Abstract

Title of thesis: GENOME ASSEMBLY TECHNIQUES

Guillaume Marçais, Doctor of Philosophy, 2011

Thesis directed by: James Yorke and Carl Kingsford

Since the publication of the human genome in 2001, the price and the time of DNA sequencing have dropped dramatically. The genome of many more species have since been sequenced, and genome sequencing is an ever more important tool for biologists. This trend will likely revolutionize biology and medicine in the near future where the genome sequence of each individual person, instead of a model genome for the human, becomes readily accessible.

Nevertheless, genome assembly remains a challenging computational problem, even more so with second generation sequencing technologies which generate a greater amount of data and make the assembly process more complex. Research to quickly, cheaply and accurately assemble the increasing amount of DNA sequenced is of great practical importance.

In the first part of this thesis, we present two software developed to improve genome assemblies. First, Jellyfish is a fast k-mer counter, capable of handling large data sets. k-mer frequencies are central to many tasks in genome assembly (e.g. for error correction, finding read overlaps) and other study of the genome (e.g. finding highly repeated sequences such as transposons). Second, Chromosome Builder is a scaffolder and contig placement software. It aims at improving the accuracy of genome assembly.

In the second part of this thesis we explore several problems dealing with graphs. The theory of graphs can be used to solve many computational problems. For exam-

ple, the genome assembly problem can be represented as finding an Eulerian path in a de Bruijn graph. The physical interactions between proteins (PPI network), or between transcription factors and genes (regulatory networks), are naturally expressed as graphs.

First, we introduce the concept of "exactly 3-edge-connected" graphs. These graphs have only a remote biological motivation but are interesting in their own right. Second, we study the reconstruction of ancestral network which aims at inferring the state of ancestral species' biological networks based on the networks of current species.

GENOME ASSEMBLY TECHNIQUES

by

Guillaume Marçais

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2011

Advisory Committee:
Professor Jim Yorke, Chair
Professor Carl Kingsford, Co-Chair
Professor Uzi Vishkin
Professor Hector Bravo
Aleksey Zimin

Contents

List of Tables

List of Figures

Part I

Genome Assembly

Almost every cell in a living organism contains a set of DNA (DeoxyriboNucleic Acid) molecules. DNA is a double-stranded polymer made up of four different nucleotides (*bases*): adenine (A), guanine (G), thymine (T) and cytosine (C). The two strands of a DNA molecule are complementary, that is, if there is a T on one strand, there will be an A on the other. Similarly, C is a complement of G.

The goal of genome sequencing and assembly is to determine the sequence of the bases in the DNA of an organism. DNA molecules vary in length from a few thousand to hundreds of millions of bases. Current sequencing machines can read at most 2 000 contiguous bases from the ends of a DNA molecule. To determine the sequences, multiple copies of an organism's DNA are broken up into millions of fragments, called *inserts*, each 200 to 200 000 bases long. No information about the inserts' locations in the original DNA can be retained. Sequencing machines then read 50 to 1 000 bases of the ends of each insert. The sequence of bases on the end of an insert is called a *read*; if both ends of an insert are sequenced, then the pair of reads form a *mate pair*. A *library* is a set of inserts that have been selected to have approximately the same length. Sequencing centers report an estimated mean and standard deviation of the distribution of the insert lengths for each library.

Since each read represents a small fraction of an organism's DNA, the reads must be *assembled*. There are two main assembly approaches: overlap-layout-consensus and Eulerian path.

In the overlap-layout-consensus assembly paradigm, assembly software first looks for reads that have common sequence on their ends; such a pair of reads is said to *overlap*. The assembler then uses this overlap information and along with mate pair information provided by the sequencing center, to create contiguous overlaping collections of reads called *contigs*. Finally the consensus stage determines the most likely sequence by examining the multi-alignment of reads.

A contig is the largest piece of contiguous sequence that the assembler is able to reconstruct from the read data. Contigs could end for multiple reasons: ambiguities in the overlaps; missing overlaps due to poor quality or missing data; errors in data pre-processing.

Next, the assembler relies on the mate-pair information, to create larger data structures, called *scaffolds* or *supercontigs*, which are ordered collections of oriented contigs. Scaffolds contain estimates of the lengths of the *gaps* between contigs. This paradigm is used, for example, by the Celera Assembler [64, 69] and Arachne [38].

In the Eulerian paradigm, the sequencing reads are further broken down into k-mers (all the sub-sequences of length k in the reads). The assembler constructs a *De Bruijn graph*. Each vertex in the graph represents a k-mer. Vertices v_1 and v_2 are connected by a directed edge if the $(k-1)$-suffix of v_1 is a $(k-1)$-prefix of v_2. Any Eulerian path, i.e. a path containing every edge of the graph, is a potential assembly. Read and mate-pair information impose constraints on a valid Eulerian path. The final assembly is the Eulerian path most likely to represent the genome based on theses contraints. Examples of assemblers that apply this paradigm include Euler [76], AllPaths [7, 28], Soap de novo [52].

Using various laboratory techniques and targeted supplemental sequencing, it is possible to improve the quality of an assembly or selected parts of an assembly, a process called *finishing* or *gap closure*. High quality sequence generated by this finishing step is called *finished sequence*. Finished sequences are invaluable to estimate the accuracy of genome assembly procedures [9].

Different sequencing protocols influence the way the assembly is performed. A Bacterial Artificial Chromosome is a fragment of DNA sequence approximately $150 - 350\,\text{kb}$ long [89]. The BAC-by-BAC sequencing technique breaks down the assembly problem into smaller problems of a BAC size. First, the sequencing center determines experimentally a set of BACs which covers the entire genome, and, ideally,

which is as small as possible (e.g. [92]). Then, each BAC is individually sequenced and assembled. Finally, based on the overlaping sequence of the BACs, the complete sequence of the genome is recovered.

The idea behind this approach is to reduce the number of large scale misassemblies by localizing the assembly problem. The drawbacks of this approach include a significant cost in building the tiling of BACs and difficulties in patching together the assembled BACs. Moreover extracting the tiling of BAC is error prone and the BACs may not cover the entire genome. Any sequence not covered by a BAC in the tiling path will not be present in the assembly.

The whole genome shotgun sequence (WGS) sequences the entire genome at once. While assembling such dataset is more computationally expensive, it is overall simpler and cheaper.

The assembly of the *Bos taurus* genome, in Chapter 1, is the first large scale assembly project in which I was involved. My experience on this project motivated the development of the other two projects presented in this part of this thesis: Jellyfish (Chapter 2) and the Chromosome Builder (Chapter 3). Jellyfish, is a fast k-mer counter. Counting k-mers, i.e. counting the number of occurrences of every substring of length k, has many application in bioinformatics. In genome assembly, k-mer counts are used, for example, for error correcting the input reads, finding overlaps between reads and building uniquely assemblable sequences. The Chromosome Builder is a *scaffolder*, i.e. it builds scaffolds from contigs and mate pairs between these contigs. Unlike traditional scaffolders, the Chromosome Builder is also able to use information from a "marker map" to improve the quality of the scaffolds, and place the scaffolds at their correct position on their respective chromosome. A *marker map* is a set of short sequences, usually around 100 bp, whose position in the genome is known through some biological experiment.

Chapter 1

Bos taurus assembly

This chapter is based on work published in "Aleksey V Zimin, Arthur L Delcher, Liliana Florea, David R Kelley, Michael C Schatz, Daniela Puiu, Finnian Hanrahan, Geo Pertea, Curtis P Van Tassell, Tad S Sonstegard, Guillaume Marçais, Michael Roberts, Poorani Subramanian, James A Yorke, and Steven L Salzberg. A whole-genome assembly of the domestic cow, bos taurus. *Genome Biol*, 10(4):R42, 2009."

1.1 Introduction

The human genome project [101] established the feasibility of determining the genome sequences of mammalian species from shotgun sequencing data. Since its completion, other mammalian genome assemblies, including those of cow [105, 54], mouse [68], and chimp [11], have become available. Each such published assembly consists of thousands of contiguous fragments of DNA separated by gaps, where the genome sequence could not be determined. It is likely that some fragments in the final sequence have been misassembled or mistakenly repeated. Genome assemblies vary in their quality and completeness [85].

In an effort to further improve the draft genome of the cow (*Bos taurus*) sponsored by the U.S. Department of Agriculture (USDA), our group at the University of Maryland (UMD) reassembled this genome using a different assembler software. The

first assemblies of the cow genome were created by a team lead by the Baylor College of Medicine (BCM), using their assembly program ATLAS [33].

To produce their Btau4.0 assembly of the *Bos taurus* genome, Baylor used a hybrid assembly strategy, combining WGS with the BAC-by-BAC technique. This procedure follows the BAC-by-BAC approach except that the BACs are assembled independently using an extended set of reads: the BAC reads and any WGS reads that has an overlap with a BAC read.

Baylor's hybrid strategy has some advantages over a purely BAC-by-BAC assembly. Because WGS reads are cheaper to produce, this type of assembly is more cost-effective. On the other hand, such an assembly is more complex. Furthermore, it still suffers from two major problems of a BAC-based assembly: individual assembled BACs are difficult to merge and the assembly misses parts of the genome that are not covered by a BAC.

Although the *Bos taurus* read data was designed to be assembled using Baylor's hybrid strategy, we believed we could avoid many of Baylor's problems by using a WGS assembly software, the Celera Assembler (CA) [69].

1.2 Methods

In this section, I first overview the assembly strategy. I then describe the specific problems in the assembly process for which I developed and implemented solutions.

1.2.1 Assembly strategy overview

We performed the following steps to create the cow assembly. First, we cleaned up the data, in particular, by detecting and trimming the vector sequence from the reads. Then, we computed overlaps using the UMD overlapper [83] and assembled the reads into contigs and scaffolds using a modified version of the Celera assembler (version

4.0). Finally, we ordered and oriented the scaffolds along the chromosomes using a combination of two genetic maps. More details are available in the publication [105].

1.2.2 Non-random BAC read data

We say *reads are uniformly distributed* on the genome if their positions are distributed as if they were chosen at random from a uniform distribution. Two-thirds of our data consisted of WGS reads produced with Sanger sequencing technology, which are approximately uniformly distributed on the genome. BAC reads made up the remainder of our sequencing data. Although the positions of the shotgun reads from each BAC have an approximately uniform distribution within that BAC, BAC reads as a set are not uniformly distributed on the whole genome. This occurs because (i) BACs may overlap; (ii) the depth of coverage of BACs varies from 1x to 50x; and (iii) a portion of the genome is not covered by any BACs. The CA uses a statistical test based on the Poisson distribution that assumes that the reads cover the genome uniformly. However BAC reads do not satisfy this assumption. Our goal was to incorporate the BAC reads so as not to waste data, but to do so in a way compatible with our software. In this section, we describe this statistical test, explain how we modified CA to work with non-random data, and show the impact our modifications had on the quality of our assembly.

The CA first uses overlap and mate-pair information to construct *unitigs* from the reads that can be assembled unambiguously. By design, unique and repeated sequence should not appear in the same unitig. A *repeat unitig* contains a single copy of the sequence of a repeated region and it contains all reads covering every instance of the repeated region. The repeat instances could be far apart in the genome and thus the mate-pair information of the reads in repeat unitigs should not be used for scaffolding. To distinguish between unique and repeat unitigs, the CA computes the *arrival rate statistic* (A-stat) [69] for each unitig. The A-stat compares the probability

Table 1.1: The original assembly used all reads to compute the A-stat, while the modified assembly used only the WGS reads. The N50 scaffold (resp. contig) size is defined as the size k such that 50% of the genome is contained in scaffolds (resp. contigs) larger than k.

CA Assembly	Original	Modified	Difference
Scaffold Bases	2.84 Gb	2.86 Gb	+1%
N50 Contigs	67.3 Kb	74.6 Kb	+11%
N50 Scaffolds	1.71 Mb	2.06 Mb	+20%
Percent of reads used	75%	84%	+12%

that there is one copy of a particular unitig in the genome to the probability that multiple copies are present. Given that the reads are uniformly distributed on the genome, the expected number of reads starting at each base (the *arrival rate*) follows a Poisson distribution. The CA defines the global arrival rate (GAR) as $\mu = N/G$, where N is the number of reads and G is the genome length.

For a repeat unitig, the expected arrival rate is $k\mu$ where k is the number of instances of the repeat in the genome. Given an arrival rate of $k\mu$, the probability that a unitig of length U contains n reads is

$$P(k, n) = \frac{(Uk\mu)^n e^{-Uk\mu}}{n!}\,.$$

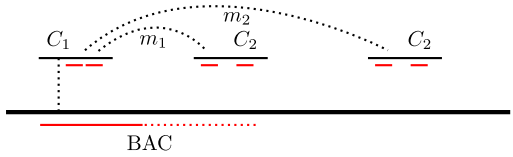
A unitig is more likely to be unique if $P(1, n) \geq P(k, n), \forall k > 1$, or equivalently if

$$A(k, n) \triangleq \frac{U}{n} - \frac{\log k}{(k-1)\mu} \geq 0\,.$$

Myers *et al.* [69] call $A(2, n)$ the A-stat. If the A-stat is positive, then one can show that $A(k, n) \geq 0, \forall k \geq 2$. Hence a positive A-stat indicates a greater likelihood that the unitig is unique in the genome.

We modified the A-stat computation in the CA to ignore the non-random BAC reads. The assembly using the modified A-stat computation resulted in 11% longer

Figure 1.1: Contig C_2 has conflicting mate-pair information, m_1 and m_2. The position implied by m_1 is consistent with the BAC in red and contig C_2 will be placed at this position in the assembly.



N50 contig size and the assembly used 12% more reads (see Table 1.1 on the preceding page).

1.2.3 Mapping sequence to the chromosomes

We used two marker maps to place our scaffolds onto chromosomes: the fingerprint map (FPMap) from the Michael Smith Genome Sciences Center, containing 124 831 markers, and the composite map (CMap) [92] containing 17 254 markers. We combined the maps into the Combined FPMap (CFPMap), and were able to align 92% (131 579) of the 142 085 CFPMap markers, to the UMD assembly. We split 19 scaffolds because inconsistencies with marker positions. Finally, we used the CFPMap to place 19 765 scaffolds onto chromosomes.

An additional 13 116 scaffolds have mate-pair links to the already placed scaffolds. However, these mate-pair links are conflicting and indicate multiple possible placements.

If for all reads in a given BAC, we obtain only one group of the correct length (< 400 kb), then this group's position is designated as the location of the BAC in the BAC map. For each of the 13 116 scaffolds with ambiguous placement, by examining the BAC reads they contained, we chose the placement that agrees the most with the BAC map.

Figure 1.1 shows such a case. Contig C_1 has been placed and every reads from the BAC shown in red (including the ones in contig C_1) are within 400 kb. Contig

Table 1.2: Comparison of UMD and BCM assemblies based on overall size and on placement of composite map (Cmap) markers.

	BCM	UMD	Difference
Sequence in chromosomes	2.47 Gb	2.61 Gb	+5.7%
N50 contig size (at 1.235 Gb)	82 Kb	93 Kb	+13%
CMap markers aligned to assembly	13 699	14 620	+6.7%

C_2 has not yet been placed, it contains reads from the red BAC, and, according to its mate-pair links, it could be placed in multiple places. We choose the placement for C_2, if one exists, such that the group of reads from the red BAC are all within 400 kb, including the reads in C_2.

Using this technique, we placed an additional 3 199 scaffolds, out of 13 116, representing 80 Mb (+3%) of sequence.

1.3 Results

1.3.1 Comparing the assemblies

Table 1.2 compares the quantitative statistics of the BCM and UMD assemblies. Our assembly has 5.7% more sequence placed onto chromosomes. This is caused in part by BCM's failure to incorporate the entire WGS data set into their assembly. The ATLAS assembly software is only able to utilize WGS reads should they fall within a BAC. Based on our assembly, an estimated 2% of the genome is not covered by BACs.

1.3.2 Comparing to finished sequence

BCM finished six BACs and deposited them in the public repository GenBank at the National Center for Biotechnology Information (NCBI). These six finished BACs

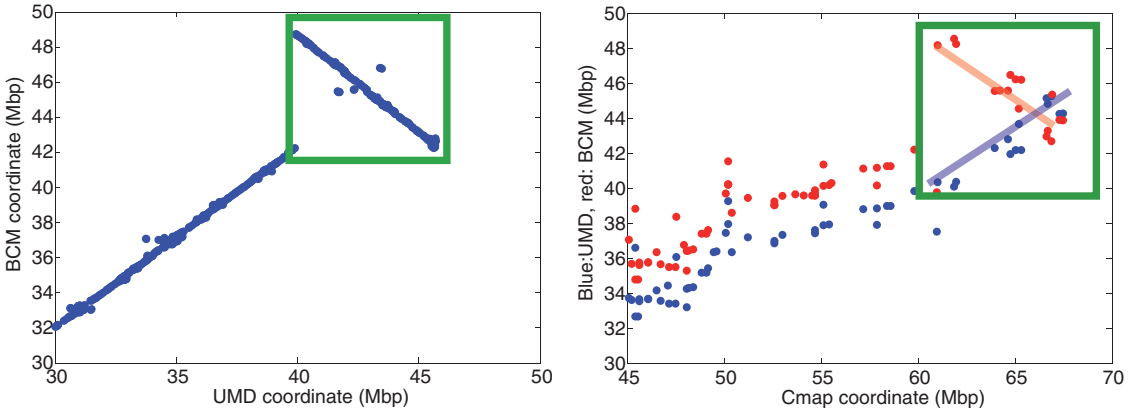
Table 1.3: Comparison of the percentage of the sequence of the BACs covered by the assemblies and the number of errors.

	BCM	UMD
Sequence coverage	90.7%	95.4%
# error / 10 Kb	95.82	58.69

are not part of either assembly, and can thus be used to compare the completeness and accuracy of the assemblies. We aligned the contigs from each assembly to the finished BACs, and measured the length and quality of the alignment. The amount of finished sequence that does not align to the BCM assembly is twice as long as that of the UMD assembly (Table 1.3). In the sequence that does align, the error rate of the BCM assembly is 66% higher.

1.3.3 Single Nucleotide Differences (SNDs)

In a base-by-base comparison, the UMD and BCM assemblies have over 2.0 million single-nucleotide differences (SNDs). Some of these might be valid haplotype differences, in which the two assemblies are both correct, while others might be errors. We focused our analysis on a subset of positions where the underlying read data indicated that the position was highly likely to be homozygous, because a large majority (or all) reads agreed with one another. We also required that each SND was flanked by 50 bp exact matches in both assemblies, which reduced the set of SNDs to 389 015. We then looked for cases where no more than one read confirmed one assembly, and all other reads (at least three) confirmed the other assembly. The UMD assembly contains 10 636 instances of these apparent errors versus 30 750 in the BCM assembly. Thus, there were approximately three times more apparently erroneous SNDs in the BCM assembly.



(a) (b)

Figure 1.2

1.3.4 Large scale disagreements

There is an alternative set of markers developed for studying biologically significant Single Nucleotide Polymorphisms (SNPs) in the *Bos taurus* genome. This set of markers is called SNP50 and is currently used for genotyping cattle [59]. It contains approximately 50 thousand 51-base marker sequences that are unique in the genome. Over 48 thousand markers aligned to the UMD assembly. This set of markers was constructed from the BCM assembly and thus the positions of all markers are known exactly in the BCM assembly. We then compared the positions of markers in the two assemblies and look for blocks of consistently disagreeing markers that may indicate large scale differences between the two assemblies. Ten of the 30 chromosomes contain one or more large ($> 500\,\text{kb}$) discrepancies, 19 in all, primarily inversions but also deletions and translocations. Figure 1.2a illustrates an inversion spanning 2.5 Mb on chromosomes 27. In this case, as in all other large discrepancies, the Cmap data support the UMD assembly, as shown in Figure 1.2b.

