.56	34.38	0.62	78.45	1.00	35,794	57
GARM (((1+2)+3)+4)	1626	621,599	98,329,510	109,228	261	16,196,216	100.24	25.81	0.04	75.80	1.20	93,129	46
GARM ((((1+2)+3)+4)+5)	3519	700,715	152,827,973	125,791	799	38,358,283	143.98	29.84	0.02	77.02	1.84	145,176	46
GARM ((((1+2)+...)+5)+6)	5091	814,188	404,136,067	169,384	1250	108,723,014	143.35	33.86	0.00	74.66	5.04	314,774	43
GARM ((((1+2)+...)+6)+7)	1	950	950	950	0	0	105.26	0.00	0.00	0.00	1.00	NA	0
GARM ((((1+2)+...)+7)+8)						Did not produce assembly files							
Metassembler (1+2)	31,853	30,053	66,646,160	3174	22	124,338	84.11	9.18	1.28	61.48	1.01	1293	77
Metassembler ((1+2)+3)	31,474	30,053	66,230,547	3193	20	117,582	83.51	9.13	1.26	61.18	1.01	1280	76
Metassembler (((1+2)+3)+4)	31,392	30,053	66,140,593	3198	20	117,582	83.46	9.14	1.16	61.10	1.01	1276	76
Metassembler ((((1+2)+3)+4)+5)	31,317	30,053	66,080,716	3202	20	117,582	83.42	9.13	1.16	61.05	1.01	1275	76
Metassembler ((((1+2)+...)+5)+6)	31,189	30,053	65,936,787	3206	20	117,582	83.15	9.08	1.16	60.94	1.01	1271	75
Metassembler ((((1+2)+...)+6)+7)	31,187	30,053	65,935,548	3208	20	117,582	83.14	9.08	1.16	60.94	1.01	1271	75
Metassembler ((((1+2)+...)+7)+8)	24,682	30,053	56,850,690	3565	19	112,449	84.52	9.19	1.26	52.57	1.01	777	64

Table A.16: Experimental results on parameter tuning (*Staphylococcus aureus*, genome size 2,872,915 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: Reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps

Reconciliation Tool	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
Staphylococcus aureus (genome size 2,872,915 bp)													
ALLPATHS-LG	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
SGA	985	16,870	2,748,664	4178	1	2431	1.02	0.11	0.00	94.44	1.00	4005	82
CISA	41	234,488	1,868,640	86,588	2	170,232	2.30	0.96	1.34	64.53	1.00	46,021	63
CISA	41	234,488	1,868,640	86,588	2	170,232	2.30	0.96	1.34	64.53	1.00	46,021	63
CISA	66	127,888	2,199,244	62,398	1	80,598	1.45	0.68	0.77	75.91	1.00	42,963	74
CISA	4	461,617	1,286,214	286,534	0	0	0.98	8.46	7240.24	38.69	1.15	NA	35
CISA	10	924,846	1,860,009	614,593	1	924,846	3.93	2.75	520.70	63.91	1.01	190,401	64
GAM_NGS	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
GAM_NGS	59	234,488	2,869,725	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
GAM_NGS	59	234,488	2,869,696	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
GAM_NGS	59	234,488	2,869,725	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
GAM_NGS	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
GARM	54	234,577	3,716,040	101,091	1	89,633	1.70	1.49	0.13	97.25	1.32	137,450	64
GARM	54	234,577	3,716,505	101,091	1	89,633	1.88	1.45	0.13	97.25	1.32	137,450	64
GARM	54	234,581	3,715,089	101,091	1	89,633	1.95	1.35	0.13	97.21	1.32	137,450	64
GARM	54	234,581	3,715,092	101,091	1	89,633	1.84	1.35	0.13	97.21	1.32	137,450	64
GARM	54	234,581	3,715,110	101,091	1	89,633	2.27	1.35	0.13	97.21	1.32	137,450	64
MIX	53	234,488	2,739,016	97,697	1	89,634	1.46	0.69	1.50	94.34	1.00	96,740	92
MIX	56	234,488	2,760,919	97,697	1	89,634	1.45	0.69	1.49	95.09	1.00	96,740	92
MIX	58	234,488	2,821,277	97,697	1	89,634	1.60	0.74	1.52	97.16	1.00	96,740	94
MIX	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
MIX	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
Metassembler	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
Metassembler	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
Metassembler	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
Metassembler	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
Metassembler	59	234,488	2,869,581	96,740	1	89,634	1.57	0.73	1.50	98.83	1.00	96,740	96
ZORRO	66	234,548	2,873,037	96,674	0	0	1.88	0.66	0.80	98.91	1.00	96,674	96
ZORRO	66	234,548	2,873,101	96,674	0	0	1.85	0.66	0.77	98.91	1.00	96,674	96
ZORRO	67	234,548	2,875,958	96,674	0	0	1.81	0.66	0.80	98.91	1.00	96,674	96
ZORRO	69	234,548	2,873,546	96,657	0	0	1.99	0.59	0.90	98.90	1.00	96,657	97
ZORRO	69	234,548	2,877,597	96,674	0	0	1.92	0.63	1.22	98.91	1.00	96,674	96