1.4 Conclusion

The *Bos taurus* assembly project motivated the development of genome assembly software in two directions: faster and memory efficient mer counting, and whole chromosome scaffolding. The need for faster mer counting was reinforced by later work on assemblies of second generation sequencing data such as the domestic turkey *M. gallopavo* [13].

The current trend in sequencing technology, referred to as Second Generation Sequencing (SGS), is to produce higher coverage (e.g. 30x to 50x) by short reads [80, 65, 86]. In comparison, Sanger sequencing required 6x to 10x coverage. SGS technologies greatly reduce the cost of sequencing data and effectively trade longer reads for deeper coverage. Seemingly simple but important tasks, such as counting the frequency of sub-strings of a given length k (called a k-mer) in the reads, become challenging on the data sets generated by SGS. In Chapter 2 we describe a fast and memory efficient software, called Jellyfish, that computes the k-mer frequencies on large data sets.

Assembly programs typically produce scaffolds; contigs are ordered and oriented relative to each other within a scaffold. The more useful output for biological studies is chromosome sequences where the contigs are ordered and oriented on the chromosome. This is usually achieved by post-processing the scaffolds using a marker map, which is a laborious process. This prompted the development of the Chromosome Builder software described in Chapter 3 that automates this procedure by using mate-pair and marker information concurrently.

Chapter 2

A fast, lock-free approach for efficient parallel counting of occurrences of k-mers

A version of this chapter first appeared in "Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011."

2.1 Introduction

Given a string S, we are often interested in counting the number of occurrences in S of every substring of length k. These length-k substrings are called k-mers and the problem of determining the number of their occurrences is called k-mer counting.

Counting the k-mers in a DNA sequence is an important step in many applications. For example, genome assemblers using the overlap-layout-consensus paradigm, such as the Celera [69, 64] and Arachne [38] assemblers, use k-mers shared by reads as seeds to find overlaps. Statistics on the number of occurrences of each k-mer are first computed and used to filter out which k-mers are used as seeds. Such k-mer count statistics are also used to estimate the genome size: if a large fraction of k-mers occur c times, we can estimate the sequencing coverage to be approximately c and derive an estimate of the genome size from c and the total length of the reads. In addition, in most short-read assembly projects, errors are corrected in the sequencing reads to improve the quality of the final assembly. For example, Kelley et al. [42] use k-mer frequencies to assess the likelihood that a misalignment between reads is a

sequencing error or a genuine difference in sequence. A third application is the detection of repeated sequences, such as transposons, which play an important biological role. De novo repeat annotation techniques find candidate regions based on k-mer frequencies [45, 8, 34, 48]. The counts of k-mers are also used to seed fast multiple sequence alignment [21]. Finally, k-mer distributions can produce new biological insights directly. Sindi et al. [90] used k-mers frequencies with large k ($20 \leq k \leq 100$) to study the mechanisms of sequence duplication in genomes.

We consider the k-mer counting problem in the context where the input string S is either one DNA sequence or a concatenation of many DNA sequences, and the alphabet is $\Sigma = \{A, C, G, T\}$. The main application to which we apply our new k-mer counting algorithms here is counting k-mers in sequencing reads from large genome sequencing projects where the length n of the sequence to process is equal to the length g of the genome sequenced times the coverage c of the sequencing project ($n = g \cdot c$). Recent sequencing techniques, using shorter reads with a much deeper coverage [86], generate large amounts of sequence and provide a major challenge for genome assembly and for k-mer counting. For example, the giant Panda [51] sequencing project generated 73x coverage yielding 176 Gb of sequence, much larger than the 5x–10x coverage a sequencing project using traditional Sanger methodology would generate.

Of course, k-mer counting can be naively implemented using a simple hash table, where keys are the k-mers and the stored values are the counts. However, this strategy is extremely slow and implementing multi-threaded access to the hash table via standard locking mechanisms results in slower performance than a single-threaded implementation [81, 61]. Typically, more advanced k-mer counters such as Tallymer [45] have been based on the suffix array data structure. Despite the recent algorithmic progress to compute the suffix array of a string, it remains a relatively expensive computational operation. Moreover, in sequencing applications, memory

requirements for a suffix array grow linearly with the product of the coverage and genome size. Meryl, the k-mer counter built into the Celera assembler, uses a sorting-based approach that sorts the k-mers in lexicographical order, however sorting billions of records quickly with limited memory is a challenging problem.

In order to process this huge increase in amount of sequence, the increasing availability of parallelism must be exploited. While the raw power of each processing core has leveled off, the number of cores per CPU is rising. Shared memory machines with 16 or more cores and 4 GB or more memory per core are commonly available in research facilities. Creation of parallel algorithms taking advantage of such a large number of cores in a shared memory environment is both a challenge and an opportunity: on the one hand, current programming paradigms either do not take advantage of the parallelism available or are difficult to implement; on the other hand, very fast programs using fine grain parallelism can be implemented.

In recent years, the MapReduce programing paradigm [15] has been used to harness the computational power of large clusters of machines. The problem of counting k-mers is easy to implement on top of a MapReduce cluster, but the straightforward implementation, where the map operation emits all k-mers associated with a 1, incurs a large overhead. To reduce this overhead, one needs to increase the amount of work done on each node at the map stage by, for example, using a fast single-machine k-mer counter like Jellyfish on each node. The map operation then emits pairs of k-mers and their counts on a subset of the data. In that sense, the use of the MapReduce paradigm and optimized k-mer counters like Jellyfish are orthogonal.

Other processing architectures such as GPU computing have also been recently exploited for achieving faster parallel execution. However, the widespread availability of multi-core CPUs make them the first and easiest choice to program, and this is likely to remain true for some time to come. CPU development is not staying idle, and facilities, such as the CAS operation (Section 2.3.2) and the SSE extension

(Section 2.3.4), are available in all modern CPUs to help achieve greater parallel execution.

Our k-mer counting algorithm is designed for shared memory parallel computers with more than one core. It uses several lock-free data structures that exploit a widely available hardware operation called "compare-and-swap" (CAS) to implement efficient shared access to the data structures. In particular, Jellyfish uses lock-free queues [61, 46] for communication between worker threads and a lock-free hash table [62, 81] to store the k-mer occurrences counts (see Section 2.3.2). Unlike a traditional data structure where access by multiple threads must be serialized by the use of a lock, lock-free data structures can be used concurrently by many threads while still preserving a coherent internal state. The lock-free data structure also provides better performance by increasing the amount of time spent in parallel execution versus serial execution. The efficient parallel data structures allow Jellyfish to count k-mers much faster than existing k-mer counting software. In our testing on a large assembly project (Section 2.4.1), Jellyfish takes minutes instead of hours.

Jellyfish is also very memory efficient. It implements a key compression scheme that allows it to use a constant amount of memory per key in the hash table for most applications, regardless of the length k of the k-mers counted (see Section 2.3.4). It also uses a bit-packed data structure to reduce wasted memory due to memory alignment requirements (see Section 2.3.3). In addition, unlike with suffix arrays, the expected storage requirement for the hash table does not grow linearly as the coverage increases. This property makes hash tables extremely attractive for use in the context of short-read sequencing projects. Jellyfish can be more than twice as memory efficient as other programs (Section 2.4.1).

Our results show that Jellyfish can count k-mers an order of magnitude or more faster than existing programs (Section 2.4.1). This suggests that lock-free hash tables are valuable for k-mer counting and possibly also in other problems where

large strings must be processed. In addition, Jellyfish's novel memory-efficient key compression approach (Section 2.3.4) allows the hash table to use a similar amount of storage as suffix arrays in most common uses. Finally, our implementation of the lock-free hash table is general and may be of use in other applications.

2.2 Methods

2.2.1 Execution Profiling

All testing and timing was performed on a 64-bit x86 AMD Opteron machine with 32 cores at 2.5 GHz and 256 GB of RAM running Linux kernel version 2.6.31. The disks are RAID-10 with sustained write throughput of 260 MB/s. The time is the wall clock time measured with the GNU `time` utility averaged over 5 runs (except runs that exceed one hour which are run only once).

To measure the memory usage, the programs were run under `strace` which logs every system call made by a process and its threads. The logs of the `strace` were parsed to compute the amount of memory used by the process by looking for the following system calls, which are the only calls available to request memory from the kernel on a Linux system: `brk`, `mmap`, `munmap`, and `mremap`. In addition, the script counts only the memory areas which are writable for the process. Read-only pages are not counted as most of them correspond to shared libraries. In some cases, this undercounts the true memory usage. For example, Tallymer maps the entire input sequence into read-only memory and accesses it in a random fashion, so all these read-only pages need to be present in memory. Jellyfish also maps the entire input sequence into read-only memory, but the sequence is accessed in a sequential fashion and only the current page needs to be present in memory. Memory usage and running time are measured in different runs as the `strace` mechanism can affect the running

time of IO heavy programs. The overall CPU and IO load are measured with the Linux `vmstat` utility.

2.2.2 Sequence Data Sets

The *M. gallopavo* reads were taken from the Turkey genome assembly project [13]. These short reads, from Roche 454 and Illumina GAII technology, total approximately 24 Gb of sequence for a genome of 1 Gb.

The sequence of *H. sapiens* (3 Gb), *D. ananassae* (3.5 Gb of reads, genome of 189 Mb), and *C. burnetii* (35.6 Gb of reads, genome of 1.99 Mb) were downloaded from the NCBI. The human sequence is the reference genome hs_ref_GRCh37 and, unlike all other data sets, consists of assembled chromosomes instead of sequencing reads.

2.2.3 Comparing with Existing k-mer Counters

We used two versions of Meryl (5.4 and 6.1) from the kmer package of the Celera assembler [64], with all default options and 32 threads. To work around an issue with Meryl version 5.4 when dealing with a multi-fasta file, all the reads where concatenated with a single N as a separator. The original multi-fasta file was used with Meryl version 6.1.

Tallymer comes from version 1.3.4 of the genometools package [45]. It was run as shown in the example of Tallymer's documentation given in the distribution. The Tallymer subroutine `suffixerator` used options `-dna -pl -tis -suf -lcp`, and the subroutine `tallymer mkindex` used options `-minocc 1 -maxocc 10000000000` so that all k-mer counts would be written to disk.

2.3 Algorithm

2.3.1 A Fast k-mer Hash Table

We design a light-weight, memory efficient, multi-threaded hash table for the k-mer counting problem. A hash table [95] is an array of (key, value) pairs, and, when applied to k-mer counting, *key* is conceptually the sequence of the k-mer, and value is the number of times that k-mer occurs. The position in the hash table of a given *key* is determined by a hashing function hash and a re-probing strategy to handle the case when two distinct keys map to the same position. In Jellyfish, if M is the length of the hash table, the i^{th} possible location for a given mer m is:

$$\text{pos}(m, i) = \text{hash}(m) + \text{reprobe}(i) \mod M. \tag{2.1}$$

In our implementation, we maintain the length n of the hash table to be a power of two, $M = 2^\ell$ for some ℓ, and the key representing the k-mer is encoded as an integer in the set $U_k = [0, 4^k - 1]$. The function hash is a function mapping U_k into $[0, M - 1]$. The design of a function hash is described in Section 2.3.4.

When a new mer is added to the hash table, we attempt to store it in $\text{pos}(m, 0)$, and if that position is already filled with a different key, we try $\text{pos}(m, 1)$ and so on up to some limit. Here, we use a quadratic reprobing function: $\text{reprobe}(i) = i(i + 1)/2$. This reprobing function has a good behavior with respect to the usage of the hash table [95] while not growing too fast, which is important for quickly sorting the hash table elements when writing the results to disk (see Section 2.3.5).

This straightforward standard scheme is both extremely slow when parallelized in the typical way using locks, and memory inefficient. In order to make it practical, we implement a lock-free strategy for allowing parallel insertions of keys and updates

to values. We also design an encoding scheme to limit the storage used for both *key* and its associated counts. These are described in the following sections.

2.3.2 Updating the Lock-free Hash Table

A lock, such as POSIX's `pthread_mutex`, can serialize access to the hash table and permits its use in a multi-threaded environment. However, if such a lock is used no concurrency is achieved, and therefore there is no gain in speed in the updates of the hash table. In addition, the overhead of maintaining the lock is incurred.

To allow concurrent update operations on the hash, we implement a lock-free hash table with open addressing [81]. Such lock-free hash tables exploit the "compare and swap" (CAS) assembly instruction that is present in all modern multi-core CPUs. The CAS instruction updates the value at a memory location provided that the memory location has not been modified by another thread. Technically (see Algorithm 1), a CAS operation does the following three operations in an atomic fashion with respect to all of the threads: reads a memory location, compare the read value to the second parameter of the CAS instruction and if the two are equal, write the memory location with the third parameter of the CAS instruction. If two threads attempt to modify the same memory location at the same time, the CAS operation can fail. The CAS operation returns the value previously held at the memory location. Hence, one can determine if the CAS operation succeeded by checking that the returned value is equal to the old value. Unlike a lock which serializes the access to some shared resource, the CAS operation only detects simultaneous access to a shared memory location. It is then the responsibility of the calling thread to take appropriate action in the event that a conflict has been detected.

The main operation (Algorithm 2) supported by the hash is to increment the value associated with a *key* without using any locks. The value increment algorithm

Algorithm 1 Compare-And-Swap operation

```
    procedure CAS(location, oldvalue, newvalue)
2:      currentvalue ← value at location
        if currentvalue = oldvalue then
4:          value at location ← newvalue
        end if
6:      return currentvalue
    end procedure
```

works in two steps. First it finds the location in the hash table that already holds the key or it claims an empty slot to store the key if the key is not present in the hash table. Second, it increments the value associated with the key.

Lines 1–7 in Algorithm 2 accomplish the first step. It finds an appropriate slot using the hash function and then does a CAS operation assuming that the entry in the hash is empty. If the returned value of the CAS operation is either EMPTY or equal to *key*, then that position is used for storing the key. Otherwise, there is a key collision: the reprobe value is incremented and we start over. The procedure fails if the maximum number of reprobes has been reached. Lines 8–12 accomplish the second step: they increment the value in an atomic way again using the CAS operation.

Two assumptions particular to k-mer counting simplify the design of the hash table. First, no entry is ever deleted and there is no need to maintain special information about deleted keys, such as tombstones [81]. Second, for k-mer counting the required size of the hash table should be easy to estimate, or potentially the entire available memory is used. Hence, in the event that the hash table is full, it will be written to disk instead of doubling its size in memory [88, 26]. See Section 2.3.5 for more details about this second assumption.

Algorithm 2 Atomic hash increment

```
    procedure increment(key, value)
 2:     K: array of keys
        V: array of values
 4:     i ← 0                                                    ▷ Claim key
        repeat
 6:         if i ≥ max_reprobe then
                return false
 8:         end if
            x ← pos(key, i)
10:         i ← i + 1
            current_key ← CAS(K[x], EMPTY, key)
12:     until current_key = EMPTY or current_key = key
        cval ← V[x]                                              ▷ Increment value
14:     repeat
            oval ← cval
16:         cval ← CAS(V[x], oval, oval + value)
        until cval = oval
18: end procedure
```

2.3.3 Reduced Memory Usage for a Hash Entry

Our implementation uses a bit-packed data structure, i.e. entries in the hash table are packed tightly instead of being aligned with computer words. Albeit more complex to implement, especially in concert with the word-aligned CAS operation, and incurring a small computational cost, such bit-packed design is much more memory efficient and makes further memory saving schemes (variable length field and key encoding) worthwhile.

In addition, using a value field large enough to encode the number of occurrences of the most highly repeated k-mer is a waste of memory. Typically, with a deep coverage sequencing of a genome and a sufficiently large k, the majority of k-mers appear only once, as they are unique due to sequencing errors and because most genomic sequences are not composed of repeats. Most of the remaining k-mers occur approximately c times, where c is the sequencing coverage. A small number of k-mers, depending on the repetitiveness of the genome, occur a large number of times. To

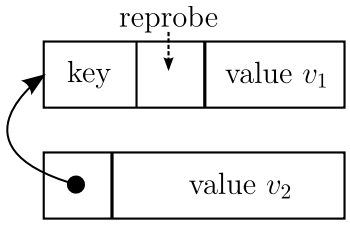


Figure 2.1: The two possible $(key, value)$ pair entries with key encoding (top) and extended value field (bottom).

account for this, Jellyfish uses a small value field and allows a *key* to have more than one entry in the hash table: key, v_1 and key, v_2. The value associated with this key is then the number obtained by the concatenation of the bits $v_1 v_2$. Moreover, to avoid the repetition of the key in the second entry, we only store a pointer (encoded as a number of reprobes) back to the previous entry. The now unused bits in the key field are used by the value field as shown in Figure 2.1.

2.3.4 Space-Efficient Encoding of Keys

The fact that an entry occurs at a known position in the hash table can be exploited to compactly store keys in the hash table in order to save a significant amount of additional memory. We choose a function $f : U_k \to U_k$ that is a bijection for which we can easily compute both f and its inverse, and set $\text{hash}(m) = f(m) \mod M$. The length $M = 2^\ell$ of the hash table is a power of 2, and the modulo M operation in the definition of the hash and pos functions (Equation 2.1) merely selects the ℓ lower bits of the sum $f(m) + \text{reprobe}(i)$. Hence, provided the value of $\text{reprobe}(i)$ is known, the position of a (key, value) pair in the hash table already encodes for the lower ℓ bits of $f(m)$. Therefore, we store the $2k - \ell$ higher bits of $f(m)$ concatenated with bits representing the reprobe count $i + 1$ in the key field of the hash. We use $i + 1$ rather than i since 0 is reserved to indicate the entries that are still empty (EMPTY in Algorithm 2).

Conversely, given this content of the key field at position x, it is easy to find the sequence of the corresponding k-mer that is stored at this position. The key field contains the $2k - \ell$ high bits of $f(m)$ and the number of reprobes i. The lower ℓ bits of the $f(m)$ can be therefore be retrieved by computing $x - \text{reprobe}(i) \mod M$. Finally, the k-mer m can be recovered by computing the inverse of f.

This scheme requires little modification to Algorithm 2. In particular, the keys do not need to be computed using the procedure described in the previous paragraph in order to be tested for equality (line 7). This is because if the content to be stored in the key field at a given position x for two k-mers m_1 and m_2 are equal, then by definition the reprobe value for both k-mers, the $2k - \ell$ higher bits of $f(m_i)$ and their position in the hash are equal, thus $f(m_1) = f(m_2)$ and $m_1 = m_2$ by the assumption that f is a bijection.

For the bijective f function, we use $f(m) = A \cdot m$, where A is a $2k \times 2k$ invertible matrix on $\mathbb{Z}/2\mathbb{Z}$. Here, m and $A \cdot m$ are interpreted either as integers or as $2k$ binary vectors. Let $\mathcal{H} = \{x \mapsto A \cdot x \mod 2^\ell \mid A \text{ is invertible}\}$ be the set of all hash functions. We will show that this set is almost an universal set of hash functions [95] in the following sense: the size of $\mathcal{H}$ and the number N of matrices A for which $A \cdot x \equiv A \cdot y \pmod{2^\ell}$ satisfy $N \approx |\mathcal{H}|/2^\ell$, provided that $2^{2k} \gg 1$ and $2^{2k-\ell} \gg 1$. In other words, the definition of the universal set of hash functions is satisfied within a small error, and the property of having few expected collisions is preserved by this approximation. The cases where this approximation breaks are the "easy" cases corresponding to a small number of possible k-mers (2^{2k} close to 1) or a hash table big enough to contain almost all the k-mers ($2^{2k-\ell}$ close to 1).

An invertible matrix A is a sequence of vectors $v_1, \ldots, v_{2k}$ such that the i^{th} vector is not in the space spanned by the vectors $v_1, \ldots, v_{i-1}$. This vector space has

size 2^{i-1} and the number of choices for v_i is $2^{2k} - 2^{i-1}$. Hence

$$|\mathcal{H}| = \prod_{i=0}^{2k-1} (2^{2k} - 2^i) = (2^{2k} - 1) \prod_{i=1}^{2k-1} (2^{2k} - 2^i). \tag{2.2}$$

For a given pair (x, y), let N be the number of invertible matrices for which $Ax \equiv Ay \pmod{2^\ell}$, or equivalently $Az \equiv 0 \pmod{2^\ell}$ by setting $z = x - y$.

Let B be an invertible matrix such that $z = Be_1$, where $e_1 = (1, 0, \ldots, 0)$, and let $C = AB$. B can be constructed by setting the first column to z and choosing the remaining columns as above to make B invertible. Then the matrix C is invertible if and only if A is invertible and $Az = ABe_1 = Ce_1$. Hence, N is the number of invertible matrices C for which $Ce_1 \equiv 0 \pmod{2^\ell}$.

Therefore there is only $2^{2k-\ell} - 1$ choice for v_1, the first column of C, and the number of choices for $v_i, i > 1$, is unchanged. Hence

$$N = (2^{2k-\ell} - 1) \prod_{i=1}^{2k-1} (2^{2k} - 2^i). \tag{2.3}$$

Provided that $2^{2k} \gg 1$ and $2^{2k-\ell} \gg 1$, the relation $N \approx |\mathcal{H}|/2^\ell$ holds.

The matrix A is chosen by iteratively drawing uniformly a random matrix out of the 2^{4k^2} possible binary matrices of this size, until it is not singular. The proportion of invertible matrices is

$$P_k = \frac{|\mathcal{H}|}{2^{4k^2}} = \prod_{i=1}^{2k} \left(1 - \frac{1}{2^i}\right). \tag{2.4}$$

P_k is a decreasing sequence. Moreover, $\ln P_k$ is a series which sumhand is $\ln(1-2^{-i}) \sim 2^{-i}$, hence $\ln P_k$ is converging to a finite limit and P_k is converging to a positive limit. Numerically, the limit of P_k is > 0.28. Hence, by random drawing, an invertible matrix will be found in an expected 4 steps regardless of the size k. Faster algorithms to find an invertible matrix exist [82], but would have no impact on the execution speed

of Jellyfish. Thus an invertible, bijective hash function that is efficiently computable and that reduces the storage per key significantly is achieved.

To compute the binary matrix product $A \cdot m$, we use the Streaming SIMD Extensions (SSE) instruction set of modern processors, if available. SSE instructions work on large registers (128 bits), treating them as vectors (e.g., of four 32-bit integers or two 64-bit integers). An SSE instruction performs the same operation on each element of the vector (or on each pair of elements of a pair of vectors) in parallel. For a 44×44 binary matrix A required to hash 22-mers, the SSE implementations computes 34.5 million multiplications per second on our test system versus 19.4 million multiplications per second for the C++ implementation that does not use SSE.

Surprisingly, in many applications, the above scheme uses an amount of space per key that is independent of the length of the k-mer and the length of the input string. Often k is chosen so that the event that a given k-mer appears more than once in the input sequence is significant. So k is chosen large enough for the probability of a k-mer to appear twice in a random string of length n to be less than some constant threshold p (e.g. $p < 1\%$). We will show that in this case the length k and the size of the hash table $M = 2^\ell$ are such that the size of the key field $|key|$, which contains the $2k - \ell$ high bits of $f(key)$ and the reprobe count, is independent of k and n.

Suppose there are n k-mers in the input chosen at random, then each has an expected number of occurrences of $\mu_k = 4^{-k}n$. The number of occurrences of a k-mer follows a Poisson distribution with mean μ_k and k is chosen so that

$$\Pr(k\text{-mer is repeated}) = 1 - (1 + \mu_k)e^{-\mu_k} \leq p. \tag{2.5}$$

Because $1-(1+\mu_k)e^{-\mu_k} \leq 1-(1+\mu_k)(1-\mu_k) = \mu_k^2$, the condition $\mu_k \leq \sqrt{p}$ implies the condition in (2.5). Solving for k gives

$$k \geq \lceil \log_4 n - \log_4(\sqrt{p}) \rceil = \left\lceil \frac{\log_2 n}{2} - \frac{\log_4(p)}{2} \right\rceil . \tag{2.6}$$

On the other hand, to accommodate all the k-mers in the input, the size of the hash table $M = 2^\ell$ satisfies $\ell \geq \lceil \log_2 n \rceil + 1$. Hence, for the smallest choice of k that satisfies (2.6) we have near equality in (2.6), and the number of bits to store for each key is

$$\begin{aligned} |key| &= 2k - \ell + \lceil \log_2(max\_reprobe + 1) \rceil & (2.7) \\ &\leq \lceil \log_4(p) \rceil + \lceil \log_2(max\_reprobe + 1) \rceil + 3 \,, & (2.8) \end{aligned}$$

which is independent of k and n.

2.3.5 Fast Merging of Intermediate Hash Tables

Once computed, the hash table is written to disk as a list of (key, value) records. The list is sorted according to the lower l bits of the hash value of the mers, which is $\text{pos}(m, 0)$, and ties are broken lexicographically. Sorting the output has the advantage that the results can be queried quickly using a binary search. More interestingly, it has the advantage that it allows two or more hash tables to be merged into one easily. This situation occurs when there is not enough memory to carry out the entire computation and intermediary results are saved to disk. Jellyfish will detect when the hash table needs to expand beyond the available memory and will instead write the current k-mer counts to disk, clear the hash table, and begin counting afresh. The intermediate results can be merged in limited memory as described below.

In memory, the entries in the hash table are loosely sorted in the following sense that can be exploited to sort the output in linear time. Let $\text{pos}(m)$ be the final position of a mer m in the hash table. Then $\text{pos}(m) = \text{pos}(m, i)$ for some $i \in [0, \mathit{max\_reprobe} - 1]$. If $\text{pos}(m_1, 0) + \text{reprobe}(\mathit{max\_reprobe}) < \text{pos}(m_2, 0)$, then $\text{pos}(m_1) < \text{pos}(m_2)$. Hence, in order to sort the output, we only need to resolve the proper ordering of the entries within a window of length $\text{reprobe}(\mathit{max\_reprobe})$, which is a constant with respect to the size of the hash table, the input size, and k. To do so, we create a min-heap of size $\text{reprobe}(\mathit{max\_reprobe})$ using the ordering $\text{pos}(m_1, 0) < \text{pos}(m_2, 0)$ and lexicographic order to break ties. The elements to write out to disk are read from the head of the heap, and as elements are removed from the heap, new elements are read from the hash table and inserted in the heap.

This sorting of the output is run in parallel to the writing to disk. The hash table is divided into m slices and thread i ($0 \leq i < \mathit{num\_threads}$) will process the slices numbered $i + j \cdot \mathit{num\_threads}$. The sorted result is buffered in each thread. To guarantee that the final output is in correct global order, the threads are organized in a token ring: only the thread which owns the token is allowed to write its own buffer to disk, thread 0 begins with the token and thread i will pass the token to thread $(i + 1) \mod \mathit{num\_threads}$. Because writing to a single disk is inherently a serial process, no efficiency is lost by having only 1 thread write to disk at a time.

2.3.6 Distributed Locks to Reduce Contention When Writing Table to Disk

When a thread fails to add a key into the hash table because the table is full, the hash table is written to disk and reinitialized. For data consistency, all threads must be prevented from making any updates to the hash table while it is written to disk. A reader-writer lock (e.g. POSIX's `pthread_rwlock`) would suffice. However, when the number of contentions is high, this performs very poorly.

Instead, we implement a distributed reader-writer lock where the frequent case (acquiring a read lock) is optimized as much as possible at the expense of the infrequent case (acquiring a write lock). Functionally, the distributed lock behaves to each thread i as a distinct reader-writer lock $rwlock_i$. For a thread to make an update to the hash table, it only needs to acquire a read lock of its own lock, $rwlock_i$. On the other hand, to write the hash table to disk, a thread i is required to acquire a write lock on all of the locks, $rwlock_j \, \forall j$. In this scheme, the frequent case involves only acquiring a lock with no contention, which is fairly fast.

The implementation again uses the CAS operation instead of POSIX `pthread_rwlock` reader-writer locks. Each thread maintains a status variable which can have three states: FREE, INUSE, BLOCKED. The frequent non-contentious case is as follows: before an update, a CAS operation is made to change the status from FREE to INUSE. In case of success, the read lock is considered acquired and the thread can proceed with the update. After the update, a compare-and-swap operation is made to change the status from INUSE to FREE. In case of success, the read lock is considered released and the thread is done. In this frequent non-contentious case our implementation incurs only the cost of two compare-and-swap operations.

A thread that discovers a full hash table when it tries to add a key will set the status variable of every other thread to BLOCKED. Using a condition variable, it will then wait for every thread that was in the INUSE state to finish their update, and then proceed to write the hash table to disk.

While the writing is occurring, every thread's status variable will be BLOCKED and any thread will fail in an attempt to to change its status from FREE to INUSE using the CAS operation. If this occurs, the thread waits for the writing of the hash to disk to be finished (its status changed from BLOCKED to FREE). If a thread fails to change its status from INUSE to FREE it notifies, using the condition variable, the thread that wants to write the hash to disk, that it is done with its update.

2.3.7 Analysis of Running Time

The time to compute one hash value via matrix multiplication is $O(k)$ assuming the k-mer fits in one machine word, and the time to insert one k-mer in the hash table is $O(k + max\_reprobe)$. To tally n k-mers in the hash takes $O(n(k + max\_reprobe))$ time. With the choice of quadratic reprobing, the size of the min-heap used to sort the hash table while writing to disk is $O(max\_reprobe^2)$. Writing n elements to disk involves n `insert` and `deleteMin` operations on the min-heap, hence a cost of $O(n \log(max\_reprobe))$. Hence creating the hash tables takes time linear in n.

In the case where t intermediary hash tables of size $s_i, 1 \leq i \leq t$ with $\sum_{i=1}^{t} s_i = n$ were written to disk, the time to create all t hash tables is $O(n(k + max\_reprobe + \log(max\_reprobe)))$. The time to merge the t hash tables is $O(n \log t)$. If a large amount of memory is available and the number of hash tables created is constant ($t = O(1)$), then the total runtime is linear in n. In this case, our algorithm is similar to counting sort [87, 95] where the array counting the number of occurrences of each element to sort is replaced by a hash table.

At the other extreme, if a small amount of memory is available and the number of hash tables created is proportional to n, then the total runtime is $O(n \log n)$. In this later case, the theoretical worst-case performance of the algorithm has degenerated to that of a heap sort (since the time to merge now dominates), although in practice the memory usage and running time will be significantly faster.

2.4 Results

2.4.1 Speed and Memory Usage on Turkey Sequencing Reads

The memory usage and timing for counting k-mers on sequencing reads of the 1 GB Turkey genome for various levels of coverage are shown in Figures 2.2 and 2.3.

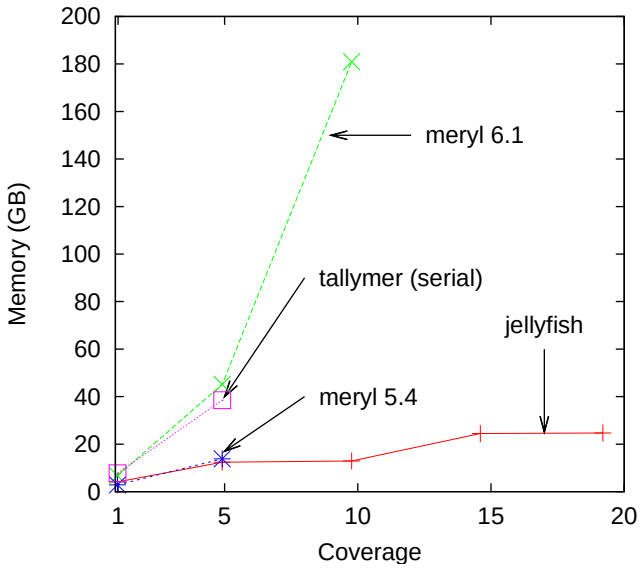


Figure 2.2: Memory usage for various levels of sequencing coverage on reads generated during the Turkey genome project when counting 22-mers. Except for Tallymer (which is inherently single-threaded) all programs were run using 32 threads. The memory usage for the serial and 32-thread versions of Jellyfish is almost identical (results are shown using 32 threads).

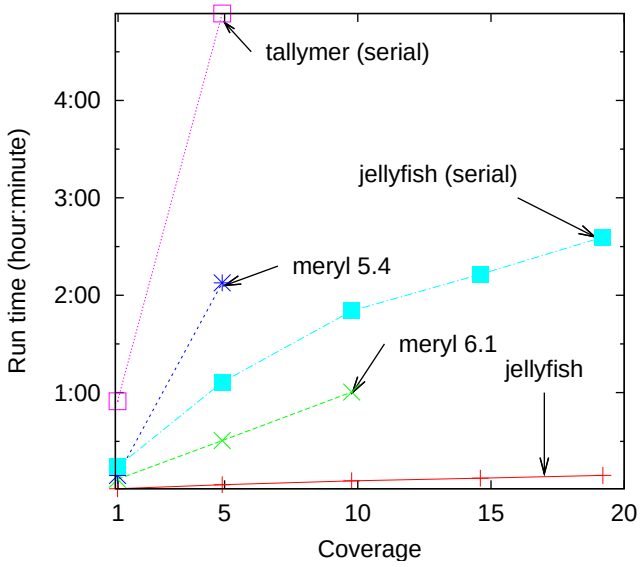


Figure 2.3: Computation time versus sequencing coverage on reads generated during the Turkey genome project. Except as noted all programs were run using 32 threads.

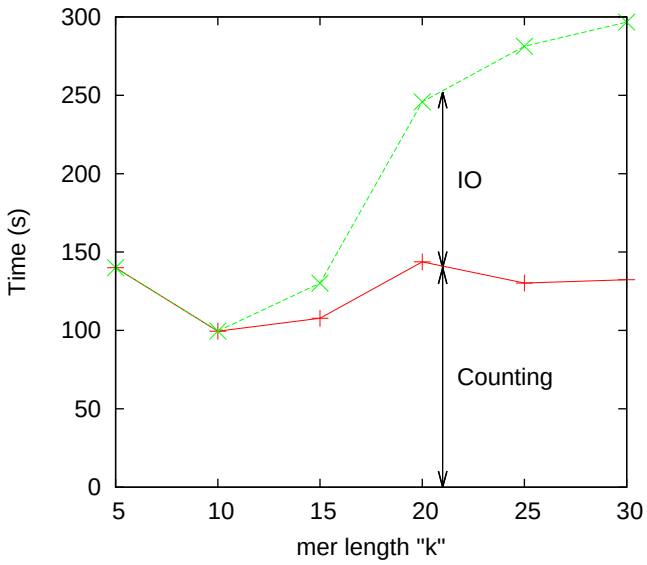


Figure 2.4: Detailed running time of Jellyfish counting k-mers for different values of k on coverage 5x of the Turkey reads.

Jellyfish requires far less memory than the current versions of either Meryl or Tallymer (Figure 2.2). The memory usage of Jellyfish is approximately the same for coverage 5x and 10x (or for 15x and 20x) because the size of the hash table is constrained to be a power of two and the same table size of 2^{32} (or 2^{33}) entries is used. Tallymer does not support multi-threaded operation. When run in serial mode, the memory usage for Jellyfish is almost identical with the usage in multi-threaded mode. Meryl version 5.4 contained a software error that prevented it from correctly parsing large input files and was run only up to 5x coverage. At coverage 5x, Jellyfish used only slightly less memory than Meryl 5.4. Meryl version 6.1 ran out of memory for coverage 15x and 20x, and it appears that trade-offs between speed and memory usage were changed between versions 5.4 and 6.1.

Jellyfish is also much faster than Meryl and Tallymer (Figure 2.3). At coverage 5x, representing approximately 5 Gb of sequence, Jellyfish counts 22-mers in under 4 minutes, while the other approaches take between 30 minutes and 4.9 hours. Jellyfish

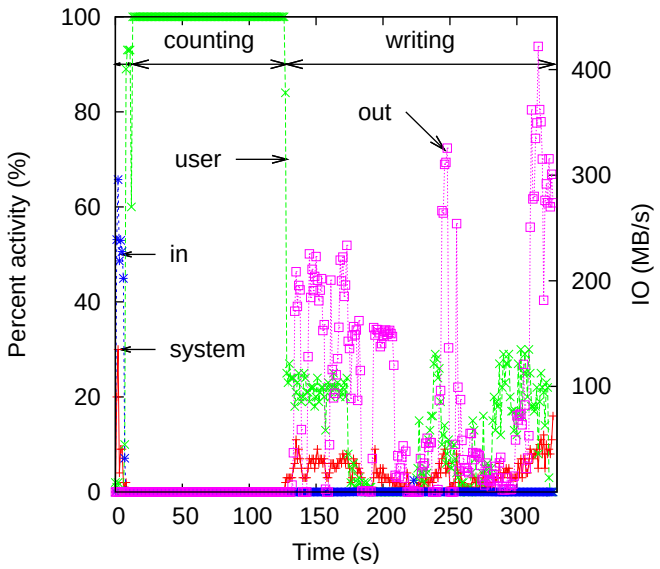


Figure 2.5: A trace of Jellyfish's CPU usage and IO throughput on counting 22-mers on coverage 5x of the Turkey reads with 32 threads. CPU usage is split into "system" (corresponding to all system calls for memory allocation, read/write from disk, etc.) and "user" (the program). The "percent activity" is a global activity measure over all 32 cores. The IO throughput is split into "in" for input and "out" for output.

is also able to count 22-mers at coverages $>$ 10x where the other programs fail or take over 5 hours.

Figure 2.4 shows the impact of varying the mer length k on computation time, showing the contribution of both IO and the actual counting. As k increases from 5 to 30, the counting time stays approximately the same, while the IO time grows significantly because of the larger number of distinct k-mers that must be output. For small values of k (here < 15) Jellyfish uses direct indexing (where there is one entry in the table for each of the 4^k possible k-mers). For very low k (here $k = 5$), the effect of multiple threads trying to increment the same memory location is compounded and explains the longer counting time for $k = 5$ than for $k = 10$.

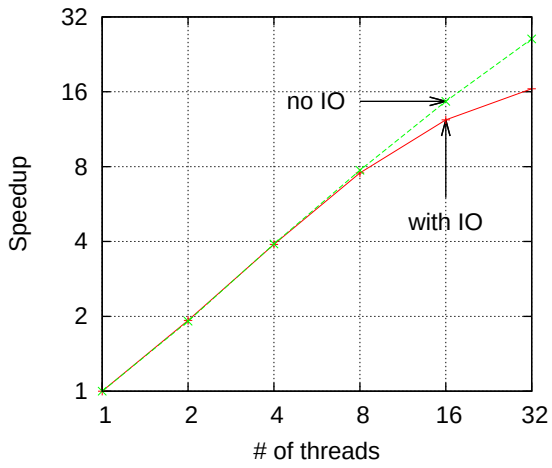


Figure 2.6: Speedup versus number of threads on coverage 5x of the Turkey reads. On this log-log scale plot, a perfectly linear speedup would correspond to a diagonal line. The "no IO" curves includes only the initialization and counting phase times. The curve marked "with IO" counts the total runtime.