Table A.17: Experimental results on parameter tuning (*Rhodobacter sphaeroides*, genome size 4,603,060 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: Statistics reported are for contigs; the number of mismatches/indels/Ns are per 100 Kbps;

Reconciliation Tool	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
ALLPATHS-LG	203	106,467	4,587,354	42,455	10	404,185	6.33	4.77	2.79	99.20	1.00	41,487	93
SGA	2173	29,520	4,188,432	2530	1	4048	5.70	2.47	0.00	90.70	1.00	2280	78
CISA	173	106,467	3,912,727	42,455	5	260,432	7.16	4.80	2.68	84.60	1.00	35,058	80
CISA	173	106,467	3,912,727	42,455	5	260,432	7.16	4.80	2.68	84.60	1.00	35,058	80
CISA	173	106,467	3,912,727	42,455	5	260,432	7.16	4.80	2.68	84.60	1.00	35,058	80
CISA	21	913,837	1,328,158	913,837	6	1,228,349	8.65	5.90	714.30	84.60	1.02	NA	28
CISA	21	913,837	1,328,158	913,837	6	1,228,349	8.65	5.90	714.30	84.60	1.02	NA	28
GAA	2348	106,467	8,730,704	10,451	11	408,233	6.74	4.79	1.47	99.32	1.91	41,487	25
GAA	2348	106,467	8,730,704	10,451	11	408,233	6.74	4.79	1.47	99.32	1.91	41,487	28
GAA	2348	106,467	8,730,704	10,451	11	408,233	6.74	4.79	1.47	99.32	1.91	41,487	29
GAA	2348	106,467	8,730,704	10,451	11	408,233	6.74	4.79	1.47	99.32	1.91	41,487	29
GAA	2348	106,467	8,730,704	10,451	11	408,233	6.74	4.79	1.47	99.32	1.91	41,487	29
GAM_NGS	201	106,467	4,588,158	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	93
GAM_NGS	201	106,467	4,588,158	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	93
GAM_NGS	202	106,467	4,588,119	42,455	10	404,185	6.61	4.77	2.79	99.22	1.00	41,487	93
GAM_NGS	202	106,467	4,588,119	42,455	10	404,185	6.61	4.77	2.79	99.22	1.00	41,487	93
GAM_NGS	203	106,467	4,711,689	42,802	11	515,103	6.61	4.77	2.74	99.22	1.03	42,802	91
GARM	197	97,474	5,615,046	47,696	3	80,924	9.52	6.42	0.00	86.23	1.42	53,525	0
GARM	199	97,480	5,518,631	47,676	3	80,924	9.55	6.40	0.00	86.26	1.39	52,061	52
GARM	201	97,474	5,616,723	47,696	3	80,924	9.47	6.40	0.00	86.26	1.42	53,525	52
GARM	201	97,476	5,616,676	47,696	3	80,924	9.55	6.30	0.00	86.26	1.42	53,525	52
GARM	205	97,456	5,749,214	47,696	4	142,754	8.36	6.17	0.00	86.28	1.45	59,372	50
MIX	194	106,467	4,351,909	42,455	7	353,063	5.28	4.62	2.73	94.14	1.00	40,555	88
MIX	196	106,467	4,423,168	42,455	7	353,063	5.20	4.59	2.74	95.69	1.00	41,487	90
MIX	197	106,467	4,466,781	42,455	7	353,063	5.33	4.63	2.71	96.64	1.00	41,487	91
MIX	199	106,467	4,481,045	42,455	7	353,063	5.13	4.68	2.70	96.95	1.00	41,487	91
MIX	201	106,467	4,535,084	42,444	7	353,063	5.25	4.72	2.67	98.12	1.00	41,487	92
Metassembler	200	106,467	4,587,010	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	93
Metassembler	200	106,467	4,587,010	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	93
Metassembler	200	106,467	4,587,010	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	93
Metassembler	200	106,467	4,587,010	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	93
Metassembler	200	106,467	4,587,010	42,455	10	408,636	6.61	4.77	2.79	99.22	1.00	41,487	93
ZORRO	222	106,470	4,609,096	41,918	9	388,589	7.09	4.81	1.41	99.28	1.01	41,235	93
ZORRO	224	105,319	4,616,066	42,444	8	296,208	7.16	4.84	1.54	99.28	1.01	41,487	92
ZORRO	231	105,269	4,594,505	37,312	8	215,802	6.98	4.70	1.41	99.30	1.00	37,195	93
ZORRO	231	105,269	4,595,263	37,312	8	215,799	6.94	4.70	1.46	99.30	1.00	37,195	93
ZORRO	231	105,315	4,596,344	37,312	8	215,902	6.96	4.68	1.46	99.29	1.01	37,195	93