2.4.2 Jellyfish's Architecture Allows for a High Degree of Parallelism

The CPU and IO usage of Jellyfish on 32 threads while counting 22-mers on coverage 5x of the turkey reads is shown in Figure 2.5. There are three distinct phases in the trace. First, initialization, when memory is zeroed out and the input file is aggressively pre-loaded in cache by the operating system. This lasts 14 seconds. The second phase is active counting which uses 100% of the 32 CPUs available on the machine. Thanks to the lock-free design, the threads almost never wait on each other. The CAS operation is a CPU operation, not a system call that would require an expensive context switch. Therefore, the counting phase is fast and the operating system uses no computational resources during this phase. The final phase is writing, where the results are sorted and written to disk. In this phase, the operations are bounded by IO bandwidth. The default output format, used when creating this trace, is designed to be easy to parse rather than compact. A more compact file format would lead to faster execution time in the third phase.

Figure 2.6 shows the speedup obtained when increasing the number of threads used. Again, the k-mers are counted on coverage 5x of the Turkey reads. On the upper curve (labeled "no IO") takes into account only the initialization and counting phases, i.e. only the hash table operations. The lower curve (labeled "with IO") also includes the writing phase where the result is sorted and written to disk. The hash operations have an almost linear speed-up up to 32 threads (the number of cores on the test machine). When including the writing to disk, the speed-up is linear up to 4 threads and nearly linear up through 8 threads. Then, the sorting is done fast enough that the IO bandwidth is the bottleneck (as seen in Figure 2.5) and the speed-up levels off.

2.4.3 Timing Results for Other Genomes

Jellyfish is able to process genomes or the reads from recent sequencing projects in only a few minutes. Table 2.1 contains the timing and memory usage of Jellyfish on several data sets, computed with 32 threads. The number of different k-mers in each data set is reported as "distinct", and the total number of k-mers is also shown.

The timing information for Turkey coverage 20x is included for comparison. Even on the reads of a very repetitive genome, such as *Z. mays*, Jellyfish takes less than 20 minutes, while computing the suffix array with Tallymer would take about 24 hours.

2.5 Conclusion

Increasingly, practical computation on large collections of genomic sequences requires software which can use parallel computer architectures that are commonly available today. The lock-free operations used in Jellyfish permit the design of truly concurrent

Table 2.1: Performance of Jellyfish on the chromosomes of the human genome and the reads from several sequencing projects.

Organism	Time	RAM	# of 22-mers ($\times 10^6$)	
	m:s	Gb	distinct	total
H. sapiens	3:33	11.8	2 351	2 861
Z. mays	18:14	55.9	7 161	26 653
M. gallopavo	9:01	24.7	5 503	19 446
D. ananassae	2:19	7.35	1 197	2 936
C. burnetii	0:02	1.25	10.2	34.2

data structures that are fast in serial mode and scale almost linearly with the number of processors used.

Jellyfish can tackle k-mer counting on the large data sets available today. As short-read sequencing projects become more common and achieve larger and larger coverage, efficient k-mer counting will become increasingly important. The hash table at the heart of Jellyfish is a versatile and widely used data structure. Proper optimizations make Jellyfish's hash table competitive in both time and space even compared with other data structures specifically design for string processing, such as suffix arrays, as implemented in competing k-mer counting packages.

Chapter 3

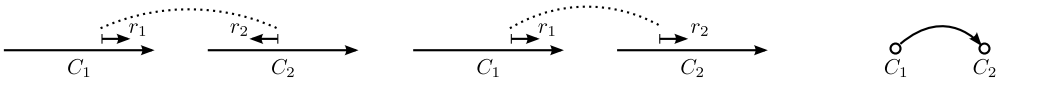
Chromosome Builder

3.1 Introduction

3.1.1 The goals of scaffolding

The scaffolding stage of a genome assembly solves two problems: finding relative orientations of the contigs and finding relative positions of the contigs. In addition, if a marker map is present, the chromosome builder considers also the absolute orientation and absolute position of contigs on the chromosomes. We shall describe these four problems in more detail.

Orientation of contigs. The sequence of each contig represents either strand of the DNA molecule. Two contigs may represent different strands. To create larger structures made of contigs, i.e. scaffolds (a.k.a. supercontigs), it is necessary to *orient* the contigs with respect to each other, so that they all represent the same strand. It is a relative orientation as it is not known which strand the contig represents, only that they all represent the same one. Because of the presence of erroneous mate pairs in the data, solving the orientation problem is an NP hard problem [41]. In practice, assembler programs use various heuristics to obtain a reasonable solution [79]. The Chromosome Builder assumes that the relative orientation problem has been solved. This means that the software silently ignores mate pairs which are not consistent with the given orientation. A mate pair (r_1, r_2) between contigs C_1 and C_2 is consistent



(a) Consistent orientation (b) Inconsistent orientation (c) Directed edge

Figure 3.1: In 3.1a, the mate pair (r_1, r_2) is consistent with the orientation of the contigs C_1 and C_2, as r_1 and C_1 have the same orientation while r_2 and C_2 have opposite orientation. In 3.1b is shown the case where the mate pair (r_1, r_2) does not have a consistent orientation with C_1 and C_2. The mate pair in 3.1a contributes the oriented edge from C_1 to C_2 as shown in 3.1c to the mate pair graph while the mate pair in 3.1b is silently ignored.

with the orientation of C_1 and C_2 if one read has the same orientation as its contigs and the other has the opposite orientation, as shown in Figure 3.1.

Scaffolding. Once the contigs are oriented with respect to each other, scaffolding consists in finding the order in which the contigs appear in the DNA and an estimation of the distances, commonly referred as "gaps", between the contigs in this ordering. If two contigs overlap each other, the gap is negative. This ordering problem is easily expressed as a graph problem. Consider a directed graph where each contig is a vertex and for each mate pair (r_1, r_2) between two contigs, there is a directed link from the contig in the same orientation with its reads to the contig in opposite orientation with its read, as in Figure 3.1. In the presence of errors, finding the most parsimonious ordering is equivalent to solving the minimum feedback arc set, which is NP hard.

Positioning contigs on chromosomes. The markers are short fragments of DNA ($\sim$ 100 bp) whose sequence and position in the genome are known, albeit often with a large standard deviation ($\sim$ 100 kbp). If a marker map is available, it is possible to determine the absolute orientation and absolute position of the contigs, i.e. not only determining the orientation and position of the contigs relative to each other, but also the actual position on the chromosome [50]. The orientation of these markers on

the chromosome is usually not known. Hence, to orient a scaffold, it must have at least two markers separated by several standard deviations.

The traditional approach is to create scaffolds ignoring markers, and then use marker information to place scaffolds on chromosomes. Some breakage of scaffolds may be required when there is no way to place a scaffold that is consistent with the markers. Our view is that markers provide information useful not only to positioning, but also to scaffolding (see section 1.4). Moreover, by merging the scaffolding and placement operations, all data available is used for both operations.

3.1.2 Solving the ideal case

Chromosome Builder is a computer program designed to determine the most likely placement of contigs on chromosomes. The input to the program is a collection of contigs, mate pairs and markers. The contigs are oriented relative to each other, the mate pairs each connect two contigs, and the markers are sequences in contigs such that the approximate location of the sequence in a chromosome is known. In reality, a significant portion of this data is simply wrong. For example, what is listed as a contig may be a chimeric merger of two unrelated contigs. The primary task is to ferret out this erroneous data. However, we begin by describing how we would solve the problem in the ideal case where none of the data is seriously wrong, that is none of the data needs to be excluded to obtain a reasonable solution. The general case requires a careful selection of the data to be used (see Section 3.2.1). More specifically the ideal case is specified as follows.

1. Let C be a collection of n contigs that all are subsequences from the same chromosome. The orientations of each contig on the chromosome is known. Let $x_1, \ldots, x_n$ denote the unknown positions of the first base of each contig.

2. Let $\mathcal{P}_0$ be a collection of mate pairs that connect contigs. Each mate pair p connects contig $l(p)$ to contig $r(p)$. The expected length $\delta_p^\star$ of the mate pair and its standard deviation σ_p are known. The length $\delta_p^\star$ and the positions of the reads of the mate pair in the contigs corresponds to a distance δ_p between the beginnings of the contigs, that is, $x_{r(p)} - x_{l(p)}$ (see Figure 3.2c).

3. Let $\mathcal{M}_0$ be a collection of markers. For $m \in \mathcal{M}$, write $c(m)$ for the contig to which the sequence of m aligns. Write $\rho_m^\star$ for the expected position of the marker and σ_m for its standard deviation. The position of the marker in the contig and $\rho_m^\star$ corresponds to a position ρ_m of the beginning of the contig, that is $x_{c(m)}$ (see Figure 3.2d).

4. The estimates of the positions ρ_m and of the distances between the contigs δ_p are independent random variables.

We estimate the positions of contigs by minimizing the following function of the positions.

$$f(x_1, \ldots, x_n; \mathcal{P}_0, \mathcal{M}_0) = \sum_{p \in \mathcal{P}_0} \frac{\left(x_{r(p)} - x_{l(p)} - \delta_p\right)^2}{\sigma_p} + \sum_{m \in \mathcal{M}_0} \frac{\left(x_{c(m)} - \rho_m\right)^2}{\sigma_m}.$$

That is, the gradient of f with respect to $x = (x_1, \ldots, x_n)$ is set equal to 0 and the resulting linear system is solved.

We justify the weightings (the $1/\sigma$ factors) as follows. The linear system obtained by setting the gradient of f equal to 0 has an intuitive meaning that can be explained most easily by assuming there are no markers. The discussion with the markers is analogous. The ith equation, $\partial f / \partial x_i = 0$, is the sum of k_i terms where k_i is the number of mate pairs with one read in contig i. The term for mate pair p is either

$$\frac{\delta_p - (x_{r(p)} - x_i)}{\sigma_p} \quad \text{or} \quad \frac{(x_i - x_{l(p)}) - \delta_p}{\sigma_p} \tag{3.1}$$

and represents the deviation $(x_{r(p)} - x_i)$ (in the first case) from the expected value δ_p — measured as a multiple of the standard deviation. We can also refer to these quantities as the *strain* s_p imposed on a mate pair due to the values of the positions of the two contigs the mate pair connects. The ith equation says that the sum of the strains on the mate pairs connecting contig i is 0. The strain of mate pair p is inversely proportional to the standard deviation of its library. Suppose for example one of the mate pairs, p_0, would have strain 0 if $x_i = a$ while the remaining $k_i - 1$ mate pairs would have strain 0 if $x_i = b$. Then, the solution would place a strain on mate pair p_0 equal to the sum of the strain on the other mate pairs. (The strain on p_0 would have the opposite sign of the strain on all other mate pairs.)

We also experimented with the weighting being $1/\sigma_p^2$. The result was that many inserts with large standard deviation had larger strains than short inserts. The signature of an "extremely" stretched mate pair is to have a large strain, e.g. $|s_p| > 3$, that is, the mate pair is stretched by more than 3 standard deviation. Using the $1/\sigma_p^2$ weighting results in many extremely stretched inserts with large σ_p.

3.2 Methods

3.2.1 Problem statement

The Chromosome Builder computes the position of the first base of each contig. Recall the data available is the list and length of contigs; mate-pair library information (mean length and standard deviation); the position and orientation of the reads into the contigs; optionally, the position of markers in the chromosome and in the contigs.

The data for the mate-pairs and markers is not known precisely. We try to identify and eliminate erroneous information, such as:

- bad mate-pairs result from either having two unrelated reads reported as mated, or a read can be positioned in the wrong instance of a repeat region;

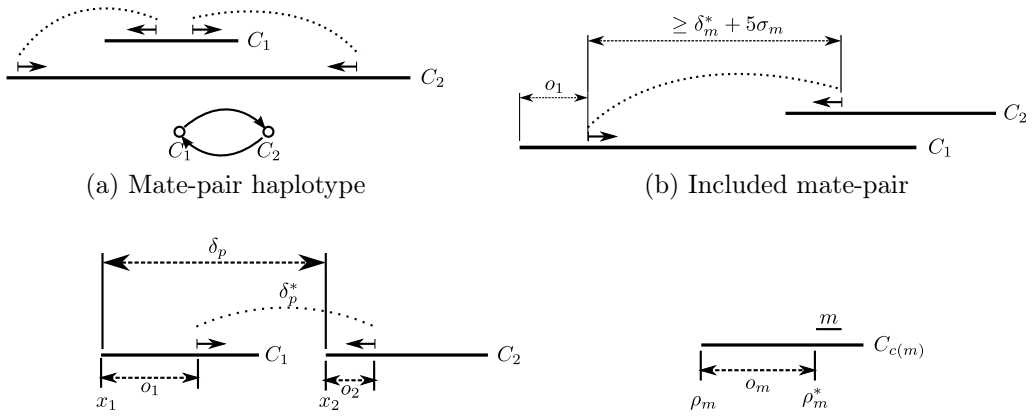
- bad contigs result from misassemblies, where two unrelated fragment of sequence are glued together in a contiguous piece;
- bad markers where the reported marker positions is simply wrong;
- haplotype variants of contigs present in the data.

The Chromosome Builder is centered around iteratively selecting data to be used by our iterated least squares solver that will be described below (section 3.2.3). We describe here the pre-processing and post-processing operations designed to make the procedure robust to the errors above. The pre-processing operations attempt to cleanup the input data and remove the data which is most likely erroneous. The post-processing operations checks for errors in the layout generated by the least squares solver and further removes the contigs or mate-pairs which are the likely cause of the errors. The entire procedure (pre-processing, least squares solver, post-processing) is then repeated until the contig layout converges. This usually occurs at the first iteration. Every piece of information which was removed during one iteration is not used in the subsequent iterations.

3.2.2 Pre-processing: data cleanup

The following cleanup operations are performed: removal of mate-pair haplotypes, of contained mate-pairs, and of duplicated mate-pairs; verification of mate-pair consistency.

Mate-pair haplotypes. A contig C_1 is a mate-pair haplotype of a contig C_2 if $|C_1| < |C_2|$ and there exists anti-parallel edges between C_1 and C_2 in the mate-pair graph, as seen in Figure 3.2a. In this case, the contig C_1 and all its mate-pairs are removed from the data set.



(a) Mate-pair haplotype

(b) Included mate-pair

(c) Implied distance

(d) Implied position

Figure 3.2: In 3.2a, contig C_1 has mate pairs linking to contig C_2 on the $3'$ and $5'$ ends. In the mate-pair graph, there are two anti-parallel edges between C_1 and C_2. This would typically occur when C_1 is a haplotype variant of C_2. In 3.2b, the mate pair between C_1 and C_2 is included because even if stretched by more than 5 standard deviations, the contigs it links overlap. 3.2c shows the implied distance between the first bases of contigs C_1 and C_2 implied by a mate pair with mean δ_m^* is $\delta_m = x_1 - x_2 = o_1 + \delta_m^* - o_2$. 3.2d shows the starting position of a contig $\rho_m = \rho_m^* - o_m$ implied by a marker reported at position ρ_m^* and aligning with offset o_m.

Validating mate pairs. A chimeric mate pair consists of two independent reads that are accidentally declared mates. Inclusion of such a mate pair is in effect a statement that the two contigs that contain the reads and are far apart in the genome should placed next to each other. If the two contigs are in different regions that have no legitimate mate pairs connecting them, then the chimeric mate pair's incorrect information is not disputed. There is no contradictory information. As a result the two regions can be drawn together in the contig's layout, intermingling contigs that should be far apart. Our approach to deal with this problem is to remove from consideration every mate pair which is not validated by other mate-pair data as described below.

A mate-pair is *validated* if there exists another mate pair, between the same pair of contigs C_1 and C_2, which imply the same distance between C_1 and C_2 within 3σ. As shown in Figure 3.2c, the distance between C_1 and C_2 by a mate-pair is $\delta_m = o_1 + \delta_m^* - o_2$. Two mate-pairs between contigs C_1 and C_2 are consistent if

their implied distance δ_m and $\delta_{m'}$ satisfy $|\delta_m - \delta_{m'}| \leq 3\sigma$. A mate-pair which is not consistent is removed from the data set.

Replicated mate pair removal. Both 454 [57] and Illumina [3] sequencing are prone to chemically generate multiple copies of a mate pair when there should be only one. If multiple copies of a mate pair are kept in the data set, they will appear to validate each other.

We say two reads in the same contig are *coincident* if their starting positions are the same, within a few bases. We say two mate pairs m_1 and m_2 are *replicates* of each other if each of their reads is coincident with one from the other mate pair. Only one instance of the replicated mate pairs is kept in the data set.

Reads in a repeat region. Some assemblers flag the reads which are placed in a repeat region. Because of the possibility for such a read to be placed in the wrong instance of the repeat, Chromosome Builder ignores any mate pair containing at least one read placed in a repeat.

Contained mate pairs. Certain situations involving validated mate pairs suggest that the contigs may be misassembled or may be from different haplotypes representing the same region. Such possibilities may require further investigations which are outside the scope of this work. For preprocessing, we assume the contigs are valid and different contigs represent non-overlapping regions of the genome. From the position of a read in a contig and the size of the contig, we can tell whether its mate should lie in the same contig, judging from the mean and standard deviation of the mate-pair library.

A mate-pair with mean δ_m^* and standard deviation σ_m between two contigs C_1 and C_2 is *contained* if the mate-pair fits entirely in contig C_1 or C_2. Using the notation of Figure 3.2b, the mate-pair is contained in contig C_1 if the following holds:

$o_1 + \delta_m^* + 5\sigma_m \leq |C_1|$. If validated mate pairs are also contained, the software flags the contigs as being potentially misassembled.

3.2.3 Iterative solver

Even after the data cleaning described above, the layout created by solving this least squares problem is likely to be incorrect. The iterative solver repeatedly solves a least square optimization, removing at each iteration the most "unhappy" elements (mate pair or marker), until all the remaining elements are "happy".

More precisely, the sets $\mathcal{P}_0$ and $\mathcal{M}_0$ are the sets of mate pairs and markers after the cleaning operations. Then, each iteration consists in setting

$$(x_1^{(i)}, \ldots, x_n^{(i)}) = \underset{(x_1,\ldots,x_n)}{\arg\min}\, f(x_1, \ldots, x_n; \mathcal{P}_i, \mathcal{M}_i).$$

Then, the sets $\mathcal{P}_{i+1}$ and $\mathcal{M}_{i+1}$ are obtained respectively from $\mathcal{P}_i$ and $\mathcal{M}_i$ by removing 0.5% of the most strained elements, where the strain is defined in eq 3.1 computed for the current solution $(x_1^{(i)}, \ldots, x_n^{(i)})$.

Also, we ensure that for any contig C, at most one mate pair or marker involving C is removed. This last condition allows for the detection of misassembled contigs in the post-processing stage, as explained in Section 3.2.4. A mate pair or marker which was removed from $\mathcal{P}_i$ or $\mathcal{M}_i$ at a previous iteration reintegrates its set if its strain becomes less than 3 in absolute value.

The iteration stops when there is no more mate pair or marker with a strain greater than 3 in absolute value.

3.2.4 Post-processing: error detection

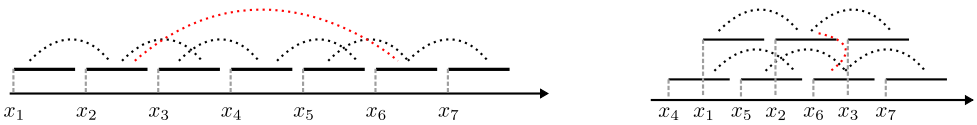
Three tests are performed on the layout created by the iterative solver: tests for misoriented contigs or collection of contigs, misassembled contigs, and what we call "folded scaffolds".

Misoriented contigs. While each contig is provided with an orientation, that orientation can be wrong. It is easier to detect misorientation for large collection of contigs. This test checks that the orientation of the connected components is in agreement with the marker map. The contig layout generated by the least squares solver gives positions for the markers. The slope of the linear regression between the marker positions in the map and in the computed layout should be close to 1. A negative slope indicates that the contigs in a connected component have an incorrect orientation.

Misassembled contigs. A contig is *misassembled* if its sequence does not exist in the genome. This test intends to detect contigs made up of two piece of sequence which do not come from the same region of the genome. The mate-pairs of these two pieces cannot be satisfy simultaneously. Hence, the misassembled contig test detects contigs with a cluster of satisfied mate-pairs and a cluster of unsatisfied mate-pairs. These contigs and incident mate pairs are removed from the data set.

Folded scaffolds. An erroneous mate-pair can force two distant contigs to be placed close to one another. This has the effect of "folding" the scaffold onto itself, as shown on Figure 3.3. The connected component is then made of two parts which are individually correctly scaffolded, but the two parts are placed on top of each other instead of being placed one after the other.

This test detects this situation by looking for discrepancies in the distances between every pair of contigs in the mate-pair graph and in the layout. The distance



(a) Correct layout (b) Folded layout

Figure 3.3: In 3.3a the correct layout of 7 contigs. The mate-pairs are represented by doted lines. One erroneous mate-pair, shown in red, links two distant contigs. In 3.3b, the folded layout when the red mate-pair is erroneously trusted by the least squares procedure.

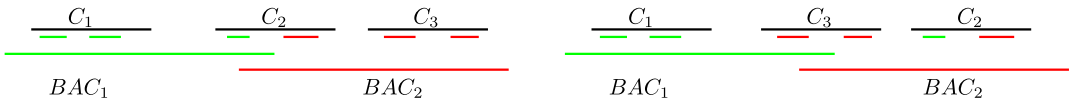
in the mate-pair graph is the length of the shortest path between two contigs. The distance in the layout is the distance of the first bases of the two contigs. In a correct layout, the two distances have roughly a linear relationship.

In the folded layout, every path between contigs in different parts must pass through one of the erroneous mate-pairs. Hence, this test first computes the number of times a mate-pair is traversed by the all the shortest paths between contigs adjacent in the layout. Then it finds the mate-pairs which are overly represented. These offending mate-pairs are removed from the data set.

3.3 Results

We applied an early version of the Chromosome Builder to position the contigs on the chromosomes in the UMD *Bos taurus* assembly version 3.1 (see Chapter 1). This assembly has recently been annotated by NCBI and is currently the official reference genome. In this section we analyze the positions of the BAC reads on the chromosomes in the 3.1 assembly and show that they are more consistent than in Btau 4.1 by Baylor.

By construction, all the reads from the same BAC come from the same neighborhood of length less than 400 kb. In our assembly of the *Bos taurus* genome, we used this *BAC read co-locality* only to place additional sequence (see Section 1.2.3), representing 3% of the total sequence. Although ATLAS assembles each BAC separately and thus implicitly uses this information, it can make mistakes when stitching



(a) Correct contig ordering. (b) Inversion of C_3 and $C2$.

Figure 3.4: In 3.4a, contigs C_2 and C_3 are in a correct order and the BAC name tags follow a logical succession. In 3.4b, contigs C_2 and C_3 are in reversed order and display the interleaving pattern of BAC name tags.

the BAC assemblies together. Here, we compare the consistency of the BAC read co-locality information in the two assemblies to evaluate their quality.

We assign "tags" to each contig. A tag corresponds to a BAC, and a contig containing three or more reads from a BAC is tagged by that BAC name. A contig may have zero, one, or more tags. The positions of the tagged contigs on the chromosome should be consistent: two contigs sharing a tag should be placed no more than a BAC length apart, and, by the same token, the contigs sharing tags should cluster together.

In this study, we consider only contigs that are longer than 5 kb. Given the coverage by BAC reads, the expected number of BAC reads in such contigs is over 15. Hence, in a correct ordering, a contig C flanked by two contigs tagged by BAC B should also be tagged with B with high probability. See Figure 3.4a.

We search for tag sequences which are unlikely. More specifically, since the number of possible erroneous patterns is large, we chose the sequence of tags $B_1 - B_2 - B_1 - B_2$ to compare the two assemblies. This pattern is indicative of an error with high probability. We find 478 such patterns in the BCM assembly, versus 160 in the UMD assembly, a 66% improvement.

Part II

Graph problems

In this part, we explore some problems in graph theory. We first describe the seeded complex detection problem which lead to the study of exactly k-edge connected graphs in Chapter 4. Although biological problems motivated the study of these graphs, the results obtained are of theoretical nature and only remotely connected to the initial inquiry, but interesting in their own right.

In Chapter 5, we consider the problem of reconstructing, under the assumption of parsimony, the biological network of some ancient species which is a common ancestor to two or more extant species. We present a framework to encode the evolution of the biological network and an algorithm which finds efficiently, in practice, the optimal ancestral network.

The seeded complex problem. A biological complex is a group of proteins that bind together to perform some biological function. Finding which proteins form complexes is of great biological interest. Experiments to discover the structure of such complexes, like crystallography, are time consuming and expensive. This motivates the development of computational methods to detect complexes from the results of high-throughput experiments, such as Protein-Protein-Interaction (PPI) networks generated by Yeast Two Hybrid (Y2H) experiments [100, 37]. A PPI network is represented by a weighted graph $G = (V, E, w)$, where V is the set of all proteins, E is the set of physical binary interactions which have been experimentally observed and $w : E \to [0, 1]$ is the probability for this edge to be a true one.

Databases, such as MIPS [73], which catalog known complexes, are incomplete. Only a small subset of all complexes existing in an organism are present in the database and the complexes cataloged are not complete, i.e. some proteins could be missing. In the seeded complex problem, one is interested in augmenting the description of existing complexes. Formally, given a PPI networks represented as a weighted graph $G = (V, E, w)$ and a set of proteins $C \subset V$ which are believed to be part of

one complex, find a superset set of proteins $C' \supset C$ which are likely also part of the same complex.

We considered trying to answer the following question as an approach to solving the problem above. Given the set of proteins C in our graph G, what is the most parsimonious way to connect all these vertices in G? Computationally, this is equivalent to the minimum Steiner tree problem [19, 84]. More precisely, we considered the following metric. If $C \subset V$ is a set of vertices, then the Steiner weight $\mathrm{sw}(C)$ is the weight of a minimum Steiner spanning C. Then, for any vertex $p \in V$, the Steiner distance of p to C is

$$\mathrm{sd}(p, C) = \frac{\mathrm{sw}\left(C \cup \{p\}\right)}{\mathrm{sw}(C)} - 1\,.$$

If, and only if, p is on one minimum Steiner tree of C, then $\mathrm{sd}(p, C) = 0$. Otherwise, this measures how much a Steiner tree of C has to be modified to incorporate the protein p.

The algorithm to solve the seeded complex problem consists by computing the Steiner distance $\mathrm{sd}(p, C)$ for every protein in the graph and returning the set of proteins with a Steiner distance to C less than a threshold.

Computing minimum Steiner trees. The minimum Steiner tree problem is NP-hard [40]. Hence, solving the minimum Steiner tree problem is computationally expensive. The Steiner distance described above requires solving many instances of the minimum Steiner tree problem. As a way to decrease the size of the minimum Steiner tree problem, we considered the following strategy. Let C be the set of required vertices. Find the biconnected component of the graph G. We now consider two sets of minimum Steiner tree problems.

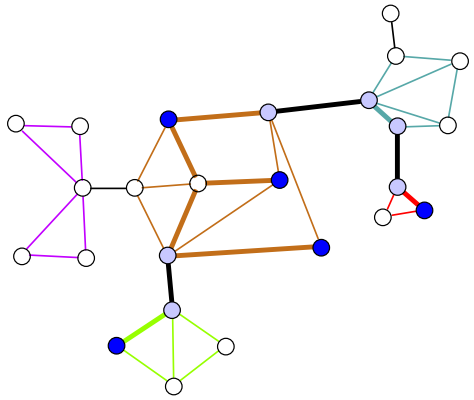


Figure 3.5: The black edges are the bridges, separating the biconnected component. Each biconnected component has a different color for its edges (purple, green, orange, blue and red). The required vertices are colored dark blue and the light blue vertices are the articulation vertices added as required in each biconnected component. The solution is shown with bold edges.

First, collapse every biconnected component into a super-vertex. The resulting graph is a tree. Mark every super-vertex which contains a required vertex as required. Solving the minimum Steiner tree problem in a tree is a trivial problem.

Second, in each biconnected component V_i, consider a minimum Steiner tree problem where the required vertices are $C \cap V_i$ and every articulation point vertex of V_i which is on the path to another biconnected component which contains required vertices. Each minimum Steiner tree problem instance is smaller than the original problem (graph is smaller and fewer required vertices).

It is then trivial to patch together a solution to the minimum Steiner tree problem in G from the solutions in the tree and in each biconnected component V_i, as shown in Figure 3.5.

Exactly connected graphs. A natural question is whether this scheme can be extended further. That is, take a biconnected component and collapse into a super-vertex every vertex which are connected by at least 3 edge-disjoint paths. Can the solutions to the minimum Steiner problems in the graph of super-vertices and in each super-vertex

be patched together in a solution to the problem in G? Although we do not have an answer to this question, we studied the following relevant class of graph.

A graph G' obtained from k-edge-connected graph G by collapsing into super-vertices the vertices which are connected by at least $k+1$ edge-disjoint paths has the following property: between every two vertices of G' there are exactly k edge-disjoint paths. Chapter 4 characterizes the exactly k-edge-connected graphs for $k = 1, 2, 3$, and some variants: exactly connected and planar, and minimally connected graphs.

Parsimonious ancestral network reconstruction. Biological networks, such as PPI or regulatory networks, of modern species can be observed. Phylogenetic trees relate extant and extinct species based on common traits, genetic information and fossil records. On the other hand, the actual biological network of an extinct species, or the evolution of the biological network of an ancient species to a modern one, is inaccessible by experimental means. Nevertheless, one would like to understand how modern features in extant species arose.

We consider the problem of reconstructing, in the most parsimonious way, the history of the creation and deletion of edges in a biological network as it evolves from an ancient species to a modern one. We assume that we have complete homology information between the proteins in the extant network, i.e. we have a binary tree completely describing how every vertex in the extant network appeared from earlier vertices and gene duplication. Moreover, we assume that right after duplication, the two genes are identical and have the same set of edges in the network as their ancestor. After duplication, the genes diverge by creation of new edges or deletion of existing edges. In this context, a most parsimonious history is one with the fewest number of creation and deletion events.

In Chapter 5, we show a method to reconstruct such a parsimonious histories. Although not theoretically guaranteed, this method returns in practice the optimal

or near optimal solution in polynomial time. Finally, we present variations on the method for different type of graphs — directed, with self-loops — and a way to reconstruct the biological network of a common ancestor to two related species.

Chapter 4

A Synthesis for Exactly 3-edge-connected Graphs

4.1 Introduction

A graph that is k-edge-connected and k-regular has the property that, between any pair of distinct vertices u and v, there are k and only k edge disjoint $u-v$ paths. The class of k-edge-connected, k-regular graphs has been heavily studied, in particular the case $k = 3$. On the other hand, the class of graphs for which there are exactly k edge-disjoint paths between every pair of vertices is much larger than the k-edge-connected, k-regular graphs. For example, such a graph can have arbitrarily many vertices of arbitrarily large degree. We call a graph with exactly k edge-disjoint paths between every pair of vertices *exactly k-edge-connected*. In this chapter, we will completely characterize the exactly k-edge-connected graphs for $k = 1, 2, 3$ and study some of their properties.

If C is an induced cycle in a graph G, then the graph G' obtained from G by collapsing C into a single vertex has a lower or equal vertex connectivity and greater or equal edge connectivity than G. Various conditions to guarantee the existence of an induced cycle C in a biconnected graph G such that G' is also biconnected have been given in [98, 97, 22, 29]. Such cycles are called *non-separating induced cycle*s. To prove the correctness of our characterization of exactly 3-edge-connected graphs, we will prove (Theorem 4.3) the existence of induced cycles which, when collapsed, simultaneously preserve 2-vertex-connectivity and 3-edge-connectivity.

Exactly connected graphs arise in several contexts. For example, they are obtained if all pairs of vertices of higher local edge-connectivity are merged. Suppose G is a k-edge-connected graph, but not necessarily exactly k-edge-connected. If the sets of vertices that are mutually connected by $> k$ edge-disjoint paths are collapsed into supernodes, then the resulting multigraph G' is either a graph of a single vertex or is exactly k-edge-connected. This follows because G' is at least k-edge-connected as G was, and no two supernodes can be connected by $> k$ edge-disjoint paths, otherwise they would have been merged. That such supernodes are partition the nodes of G [60] follows directly from Menger's theorem [17].

All exactly k-edge-connected graphs are both edge-minimal and edge-maximal in the sense that removing or adding an edge will destroy exact k-edge connectivity. k-connected edge-minimal graphs, also called minimally connected graphs, have received a lot of attention [4, 32, 78, 31, 55]. On the other hand, not all edge-minimal k-edge-connected graphs are exactly k-edge-connected. Hence, the class of exactly k-edge-connected graphs is a strict subset of the edge-minimal k-edge-connected graphs. The Harary graphs [30] (graphs satisfying a connectivity requirement k with a minimum number of edges) are exactly k-edge-connected, but there is only one Harary graph for each pair (n, k), where n is the order (number of vertices) of the graph and k the connectivity requirement. We shall generalize this concept in Section 4.4.2.

A synthesis of a class of graphs is a set of elementary operations by which all graph of the class can be generated. A synthesis provides a complete characterization of its elements and provides a natural way to reason by induction about the class of graphs. Syntheses for many classes of graphs already exist [2, 10, 35, 14, 44, 39, 99, 93, 30, 18]. Development of syntheses for additional classes of graphs is an active area of current interest.

The exactly 1-edge-connected graphs are simply the trees. The exactly 2-edge-connected graphs are "trees of cycles." More formally, if G is exactly 2-edge-connected,

it is the union of simple cycles such that the block-cutvertex graph $H = (V, E)$ is a tree, where V contains a vertex u_c for each cycle c and a vertex u_v for each vertex $v \in G$ shared by more that one cycle, and where $E = \{\{u_c, u_v\} : v \in c\}$. Every 2-edge-connected graph can be generated from a Whitney-Robbins synthesis [30]. The exactly 2-edge-connected graphs can be generated via a "Robbins synthesis":

Proposition 4.1. *An exactly 2-edge-connected graph G is obtained from a Robbins synthesis. In other words, there exists a sequence of graph $G_i, 0 \leq i \leq l$ such that G_0 is a simple cycle, $G_l = G$ and G_{i+1} is obtained from G_i by a cycle addition.*

Proof. G is 2-edge-connected, so it is the result of a Whitney-Robbins synthesis. Let G_i $(0 \leq i \leq l)$ such that G_0 is a simple cycle and G_{i+1} is obtained from G_i by a path or a cycle addition. Note that G_0 is exactly 2-edge-connected. Suppose that G_i is exactly 2-edge-connected and that G_{i+1} is obtained by a path addition, between u and v. There are 2 edge-disjoint paths between u and v in G_i. The newly added path is obviously edge-disjoint from the paths in G_i. So there are 3 edge-disjoint paths in G_{i+1} between u and v. Given that no edge or vertex is ever removed, there are also 3 edge-disjoint paths in G between u and v, which is a contradiction. On the other hand, cycle addition does not create any new edge-disjoint paths between existing vertices. So G is obtained from G_0 by a sequence of cycle additions. □

The situation is more complicated for $k = 3$. Existing syntheses cannot be easily modified to generate exactly 3-edge-connected graphs. The synthesis of exactly 3-edge-connected graphs is presented in Section 4.2. The existence of certain type of non-separating cycles in quasi 3-regular graphs, on which the synthesis depends, is presented in Section 4.3. Finally, in Section 4.4, we describe some of the properties of exactly edge-connected graphs.

4.2 A Synthesis for Exactly 3-edge-connected Graphs

We begin by describing several operations that preserve exact 3-edge-connectedness when applied to any exactly 3-edge-connected graph. Eventually, the synthesis will exploit only two of these operations: block gluing and cycle expansion. The other operations will play a role in the proof of the synthesis and are also interesting in their own right. In the following, *biconnected* graphs are graphs which are 2-vertex-connected. A vertex whose removal would disconnect a connected but not biconnected graph is called an *articulation point*. A *block* is a maximal connected subgraph that is biconnected. Throughout, all sets should be considered multisets and all graphs are multigraphs and loopless. The notation $(u, v)^r$ denotes an undirected edge between u and v of multiplicity r (when $r = 1$, it is omitted).

4.2.1 Gluing Operations.

In this section, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two exactly k-edge-connected graphs with $V_1 \cap V_2 = \emptyset$. We will define operations to glue these graphs together into $G = (V, E)$ that is also exactly k-edge-connected. See Figure 4.1 for illustrations of several of the operations. The first collection of these operations exploits the relationship between exact connectivity and the blocks, or biconnected components, of a graph.

Definition 4.1 (Block gluing)**.** $G = (V, E)$ is the graph obtained by identifying one vertex from $G_1 = (V_1, E_1)$ and one vertex from $G_2 = (V_2, E_2)$. Formally, let $u_1 \in V_1$ and $u_2 \in V_2$, let $N(u_1)$ and $N(u_2)$ be their respective neighboring vertices and let $X = \{(u_1, v) : v \in N(u_1)\} \cup \{(u_2, v) : v \in N(u_2)\}$ be the adjacent edges of u_1 and u_2.

Then construct $G = (V, E)$ by setting

$$V = V_1 \cup V_2 \cup \{u\} \setminus \{u_1, u_2\}$$
$$E = E_1 \cup E_2 \cup \{(u, v) : v \in N(u_1) \cup N(u_2)\} \setminus X.$$

This is called *block gluing* because the vertex u becomes an articulation point in G.

Proposition 4.2. *If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are exactly k-edge-connected, then any block gluing G of G_1 and G_2 is exactly k-edge-connected.*

Proof. Any pair of vertices that are in the same block of G are connected by exactly k edge-disjoint paths because they were before the gluing and no additional edge-disjoint path can be created by leaving the block and re-entering it. Let u be the articulation point created by the block gluing, and let $w \in V_1, v \in V_2$ be a pair of vertices in different blocks of G. Vertices w, u are connected by k edge-disjoint paths and u, v are also connected by k edge-disjoint paths. Hence, there are $\geq k$ edge-disjoint paths between w, v. Any such path must pass through u, so there cannot be more than k without creating $> k$ edge-disjoint paths between w and u_1 in G_1, which cannot happen. □

Corollary 4.1. *The subgraph induced by a block of an exactly k-edge-connected graph is exactly k-edge-connected.*

Definition 4.2 (k-bridge addition)**.** $G = (V, E)$ is obtained from G_1 by adding a vertex and k parallel edges from the vertex to an existing vertex in G_1. Formally, $V = V_1 \cup \{u\}$ and $E = E_1 \cup \{(v, u)^k\}$.

Corollary 4.2. *If G is exactly k-edge-connected, then any k-bridge addition preserves exact k-edge-connectivity.*

We also have a vertex gluing that allows two blocks to be merged into one, and the converse operation, vertex splitting, which breaks a graph at a minimum cut.