Table A.18: Experimental results on parameter tuning (*Hg_chr14*, genome size 88,289,540 bp). The table reports on quality of merged assembly compared to the two input assemblies. Notes: Reported statistics are for contigs; the number of mismatches/indels/Ns are per 100 Kbps;

Reconciliation Tool	Contigs (#)	Largest (bp)	Size (bp)	N50 (bp)	Misassembly (#)	Misassembly Length (bp)	Mismatches (#)	Indels (#)	N's (#)	Genome covered (%)	Duplication ratio	NGA50 (bp)	Genes (%)
ALLPATHS-LG	4469	240,773	84,416,102	38,359	109	1,384,277	67.71	21.79	54.60	78.48	1.00	27,772	63
SGA	33,695	30,350	75,492,807	3317	107	249,973	87.51	12.57	0.00	69.89	1.01	1945	89
GAA	38,155	240,773	159,891,549	10,679	216	1,634,250	71.61	22.01	28.83	79.17	1.88	27,895	82
GAA	38,155	240,773	159,891,549	10,679	216	1,634,250	71.61	22.01	28.83	79.17	1.88	27,895	82
GAA	38,155	240,773	159,891,549	10,679	216	1,634,250	71.62	22.01	28.83	79.17	1.88	27,895	80
GAA	38,155	240,773	159,891,549	10,679	216	1,634,250	71.62	22.01	28.83	79.17	1.88	27,895	82
GAA	38,155	240,773	159,891,549	10,679	216	1,634,250	71.62	22.01	28.83	79.17	1.88	27,895	82
GAM_NGS	4228	240,773	84,488,919	39,589	111	1,579,215	68.19	21.80	54.50	78.55	1.00	28,518	63
GAM_NGS	4229	240,773	84,488,720	39,493	111	1,579,215	68.19	21.80	54.48	78.55	1.00	28,613	63
GAM_NGS	4258	240,773	84,476,409	39,118	109	1,545,656	68.09	21.78	54.51	78.55	1.00	28,211	63
GAM_NGS	4318	240,773	84,468,987	39,442	110	1,485,435	67.94	21.79	54.54	78.53	1.00	28,412	63
GAM_NGS	4345	240,773	84,463,742	38,828	110	1,470,117	67.97	21.79	54.56	78.53	1.00	28,129	63
GARM	4444	240,762	106,632,698	44,830	169	3,131,454	94.72	30.40	0.06	75.60	1.31	44,140	61
GARM	4445	240,762	108,298,281	45,234	232	5,969,742	95.37	30.30	0.06	75.67	1.33	45,010	61
GARM	4459	240,762	106,210,805	44,544	144	2,787,020	93.82	31.32	0.06	75.45	1.31	43,918	61
GARM	4495	240,767	106,334,207	44,486	154	2,774,708	93.24	31.74	0.06	75.48	1.31	43,839	61
GARM	4498	240,762	106,465,991	44,515	154	2,774,773	93.62	31.11	0.06	75.52	1.31	43,918	61
Metassembler	4359	240,773	84,259,313	38,473	109	1,384,277	67.54	21.81	54.50	78.36	1.00	27,772	63
Metassembler	4359	240,773	84,259,313	38,473	109	1,384,277	67.54	21.81	54.50	78.36	1.00	27,772	63
Metassembler	4359	240,773	84,259,313	38,473	109	1,384,277	67.54	21.81	54.50	78.36	1.00	27,772	63
Metassembler	4359	240,773	84,259,313	38,473	109	1,384,277	67.54	21.81	54.50	78.36	1.00	27,772	63
Metassembler	4359	240,773	84,259,313	38,473	109	1,384,277	67.54	21.81	54.50	78.36	1.00	27,772	63