Definition 4.3 (Vertex gluing)**.** Let u_1 and u_2 be degree-k vertices of G_1 and G_2 respectively. Construct G by removing u_1 and u_2 and pairing together the $2k$ edges from G_1 and G_2. Formally, let v_1^i and v_2^i, $1 \leq i \leq k$, be respectively the neighbors (not necessarily distinct) of u_1 in G_1 and u_2 in G_2. We then construct G by setting

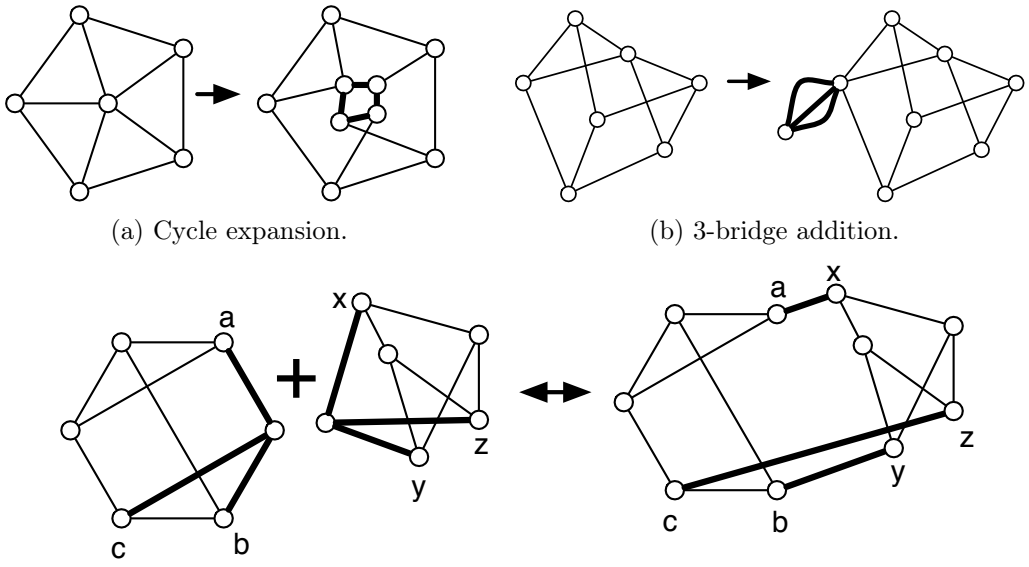
$$V = V_1 \cup V_2 \setminus \{u_1, u_2\}$$
$$E = E_1 \cup E_2 \cup \left\{\left(v_1^i, v_2^i\right) : 1 \leq i \leq k\right\} \setminus \left\{\left(u_1, v_1^i\right), \left(u_2, v_2^i\right) : 1 \leq i \leq k\right\}.$$

The graph G is a *vertex gluing* of G_1 and G_2.

Proposition 4.3. *If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are exactly k-edge-connected, then the vertex gluing $G = (V, E)$ of G_1 and G_2 is exactly k-edge-connected.*

Proof. Let $u_1 \in V_1$ and $u_2 \in V_2$ be the vertices used in the vertex gluing. Consider any pair v, w of vertices in G. If $v \in V_1$ and $w \in V_2$, then there are no more than k edge-disjoint paths between them because the cut $(V_1 \setminus \{u_1\}, V_2 \setminus \{u_2\})$ has cardinality k. On the other hand, k edge-disjoint paths exist between them because there were k edge-disjoint paths from v to u_1 and from u_2 to w which can be combined to create k edge-disjoint paths between v and w.

Suppose, instead, v, w are both in V_1 (the case for V_2 is symmetric). Then there are k edge-disjoint $v - w$ paths, say $P_i, 1 \leq i \leq k$, in G_1. At most $j \leq \lfloor k/2 \rfloor$ of these paths passe through vertex u_1, say $P_i, 1 \leq i \leq j$. Let v_i^1 and w_i^1, $1 \leq i \leq j$ be the vertices that respectively precede and follow u_1 on path P_i. Let v_i^2 and w_i^2 be respectively the neighbors of v_i^1 and w_i^1 in V_2 after vertex gluing. Let x be a vertex of G_2 distinct from u_2. Given that G_2 is k-edge-connected, there exists k edge disjoint $x - u_2$ paths, say $Q_i, 1 \leq i \leq k$, in G_2. Because u_2 is adjacent to every v_i^2 ans w_i^2, the Q_i paths create edge disjoint $x - v_i$ and $x - w_i$ paths, which avoid u_2, for $1 \leq i \leq j$. In turn, let combine paths $x - v_i$ and $x - w_i$ to get j edge disjoint $v_i - w_i$ paths, $1 \leq i \leq j$,



(a) Cycle expansion.

(b) 3-bridge addition.

(c) Vertex gluing ($\rightarrow$) and splitting ($\leftarrow$).

Figure 4.1: Several operations that preserve exact 3-edge-connectivity.

which avoid u_2 (named R_i). By combining the P_i and R_i paths, for $1 \leq i \leq j$, we constructed $v - w$ paths in G.

Conversely, there cannot be more than $\lfloor k/2 \rfloor$ edge disjoint $v - w$ paths in G detouring into G_2 because of the cut $(V_1 \setminus \{u_1\}, V_2 \setminus \{u_2\})$. If there were $> k$ edge disjoint $v - w$ paths in G, there would be $> k$ edge disjoint $v - w$ paths in G_1. So G is exactly k-edge-connected. □

Definition 4.4. An edge cut S of a graph G is called *trivial* if one of the components of $G \setminus S$ is the trivial graph (graph with one vertex and no edges).

Definition 4.5 (Vertex splitting)**.** Let $G = (V, E)$ be an exactly k-edge-connected graph and $S = \langle V_1, V_2 \rangle$ be a non-trivial minimum cut. Construct $G_1 = (V_1 \cup \{x_1\}, E_1)$ and $G_2 = (V_2 \cup \{x_2\}, E_2)$ by adding two new vertices x_1 and x_2 attached respectively to G_1 and G_2 by k new edges to the vertices adjacent to S. Formally, let $S =$

$\{(u_i, v_i) \in V_1 \times V_2 : 1 \leq i \leq k\}$ and

$$\begin{aligned} E_1 &= (V_1 \times V_1 \cap E) \cup \{(x_1, u_i) : 1 \leq i \leq k\} \\ E_2 &= (V_2 \times V_2 \cap E) \cup \{(x_2, v_i) : 1 \leq i \leq k\}. \end{aligned}$$

The pair G_1, G_2 is called a *vertex splitting* of G with respect to S.

Corollary 4.3. *Let G be an exactly k-edge-connected graph, S be a non-trivial minimum cut. G_1 and G_2 obtained by vertex splitting of G with respect to S are exactly k-edge-connected.*

4.2.2 Cycle Contraction and Expansion.

We describe now two additional operations that create and remove cycles within exactly k-edge-connected graphs. The first, cycle expansion, is the main non-trivial operation of the synthesis. A set of vertices C of a graph G is an *induced cycle* if the subgraph induced by C in G is a cycle graph. By convention, in a multigraph G, we view a double edge $(u, v)^2$ as a induced cycle of length 2. On the other hand, if $C = (u_1, \ldots, u_n, u_1)$ is an induced cycle of length $n \geq 3$, every edge in the cycle graph must be a simple edge.

Definition 4.6. A *k-regular* graph is a graph where all vertices have the same degree k. A *quasi k-regular* graph is a graph where at most one vertex has a degree different than k. An induced cycle C is said to be quasi k-regular if the degree in G of every vertex of C is k, except for at most one vertex.

Proposition 4.4 (Cycle expansion)**.** *Let $G = (V, E)$ be an exactly 3-edge-connected and biconnected graph, and let u be any vertex of G. Suppose that u has degree d. Let G' be created from G by replacing vertex u with a cycle C of no more than d vertices where all the vertices of C are connected to at least one neighbor of u, and all but*

one vertex of C is connected to exactly one neighbor of u. Formally, let $2 \leq d' \leq d$, $v_1, \ldots, v_d$ be the neighbors of u (not necessarily all distinct), $u_1, \ldots, u_{d'}$ new vertices distinct form the vertices of G, and $G' = (V', E')$ with

$$\begin{aligned} V' &= V \cup \{u_1 \ldots, u_{d'}\} \setminus \{u\} \\ E' &= E \cup \{(u_i, v_i) : 1 \leq i \leq d' - 1\} \cup \{(u_{d'}, v_i) : d' \leq i \leq d\} \setminus \{(u, v_i) : 1 \leq i \leq d\}. \end{aligned}$$

Then G' is exactly 3-edge connected.

Proof. We shall prove, in order, that there are exactly 3 edge-disjoint paths in G' between these pairs of vertices:

1. (x, y) in $V \setminus \{u\}$
2. (x, u_i) for $x \in V \setminus \{u\}$ and $1 \leq i \leq d'$
3. (u_i, u_j) for $1 \leq i < j \leq d'$

For the first case, replacing vertex u by a subgraph, does not create new paths between x and y, so there are at most 3 edge-disjoint paths. Because u is not an articulation point, there exists triplets of edge-disjoint $x - y$ paths where at most 2 go through u. These 2 edge-disjoint $x - y$ paths in G can be changed into 2 edge-disjoint $x - y$ paths in G' by using some of the edges in the expanded cycle. So there are 3 edge-disjoint $x - y$ paths in G'.

For the second case, fix $i \in [1, d]$ and let P_1, P_2 and P_3 be 3 edge-disjoint $x - u$ paths in G. These paths can be changed to three paths P_1', P_2', P_3' from x to vertices on the expanded cycle, say u_{j_1}, u_{j_2} and u_{j_3}. If i is equal to j_1, then we have 3 edge-disjoint $x - u_i$ paths: P_1', P_2' plus $u_{j_2} - u_{j_1}$ on the expanded cycle and P_3' plus $u_{j_3} - u_{j_1}$on the expanded cycle. If i is not equal to j_1, j_2 or j_3, let consider a $x - v_{j_1} - u_{j_1}$ path P (which can be constructed from an $x - v_{j_1} - u_{j_1}$ cycle in G, and G is 3 edge connected). This path P is distinct from P_1', P_2' and P_3' as it contains the

edge (v_{j_1}, u_{j_1}). If P is edge-disjoint with at least two of P'_1, P'_2 and P'_3, then we are done. Otherwise, let e be the edge closest to u_{j_1} which is common between P and, say, P'_1. Create the new path P' equal to P'_1 from x to e, and equal to P from e to u_{j_1}. By construction, this is a $x - u_{j_1}$ path which is edge-disjoint with P'_2 and P'_3.

Finally, for the last case. For any pair (i, j), there is a cycle C_{ij} in G containing the vertices u, v_i and v_j. A piece of this cycle can be changed into a $v_i - v_j$ path P which is edge-disjoint with the expanded cycle. So there are 3 edge-disjoint $v_i - v_j$ paths in G': two paths in the expanded cycle and the path P. There cannot be more than 3 edge-disjoint paths as, by construction, at least one of u_i or u_j has degree 3. □

See Figure 4.1 for an example of cycle expansion. The following proposition, where a cycle is contracted into a new vertex, is the inverse of the previous one. Because not all cycles can be so contracted, stronger hypotheses must be made on the graph G and on the cycles considered. To state those conditions, we first need some facts and definitions.

Lemma 4.1. *An exactly k-edge connected graph G which has only trivial cuts is quasi k-regular.*

Proof. Suppose there exists two vertices u and v of degree greater than k. There exists a cut separating u and v and this cut cannot be trivial. □

Proposition 4.5 (Cycle contraction)**.** *Let $G = (V, E)$ be a quasi 3-regular, exactly 3-edge-connected and biconnected graph. Let C be an induced cycle of any length which contains the vertex of higher degree. The graph G' is constructed from G by collapsing the cycle C into one vertex. Formally, if $2 \leq d' \leq d$, $u_1, \ldots, u_{d'}$ are the vertices of C, and $v_1, \ldots, v_d$ are the vertices adjacent to C in G, we construct $G' = (V', E')$ with*

$$\begin{aligned} V' &= V \setminus \{u_1, \ldots, u_{d'}\} \cup \{u\} \\ E' &= E \cup \{(u, v_i) : 1 \leq i \leq d\} \setminus (\{(u_i, v_i) : 1 \leq i \leq d' - 1\} \cup \{(u_{d'}, v_i) : d' \leq i \leq d\}) . \end{aligned}$$

Then G' is exactly 3-edge connected and quasi 3-regular.

Proof. In G', the degree of every vertex in $V \setminus \{u_1, \ldots, u_{d'}\}$ is the same as its degree in G. Given that only $u_{d'}$ in G may have a degree different than 3, G' is also quasi 3-regular. In particular, there cannot be more than 3 edge-disjoint paths between any pair of vertices in G'. For any pair of vertices x, y not on C, any $x - y$ path in G can be changed into a path in G' where the eventual edges on the cycle are removed. Furthermore, edge-disjoint paths in G are still edge-disjoint in G'. So there are 3 edge-disjoint $x - y$ paths in G'. For any x not on C, any $x - u_1$ path in G can be changed into a $x - u$ path in G'. So there are 3 edge-disjoint paths in G' between any pair of vertices. □

Note that Proposition 4.5 only guarantees that cycle contraction preserves 3-edge-connectivity, but the resulting graph may not be biconnected. Preserving this second property is the subject of Section 4.3.

4.2.3 Synthesis.

We now proceed to the synthesis of exactly 3-edge-connected graphs. The proof will rely on results from Section 4.3 Theorem 4.3, where the main technical arguments are given.

Definition 4.7. The *dumbbell graph* consists of 2 vertices and 3 parallel edges between them. It is exactly 3-edge-connected.

Theorem 4.1 (Exactly 3-edge-connected synthesis)**.** *Any exactly 3-edge-connected graph G is obtained from dumbbell graphs and the following operations: cycle expansion and block gluing.*

Proof. We proceed by induction on the order r of G. The only exactly 3-edge-connected graph of order 2 is the dumbbell and the induction hypothesis holds for

$r = 2$. Suppose the theorem holds for any graph of order $j \leq r$. Let G be an exactly 3-edge connected graph of order $r + 1$. We apply the following tests:

1. If G has an articulation point, by Corollary 4.1, each block is exactly 3-edge-connected and of order less than $r + 1$. Apply the induction to each block; G is obtained by block gluing of its blocks.

2. G is biconnected and of order at least 3, so by Theorem 4.3 (below) it contains a quasi 3-regular induced cycle C which, when collapsed, does not create an articulation point and preserves exact 3-edge-connectivity. Let G' be obtained from G by collapsing C. The order of G' is less than $r + 1$. Apply the induction to G'; G is obtained by cycle expansion of G'.

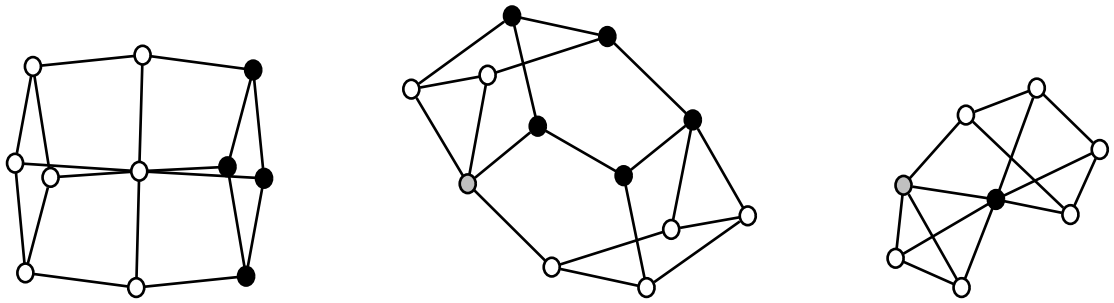
□

The main difficulty in the proof of Theorem 4.1 is that collapsing an arbitrary cycle could create an articulation point, but cycle expansion (Proposition 4.4) and cycle contraction (Proposition 4.5) are inverse operations of each other only as long as the contraction does not create an articulation point. This difficulty will be dealt with in the next section.

4.3 Existence of Non-articulation Cycles.

In this section, we shall prove the existence in an exactly 3-edge-connected graph of a quasi 3-regular induced cycle which, when collapsed, does not create an articulation point and preserves exact 3-edge-connectivity. This will culminate in Theorem 4.3, which was used to prove that all exactly 3-edge-connected graphs can be constructed from block-gluing and cycle expansion (Theorem 4.1).

Definition 4.8. Let H be a subgraph of a connected graph $G = (V, E)$ and let $B_1, \ldots, B_k$ be the sets of vertices of the connected components of $G \backslash H$. The partition $V = H \cup B_1 \cup \ldots \cup B_k$ induced by H is the *H-partition* and its size is k ($k \geq 0$).



(a) (b) (c)

Figure 4.2: The graph in (a) is 2-vertex-connected, 3-edge-connected, quasi 3-regular with minimum degree 3. The induced cycle C with the black vertices is non-separating but not collapsible. Collapsing C into a vertex v would create 4 paths between v and h, the high degree vertex at the center of the graph. The graph in (b) has two high degree vertices separated by a 3-edge-cut and the blackened induced cycle contains one high degree and goes accross the edge cut. Collapsing the cycle (figure (c)) creates 4 edge disjoint paths between the grey and black vertices.

Definition 4.9. An *articulation cycle* in $G = (V, E)$ is an induced cycle C with a C-partition of size greater than 1.

If C is an articulation cycle, then in the graph G' obtained from G by collapsing C into a vertex v, the sets $B_i \cup \{x\}, 1 \leq i \leq k$ are the blocks (biconnected components) of G'. By extension, we will call the sets B_i the *blocks* of the C-partition.

Definition 4.10. A *collapsible* cycle is a quasi 3-regular, non-articulation, induced cycle which preserves exact 3 edge-connectivity when collapsed.

An induced articulation cycle is an identical notion to an induced separating cycle. Several previous works [98, 97, 22, 29] deal with the problem of finding non-separating induced cycles in a graph, i.e. a induced cycle C in G such that $G \setminus C$ is connected. Thomassen's results [98] for 2-connected graphs of minimum degree 3 would guarantee the existence of such a non-separating induced cycle. But collapsing such cycle does not necessarily preserve exact 3-edge-connectivity as required for the synthesis (see Figure 4.2). A graph G' obtained from G by collapsing an induced non-separating cycle can have higher edge connectivity than G in two cases: collapsing C

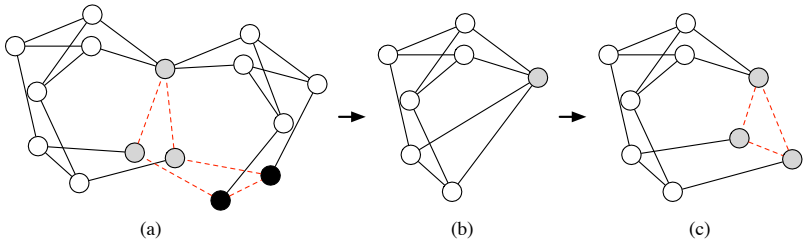


Figure 4.3: An example contraction-expansion of a cycle. (a) A quasi-3-regular exactly 3-edge-connected graph. (b) A block created by the collapse of the dashed cycle. (c) The re-expansion of the dashed cycle within the block of (b).

created a high-degree vertex which has more than 3 paths to an existing high-degree vertex of G, or cycle C contained edges of a non-trivial 3-edge-cut and collapsing C "destroyed" the cut. To prevent the former, we will impose that the cycle C contains one and only one high-degree vertex of G (hence, no new high degree vertex is created). To prevent the later, we will require that the cycle C avoids any arbitrarily chosen vertex u of G (which is sufficient, as we shall see in Theorem 4.3).

We introduce some definitions and operations relating to articulation points in exactly 3-edge-connected graphs.

Definition 4.11. Let G be a graph, C be an induced cycle in G and B be a block of the C-partition. The *contraction-expansion* of C with respect to B is a graph G' obtained from G by the following operations, applied sequentially:

- let G' be the graph induced by $B \cup C$
- smooth out the vertices of degree two in G'

The vertices of degree two that are smoothed out (replaced by edges) are the attachment vertices of the blocks other than B in the C-partition. The name contraction-expansion is justified by the proof of Lemma 4.2. See Figure 4.3 for an example of such a contraction-expansion.

Lemma 4.2. *Let G be an exactly 3-edge-connected graph, C be an induced cycle in G and B be a block of the C-partition. Then G', the contraction-expansion of C with*

respect to B, is exactly 3-edge-connected. Furthermore, if G is 3-regular then so is G'. If G is quasi 3-regular and C contains the high-degree vertex, then G' is quasi 3-regular and C' contains the high-degree vertex of G'.

Proof. G' can also be constructed by the following operations: contract the cycle C, let G' be the block which contains B and expand in G' the contracted cycle into a cycle C'. The first part of the theorem statement follows from the properties of the cycle contraction (Proposition 4.5), block gluing (Corollary 4.1) and cycle expansion (Proposition 4.4) operations. After a cycle expansion, all the vertices but one in the newly created cycle have degree 3. By construction, a vertex in C' has a degree less or equal to the corresponding vertex in C. Hence, the 3-regularity and quasi 3-regularity properties are preserved □

Lemma 4.3. *Let G be a 3-edge-connected graph. If G is 3-regular, then it is a simple graph. If G is quasi 3-regular, the high degree vertex h is an end point of any double edge.*

Proof. If G is 3-regular and has a double edge $(u, v)^2$ then u (resp. v) has only one other neighbor beside v (resp. u), say u' (resp. v'). The set $S = \{(u, u'), (v, v')\}$ is exactly the edge cut $\langle \{u, v\}, G \setminus \{u, v\} \rangle$ and has size 2, which is a contradiction. So G must be a simple graph. If G is quasi 3-regular and $(u, v)^2$ is a double edge not incident to h, then the same contradiction as above arises. So any double edge must be incident to h. □

Lemma 4.4. *Let G be a 3-regular, 3-edge-connected graph, (v, w) be an edge in G and u be a vertex in G distinct from v and w. Then G contains an induced cycle that contains edge (v, w) and does not contain u.*

Proof. Because there are 3 edge-disjoint paths between v and w, edge (v, w) is on at least 2 cycles C_1 and C_2. The cycles are vertex disjoint except for v and w because G is 3-regular. Hence, one of C_1 and C_2 does not contain u. If it contains chords, it

can be short-circuited to change it to an induced cycle, which still does not contain u. □

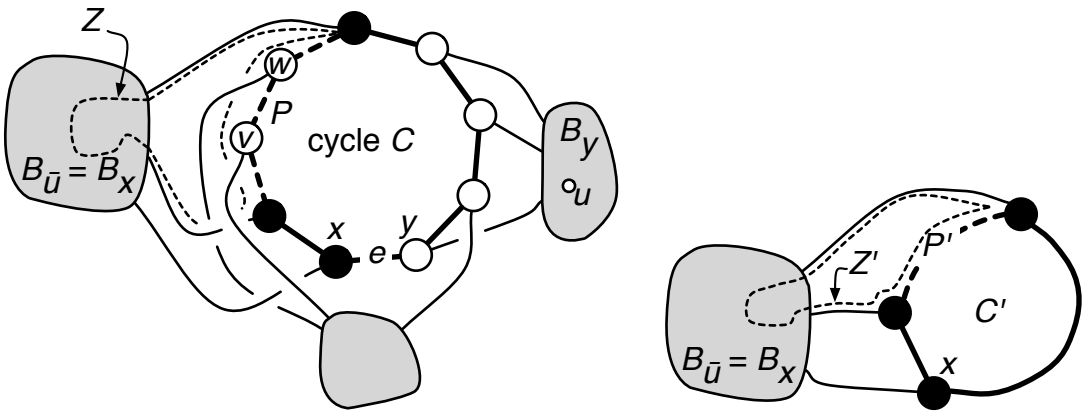
Lemma 4.5. *Let G be a quasi 3-regular, 3-edge-connected graph, (v, w) be an edge in G where v or w is the high-degree vertex and u be a vertex in G distinct from v and w. Suppose also that G has at most one double edge and u is an end point of the double edge if any. Then G contains an induced cycle that contains edge (v, w) and does not contain u.*

Proof. By an identical proof as in Lemma 4.4, there is a cycle C which contains (v, w) and not u. This cycle cannot have any double edge, and if it contains any chords it can be short-circuited. □

Lemma 4.6. *Let G be a 3-regular, 3-edge-connected graph, and let C be an induced articulation cycle with a partition of size $k > 1$. Each vertex of C connects to exactly one block. Color those vertices that connect to a given block B black and color the others white. Then there are at least 4 edges on C that connect a black vertex to a white vertex.*

Proof. Because C is a cycle, there must be an even number of black-white edges. If there were no such edges, then k would be 1. If there were only 2 such edges, they would form a 2-edge-cut. Hence $k \geq 4$. □

The following lemmas and subsequent theorem contain the main technical arguments required to ensure that a reversible cycle contraction can be performed on any biconnected, quasi 3-regular, exactly 3-connected graph. Theorem 4.2, which uses Lemma 4.7, will handle one of the cases of the induction proof of Theorem 4.1. In these proofs, we often use the the following technique: given an exactly 3-edge-connected graph G, we create a new exactly 3-edge-connected graph G' via contraction-expansion of some cycle C in G. We say a cycle $Z \neq C$ in G is the *corresponding cycle* to a cycle Z' in G' if Z uses every edge of Z' that is present in G, and if Z' uses an edge



(a) Graph G (b) Graph G'

Figure 4.4: Existence of a collapsible cycle in 3-regular graphs. G' is obtained from G by a contraction-expansion of cycle C with respect to B_x. Paths P and P' on cycles C and C' are given by thick dashed edges. Edge e is a bicolor edge that is not on path P. Cycles Z and Z' are marked with thin dashed edges.

(u, v) that is not in G then u, v are on cycle C and Z uses a path on C between u and v.

Lemma 4.7 (Collapsible cycles in 3-regular graphs)**.** *Let $G = (V, E)$ be a biconnected, exactly 3-edge connected 3-regular graph, let $(v, w) \in E$ be an edge and $u \in V$ be a vertex distinct from v and w. Then G contains a collapsible cycle which contains edge (v, w) and does not contain vertex u.*

Proof. By Lemma 4.3, G is a simple graph. We proceed by induction on n, the order of G. The lemma is true when $n = 4$ (K_4, the complete graph on 4 vertices, is the only such graph).

Let C be an induced cycle in G that contains (v, w) and doesn't contain u and that, among those, minimizes k, the size of the partition. The existence of such a cycle is guaranteed by Lemma 4.4. If $k = 1$, then C is the desired cycle.

For purpose of contradiction, suppose that $k > 1$, as in Figure 4.4. Vertex u is in only one block. Let $B_{\bar{u}}$ a block in the C-partition which does not contain u and color in C the $B_{\bar{u}}$-adjacent vertices black and the remaining vertices white. Because G is

3-regular and 3-edge-connected, C must contain at least 3 black vertices and at least 3 white vertices. By Lemma 4.6, there are at least 4 black-white edges. Therefore, there is a path P in C that starts and ends at black vertices, has all white internal vertices and uses edge (v, w). Then P does not use some black-white edge e. Let x and y be respectively the black vertex and white vertex of edge e, with associated blocks B_x (which is equal to $B_{\bar{u}}$) and B_y.

Let G' be the contraction-expansion of C with respect to B_x. Because vertex x was colored black it is still present in C' and to the path P in C corresponds an edge P' of C' (every white vertex was smoothed out). By Lemma 4.2, G' is 3-regular and exactly 3-edge-connected. G' has fewer vertices than G and by induction there exists a collapsible cycle Z' which contains the edge P' and does not contain vertex x. Because $x \in C'$, the cycles Z' and C' are distinct. Because $P' \subset Z' \cap C'$, Z' and C' are not disjoint.

Let Z be the cycle in G corresponding to Z' and let k' be the size of the Z-partition. Because Z is contained in $C \cup B_x$, if a and b are two vertices in a block B (distinct from B_x) of the C-partition, a and b must belong to the same block B' in the Z-partition. Moreover, Z' was not an articulation cycle in G', hence if a, b are two vertices in $B_x \setminus Z$, they must belong to the same block of the Z-partition. Hence, there is a surjective function from the C-partition to the Z-partition and $k' \leq k$. If B is a block in the C-partition, let note B' the unique block in the Z-partition containing B.

By construction, vertex x is not contained in the cycle Z, so the black-white edge e is also not contained in the cycle Z. Also, in $G \setminus Z$ there is a path between B'_x (the block containing x) and B'_y (the block containing y). In other words, $B'_x = B'_y$, the function from the C-partition to the Z-partition is not injective and $k' < k$. This contradicts the minimality of k and concludes the induction step. □

Theorem 4.2 (Collapsible cycles in quasi 3-regular graphs)**.** *Let $G = (V, E)$ be a biconnected, exactly 3-edge-connected, quasi 3-regular graph with high-degree vertex h and let u be a vertex of G distinct from h. Then G contains a collapsible cycle which contains h and does not contain u.*

Proof. If G is 3-regular, then this is Lemma 4.7. So we assume that G has a vertex h of degree > 3. If G has a double edge which is not incident to u (in particular if there are two or more double edges), then this double edge is a collapsible cycle, which contains h by Lemma 4.3, and avoids u. So we assume that G has at most one double edge, and if it does, u is an end point of the double edge. We proceed similarly to Lemma 4.7 and prove the statement by induction on n, the order of G. The theorem is true when $n = 4$ (again, K_4 is the only such graph).

Let C be an induced cycle in G that contains h and does not contain u and that, among those, minimizes the size k of the partition. The existence of such a cycle is guaranteed by Lemma 4.5. If $k = 1$, then C is the desired cycle. For purpose of contradiction, suppose that $k > 1$. Let $B_{\bar{u}}$ be a block in the C-partition which does not contain u. Color in C the $B_{\bar{u}}$-adjacent vertices black and the other vertices white. The two cases, where h is adjacent to $B_{\bar{u}}$ or not, are now discussed.

If h is adjacent to $B_{\bar{u}}$, then h is colored black. Because C is a cycle, there are at least 2 black-white edges. Let e be any black-white edge which does not have h as one of its end point. Edge e must exists otherwise h would be an articulation point. Let x and y be respectively the black vertex and white vertex of edge e, with associated blocks $B_x = B_{\bar{u}}$ and B_y. Consider G' the contraction-expansion of C with respect to B_x. G' contains h as h is black and, by Lemma 4.2, G' is quasi 3-regular and exactly 3-edge-connected. G' has fewer vertices than G and by induction there exists a collapsible cycle Z' which contains h and does not contain x. The corresponding cycle Z in G contains h and does not contain edge e.

If h is not adjacent to $B_{\bar{u}}$ then h is colored white. Let P be a $v - w$ path in C between two black vertices such that h is an internal vertex of P and all the internal vertices of P are white. Let e be any black-white edge which does not have v or w as an end point and let x and y be respectively the black vertex and white vertex of edge e, with associated blocks $B_x = B_{\bar{u}}$ and B_y. Let G' be the contraction-expansion of C with respect to V. To the path P now corresponds the edge (v, w) in C' and x is in G' because it is black. The high degree vertex h of G is not in G' because it is white, hence, by Lemma 4.2, G' is 3-regular and exactly 3-edge-connected. By Lemma 4.7, there exists a cycle Z' in G' which contains the edge (v, w) and does not contain vertex x. The corresponding cycle Z in G contains P and does not contain x. Hence cycle Z contains h and does not contain edge e.

In both cases, a cycle Z containing h and not containing a black-white edge e exists. An identical argument as in Lemma 4.7 shows that the Z-partition has size $k' < k$, which contradicts the minimality of k. □

We now prove the existence of a collapsible cycle in the general case, where the only conditions on G are that it is biconnected and exactly 3-edge-connected. We start with a lemma that guarentees that such a graph has vertices of degree 3. The trees, which are the exactly 1-edge-connected graph, have two leaves (vertices of degree 1), and the exactly 2-edge-connected graphs also have two vertices of degree 2. Similarly, we have the following property for an $k \geq 3$.

Lemma 4.8. *Let G be a biconnected, exactly k-edge-connected graph of order $\geq k$. Then G has at least 2 vertices of degree k.*

Proof. We proceed by induction on the number of non-trivial minimum cuts in G. If G has no non-trivial minimum cuts, then it is quasi k-regular. If G is k-regular, it has at least 2 vertices of degree k. If G is quasi k-regular with a high degree vertex, it has at least 3 vertices and therefore has at least 2 vertices of degree k.

Let $S = \langle V_1, V_2 \rangle$ be a non-trivial minimum cut and let G_1 and G_2 be the vertex splitting graphs induced by S. Call x_1 and x_2 the new vertices ($V(G_i) = V_i \cup \{x_i\}$). Suppose T were a non-trivial minimum cut in G_1. Construct a corresponding non-trivial minimum cut T' in G by changing any edge used by T that is adjacent to x_1 to the corresponding edge in S. So to any non-trivial minimum cut of G_1 or G_2 corresponds a distinct non-trivial minimum cut in G. But no non-trivial cut in G_1 or G_2 corresponds to the cut S (it would be a trivial cut in G_1 and G_2). So both G_1 and G_2 have fewer non-trivial minimum cuts than G. By induction G_1 and G_2 have 2 vertices of degree k, including x_1 and x_2. So G is obtained as a vertex gluing of G_1 and G_2 and has 2 vertices of degree k. □

Theorem 4.3 (Collapsible cycles)**.** *Let G be a biconnected, exactly 3-edge-connected graph of order at least 3 and u a vertex of degree 3. Then G contains a collapsible cycle which does not contain u.*

Proof. By induction on the number of non-trivial minimum cuts in G. If G has no non-trivial minimum cuts, then it is quasi 3-regular. By Theorem 4.2 it contains a collapsible cycle which avoids u.

Assume G has non-trivial minimum cuts. Let $S = \langle V_1, V_2 \rangle$ be a non-trivial minimum cut, $|S| = 3$, and let G_1 and G_2 be the vertex splitting graphs induced by S. Assume without loss of generality that G_1 does not contain u. By the same argument as in Lemma 4.8, G_1 has less non-trivial minimum cuts than G and by induction, G_1 has a collapsible cycle C_1 which avoids x_1.

C_1 in G is not an articulation cycle. Moreover, contracting C_1 in G also preserves exact 3-edge connectivity: collapsing a cycle does not destroy any path between the remaining vertices, it does not create 4 paths in V_1 by construction, and it does not create 4 paths between V_1 and V_2 because of the minimum cut S. Therefore, C_1 is a collapsible cycle of G which does not contain u. □

4.4 Other Properties of Exactly k-edge-connected Graphs.

4.4.1 Number of Operations.

Let $G = (V, E)$ be an exactly 3-edge-connected graph with $n_G = |V|$ and $m_G = |E|$, and let B_G be the number of blocks in G. The number N_G of synthesis operations needed to generate G is determined by m_G, n_G and B_G.

Proposition 4.6. *If $G = (V, E)$ is an exactly 3-edge-connected graph, then*

$$m_G = n_G + 2B_G + E_G - 1, \tag{4.1}$$

$$N_G = m_G - n_G - B_G \tag{4.2}$$

where E_G is the number of cycle expansions in a synthesis of G.

Proof. A dumbbell graph has 3 edges, 2 vertices and one block. It satisfies $m = n + 2B - 1$ ($B = 1$). By induction, suppose G satisfies Equation 4.1. A cycle expansion adds $d - 1$ new vertices and d new edges for some d. Hence, G' obtained from G by a cycle expansion satisfies $m_{G'} = m_G + d$, $n_{G'} = n_G + d - 1$, $B_{G'} = B_G$ and $E_{G'} = E_G + 1$, hence it satisfies Equation 4.1. Similarly, if G is the block gluing of G_1 and G_2 which both satisfy Equation 4.1, then G satisfies $m_G = m_{G_1} + m_{G_2}$, $n_G = n_{G_1} + n_{G_2} - 1$, $B_G = B_{G_1} + B_{G_2}$ and $E_G = E_{G_1} + E_{G_2}$, hence G satisfies Equation 4.1.

To create B_G blocks, $B_G - 1$ block gluing operations are needed. So the the number of operations in a synthesis of G is $N_G = B_G - 1 + E_G = m_G - n_G - B_G$. □

4.4.2 Minimum Exactly Connected Graphs.

A natural requirement for network design is to use the fewest edges possible. A k-vertex-connected or k-edge-connected graph with n vertices has at least $\lceil \frac{kn}{2} \rceil$ edges. We say a graph is *minimum* if it has exactly that many edges. The Harary [30]

graph $H_{k,n}$ is a special graph which has n vertices, is k-vertex-connected and k-edge-connected, and is minimum. They are, in addition, quasi k-regular, which implies that they are also exactly k-edge-connected. In fact, we have following relationships between minimum, almost k-regular, and exactly k-edge-connected graphs:

Definition 4.12. A graph G is *almost k-regular* if it is quasi k-regular and has maximum degree $\leq k+1$.

Proposition 4.7. *Let G be a graph. The following assertions are equivalent:*

1. *G is k-edge-connected and minimum.*
2. *G is k-edge-connected and almost k-regular.*
3. *G is exactly k-edge-connected and almost k-regular.*

We therefore have that, among the k-edge-connected graphs, the minimum graphs are a strict subset of the exactly connected graphs, which in turn are a strict subset of the edge-minimal graphs. The minimum, exactly 3-edge-connnected graphs can be generated by disallowing a cycle expansion if it would lead to $\sum_v \deg(v) > 3n+1$.

4.4.3 Planar Graphs.

Planar, exactly 3-edge-connected graphs can be generated with a slightly modified synthesis. If G is a planar, exactly 3-edge-connected graph and u is a vertex, a drawing of G on the plane defines a natural order on the neighbors of u (for example, clockwise traversal of the edges incident to u). An *order-preserving cycle expansion* is a cycle expansion of u where the vertices in the cycle are attached to the neighbors of u in the natural order. G' obtained from G by an order-preserving cycle expansion is also a planar graph. Conversely, if G is planar and C is a collapsible cycle, then C must be a face and G' obtained by collapsing C is also planar.

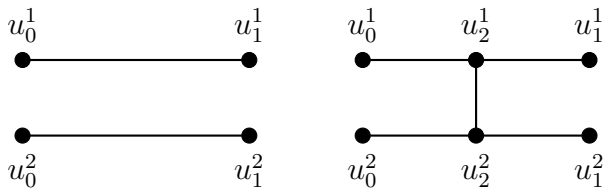


Figure 4.5: A H-expansion.

Theorem 4.4 (Planar exactly 3-edge-connected synthesis)**.** *A graph is planar and exactly 3-edge-connected if and only if it can be obtained from dumbbell graphs and the following operations: order-preserving cycle expansion and block gluing.*

The class of planar 3-regular 3-connected graphs has been heavily studied in the context of the 4-color theorem [71]. In particular, the fact that every planar 3-regular 3-connected graph has a proper edge 3-coloring is equivalent to the 4-color theorem. E. L. Johnson [39] gave a synthesis for this class of graphs. Theorem 4.3 leads to an alternative proof.

A H-expansion, as shown in Figure 4.5, is adding a new edge between two edges of the same face.

Theorem 4.5 (E. L. Johnson)**.** *Every planar cubic 3-connected graphs G on $|V| \geq 6$ vertices is obtained from a planar cubic 3-connected graph on $|V| - 2$ vertices by a H-expansion.*

Proof. Let G be a planar cubic 3-connected graph. By Theorem 4.3, there exists a collapsible cycle and let G' be obtained from G by an order-preserving collapsing of this cycle into vertex h'. If G' is the dumbbell, then G would have only 4 vertices. Hence G' contains a collapsible cycle containing h' (h' is the only vertex of degree higher than 3 in G', if any). Let G'' be obtained from G' by an order-preserving collapsing of this cycle into vertex h''. Finally, let G''' be the graph obtained from G'' by an order-preserving expansion of h'' into a cycle of size equal to the degree of

h''. Then G'' is planar 3-regular and 3-connected. G'' has one less edge and two less vertices than G. G is obtained from G'' by adding this edge, which correspond to a H-expansion. □

4.4.4 Exactly k-edge-connected and k-regular Graphs.

Although exactly k-edge-connected graph can have vertices of arbitrarily large degree, they share some properties with k-regular graphs. Ding and Chen [18] gave a general synthesis for k-regular graphs, for all k. Nevertheless, their synthesis does not lead directly to a synthesis of exactly k-edge-connected graphs.

Let G be an exactly k-edge-connected graph and let $h_1, \ldots, h_l$ be the vertices of G of degree greater than k. Let $H_1, \ldots, H_l$ be $(k-1)$-regular $(k-1)$-connected graphs such that the number of vertices in H_i is equal to the degree of h_i. Create G' by replacing, for all $i \in [1, l]$, in G vertex h_i by the graph H_i and connecting the neighbors of h_i to the vertices H_i in a one-to-one fashion. Then G' is k-regular, k-connected.

Conversely, if G' is a k-regular k-connected graph and $H_1, \ldots, H_l$ are $(k-1)$-regular induced subgraph of G'. Assume furthermore that for every pairs $i, j \in [1, l]$ there exists an edge cut of size k which separates H_i and H_j. Create G by collapsing in G' all subgraphs H_i to a single vertex h_i. Then G is exactly k-edge-connected.

Hence, we have the following characterization of exactly k-edge-connected graphs: they are obtained by a Ding-Chen [18] synthesis for k-regular graphs followed by collapsing some $(k-1)$-regular induced subgraphs. This is not a complete synthesis for two reasons. First, the Ding-Chen synthesis does not guarantee the graph is k-edge connected. Second, the collapsing operation is not a simple operation because finding a $(k-1)$-regular induced subgraph is an NP-complete problem.

4.4.5 Vertices of min-degree in edge minimal k-edge-connected graphs.

Halin [32] showed that every edge-minimal k-vertex-connected graph has at least one vertex of degree k. A similar assertation that edge-minimal k-edge-connected graphs have a vertex of degree k was proved by Lick [53]. This property of having a vertex of degree k is true for exactly k-edge-connected as shown in Lemma 4.8. The following proposition extends this property from exactly k-edge-connected graphs to edge-minimal k-edge-connected graphs and gives an alternate proof.

Proposition 4.8. *Let $G = (V, E)$ be a λ-edge-connected graph. G is edge minimal if and only if for any adjacent vertices $(u, v) \in E$, there are at most λ edge-disjoint $u - v$ paths.*

Proof. Let $e = (u, v) \in E$ such that there are $> \lambda$ edge-disjoint $u - v$ paths. In $G - e$, there are at least λ edge-disjoint $u - v$ paths. Let x and y be any two vertices, distinct from u and v. There are λ edge-disjoint $x - u$ paths in G. So, depending on whether or not edge e is on one of these $x - u$ paths, in $G - e$ there are either λ edge-disjoint $x - u$ paths, or there are $\lambda - 1$ edge-disjoint $x - u$ paths and a $x - v$ path edge-disjoint from the $\lambda - 1$ $x - u$ paths. There is a similar situation in $G - e$ between y, v and u. Let S be a set of $\lambda - 1$ distinct edges of $G - e$. In $G - e - S$, there is a $x - u$ or $x - v$ path, a $y - u$ or $y - v$ path and a $u - v$ path. Hence there is a $x - y$ path and S is not a separating set. By Menger's theorem, $G - e$ is λ-edge-connected, and G is not edge minimal. If x or y is equal to u or v, a similar argument can be made.

Conversely, suppose that for any edge $e = (u, v) \in E$ there are at most λ edge-disjoint $u - v$ paths. Then in $G - e$ there are at most $\lambda - 1$ edge-disjoint $u - v$ paths. So G is edge minimal. □

Theorem 4.6. *Every edge-minimal k-edge-connected graph has 2 vertices of degree k.*

Proof. As mention in the introduction, if G is a k-edge-connected graph and the sets of vertices that are mutually connected by $> k$ edge-disjoint paths are collapsed into supernodes, then the resulting multigraph G' is either a graph of a single vertex or is exactly k-edge-connected. If G is also edge-minimal, then G' cannot be a single vertex and by the previous proposition, any two vertices of G in the same supernode of G' are not adjacent. Hence the degree of a vertex (supernode) in G' is greater or equal to the degree in G of any vertex in that supernode. By Lemma 4.8, G' has 2 vertices of degree k which correspond to 2 vertices of degree k in G. □

Chapter 5

Parsimonious Reconstruction of Network Evolution

To appear in WABI 2011.

This project is a group effort with Rob Patro, Emre Sefer, Justin Malin, Saket Navlakha and Carl Kingsford. I was mostly involved with the development of the framework and algorithm used to solve the reconstruction problem. The growth model was developed by Justin Malin while the implementation and testing was performed by Emre Sefer and Rob Patro. We all took part in the writing of this paper.

5.1 Introduction

High-throughput experiments have revealed thousands of regulatory and protein protein interactions that occur in the cells of present-day species. To understand why these interactions take place, it is necessary to view them from an evolutionary perspective. In analogy with ancestral genome reconstruction [72], we consider the problem of predicting the topology of the common ancestor of pathways, complexes, or regulatory programs present in multiple extant species.

Generating plausible ancestral networks can help answer many natural questions that arise about how present-day networks have evolved. For example, joint histories can be used to compare the conservation and the route to divergence of corresponding processes in two species. This allows us to more finely quantify how modularity has changed over time [43] and how interactions within a protein complex may have reconfigured across species starting from a single shared state [75]. Such analysis can

also be integrated to develop better network alignment algorithms [23, 91, 20] and to study robustness and evolvability [1]. Further, inferred changes in metabolic networks can be linked to changes in the biochemical environment in which each species has evolved, and this can reveal novel mechanisms of ecological adaptation [6, 5]. Finally, comparing network histories inferred using different model parameters can be used to estimate the likelihoods of various evolutionary events [63, 70].

There has been some recent work on reconstructing ancestral interactions. Gibson and Goldberg [27] presented a framework for estimating ancestral protein interaction networks that handles gene duplication and interaction loss using gene trees reconciled against a species phylogeny. However, their approach assumes that interaction losses occur immediately after duplication and does not support interaction gain outside of gene duplication. These assumptions are limiting because interaction loses may occur well after duplication, and independent gains are believed to occur at non-trivial rates [49]. Dutkowski and Tiuryn [20] provided a probabilistic method for inferring ancestral interactions with the goal of improved network alignment. Their approach is based on constructing a Bayesian network with a tree topology where binary random variables represent existence or non-existence of potential interactions. A similar graphical model was proposed by Pinney et al. [77], who applied it to inferring ancestral interactions between bZIP proteins. In the former method, edge addition and deletion is assumed to occur only immediately following a duplication or speciation event. Further, both methods assume the relative ordering of duplication events is known even between events in unrelated homology groups. Pinney et al. [77] also explore a parsimony-based approach [66] and find it to work well; however, it too assumes a known ordering of unrelated duplication events. The main drawback of these approaches is that the assumed ordering comes from sequence-derived branch lengths, which do not necessarily agree with rates that would be estimated based on

network evolution [104]. This motivates an approach such as we describe below that does not use branch lengths as input.

Zhang and Moret [104, 103] use a maximal likelihood method to reconstruct ancestral regulatory networks as a means to improve estimation of regulatory networks in extant species. Mithani et al. [67] study the evolution of metabolic networks, but they only model the gain and loss of interactions amongst a fixed set of metabolites, whereas we also consider node duplication and loss encoded by a tree. Navlakha and Kingsford [70] present greedy algorithms for finding high-likelihood ancestral networks under several assumed models of network growth. They applied these methods to a yeast protein interaction network and a social network to estimate relative arrival times of nodes and interactions and found that the inferred histories matched many independently studied properties of network growth. This attests to the feasibility of using networks to study evolution. The authors, however, only consider a single network at a time, and there is no guarantee that independent reconstruction of two networks converge to a common ancestor.

Here, we introduce a combinatorial framework for representing histories of network evolution that can encode gene duplication, gene loss, interaction gain and interaction loss at arbitrary times and does not assume a known total ordering of duplication events. We show that nearly-minimal parsimonious histories of interaction gain and loss can be computed in practice quickly given a duplication history. In simulated settings, we show that these parsimonious histories can be used to accurately reconstruct a common ancestral regulatory network of two extant regulatory networks.

5.2 A framework for representing network histories

Any natural model of network evolution will include events for gene duplication, gene loss, edge gain, and edge loss. Many such growth models have been studied

(e.g. [12, 96, 74, 36, 1, 103, 102]). We now describe how these events can be encoded in a history graph.

Consider a set V of proteins descendent from a common ancestor by duplication events. Those duplication events can be encoded in a binary *duplication tree* T with the items of V as the leaves. An internal vertex u in T represents a duplication event of u into its left and right children, u_L and u_R. In this representation, after a duplication event, the protein represented by u conceptually does not exist anymore and has been replaced by its two children. A collection of such trees is a *duplication forest* F. The leaves of a duplication tree are labeled *Present* or *Absent*. Absent leaves represent products of duplication events that were subsequently lost.

The gain and loss of interactions can be represented with additional non-tree edges placed on a duplication forest. A non-tree edge $\{u, v\}$ represents an *edge flip event*, where the present / absent state of the edge between u and v is changed to Present if the edge is currently Absent or to Absent if the edge is currently Present. Let P_u and P_v be the paths from nodes u and v to the root. An edge exists between u and v if there are an odd number of such flip non-tree edges between nodes in P_u and P_v. Every non-tree edge between P_u and P_v, therefore, represents alternatively an edge creation or deletion between proteins u and v in the evolution of the biological network.

A graph H consisting of the union of a duplication forest and flip non-tree edges is a *network history*. A history H *constructs* a graph G when the Present leaves of the duplication forest in H correspond to the nodes of G and the flip edges of H imply an edge between u and v if and only if $\{u, v\}$ is an edge in G. See Figure 5.1 for an example history.

Not all placements of non-tree edges lead to a valid network history. The edge histories have to be consistent with some temporal embedding of the tree. Let t_u^c and t_u^d be respectively the time of creation and duplication of vertex u. Naturally, $t_u^c < t_u^d$,

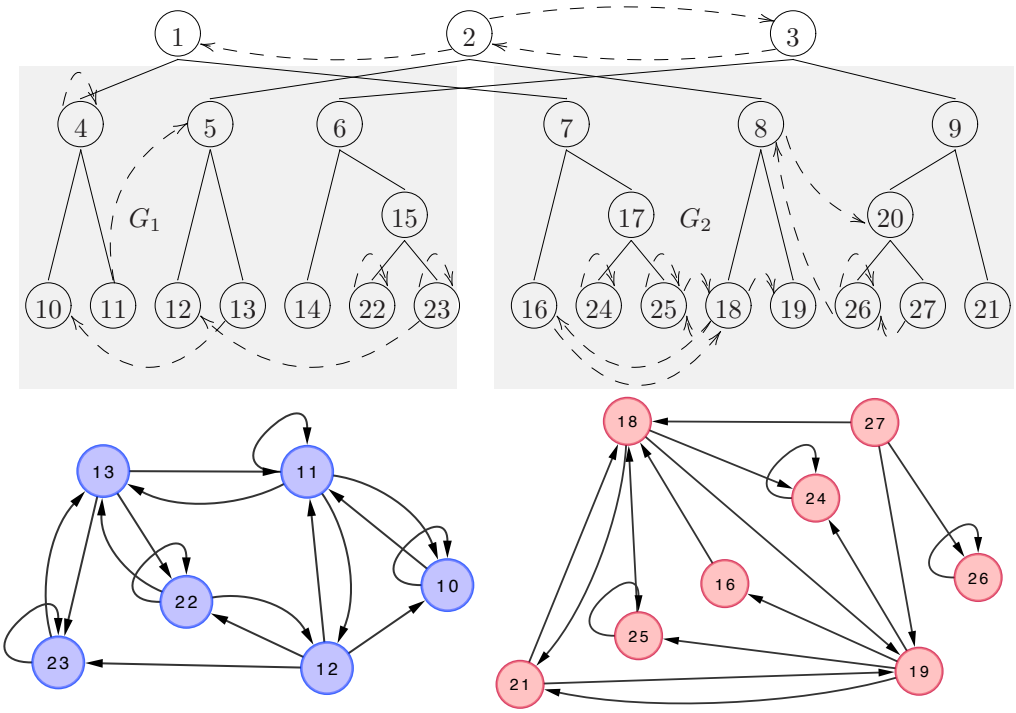


Figure 5.1: A duplication forest (solid edges at top) with the non-tree edges (dashed) necessary to construct G_1 and G_2 (shown at bottom). Nodes 1, 2, and 3 represent the 3 homology groups present in the ancestral graph.

$t_u^d = \infty$ if u is a Present leaf, and if v is the child of u, then by definition we have

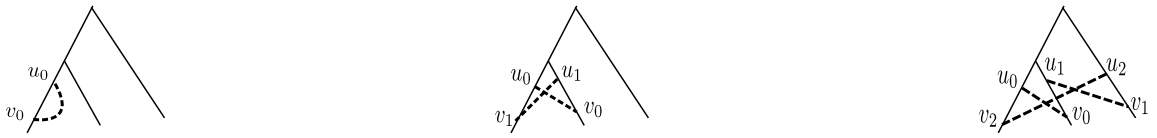
$$t_u^c < t_u^d = t_v^c < t_v^d. \tag{5.1}$$

If $\{u, v\}$ is a flip edge, then the time $t_{\{u,v\}}$ of appearance of this edge must satisfy

$$t_u^c \leq t_{\{u,v\}} < t_u^d \quad \text{and} \quad t_v^c \leq t_{\{u,v\}} < t_v^d, \tag{5.2}$$

because an event between u and v can only occur when both u and v exist. A history graph H is said to be *valid* if there exist t_u^c, t_u^d for every node u such that conditions (5.1) and (5.2) are satisfied for every non-tree edge.

Whether a particular history is valid can be checked combinatorially using the following alternative characterization of validity. A *k-blocking loop* is a set of flip edges



(a) 1 (b) 2 (c) 3

Figure 5.2: Blocking loops of size 1, 2 and 3. The solid lines represent a subset of the tree T. The dashed lines are non-tree edges representing edge flip events.

$\{\{u_i, v_i\}\}_{0 \le i < k}$ such that u_{i+1} is an ancestor of v_i in the tree for $0 \le i < k$ (where the index $i+1$ is taken modulo k). See Figure 5.2 for examples. Blocking loops are not permitted in valid histories and, conversely, the non-existence of blocking loops implies that a history is valid, as shown in Prop. 5.1.

Proposition 5.1. *A history graph H is valid if and only if it does not have any blocking loop of any length.*

Proof. Suppose there is a k-blocking loop. Using the same notation as above, we have the inequalities

$$t^d_{u_0} > t_{\{u_0,v_0\}} \ge t^c_{v_0} \ge t^d_{u_1} > t_{\{u_1,v_1\}} \ge \ldots \ge t^c_{v_{k-1}} \ge t^d_{u_0},$$

which is a contradiction. Hence, to not have any blocking loops is necessary.

Conversely, suppose that H does not have any blocking loops. We assign times to the vertices and non-tree edges using a modified depth first search (DFS) algorithm following the tree edges only (see Algorithm 3). First, the root of the tree is given a creation time of 0. During DFS, just before calling DFS recursively on the left and right children of a node u, we set the duplication time $t^d_u = \max\{\max t_{\{u,v\}} + 1, t^c_u + 1\}$, where the second max is taken over all non-tree edges adjacent to u. Also, we set the creation time of the children $t^c_{u_L} = t^c_{u_R} = t^d_u$.

When DFS visits a vertex u with some edge $\{u, v\}$ where v has not been assigned a creation time, u is added to a set Q and DFS is not called recursively on the children

of u. The main loop consist in calling DFS again on all the vertices in Q until this set is empty. By construction, the algorithm assigns times which satisfy conditions (5.1) and (5.2). Therefore, if the algorithm terminates, H is a valid history.

Algorithm 3 Modified DFS.

procedure TIMES(Q set of roots of history forest)
  for all $u \in Q$ **do**
    $t_u^c \leftarrow 0$
  end for
  while $Q \neq \emptyset$ **do**
    $Q' \leftarrow \emptyset$
    for all $u \in Q$ **do**
      $Q' \leftarrow Q' \cup \mathrm{DFS}(u, \emptyset)$
    end for
    $Q \leftarrow Q'$
  end while
end procedure

procedure DFS(u root vertex, Q vertex set)
  for all flip edge {u,v} s.t. t_v^c is set and $t_{\{u,v\}}$ is not set **do**
    $t_{\{u,v\}} \leftarrow \max\{t_u^c, t_v^c\}$
  end for
  if $\exists$ flip edge $\{u, v\}$ and t_v^c is not set **then**
    return $Q \cup \{u\}$
  else if u is a leaf **then**
    return Q
  else
    $t_u^d, t_{u_L}^c, t_{u_R}^c \leftarrow \max\left\{\{t_{\{u,v\}}\}_{\text{flip edge } \{u,v\}}, t_u^c + 1\right\}$
    return $Q \cup DFS(u_L, \emptyset) \cup DFS(u_R, \emptyset)$
  end if
end procedure

At each main iteration, the vertices in the set Q are all the vertices u for which t_u^c is set but t_u^d is not set. It suffices to show that at each such iteration, at least one of the vertices in the set Q will not be added again to Q by a call to DFS. In other words, for at least one vertex $u \in Q$, every non-tree edge $\{u, v\}$ has t_v^c set. For a contradiction, suppose not. Take $u_1 \in Q$ and $\{u_1, v_1\}$ with $t_{v_1}^c$ not set. There is necessarily an ancestor of v_1, call it u_2, which is in Q. Similarly, take $\{u_2, v_2\}$ with

$t^c_{v_2}$ not set and its ancestor $u_3 \in Q$, and so on. Because Q is finite, $u_j = u_i$ for some $j > i$, and we constructed a blocking loop. Hence, the algorithm must terminate. □

5.3 Parsimonious reconstruction of a network history

Traditional phylogenetic inference algorithms and reconciliation between gene species trees can be used to obtain duplication and speciation histories. What remains is the reconstruction of edge gain and loss events. This leads to the following problem:

Problem 5.1. Given a duplication forest F and an extant network G, find H, a valid history constructing G, with a minimum number of flip edges.

We will show that nearly optimal solutions to this problem for a large range of instances can be solved in polynomial time in practice. Whether Problem 5.1 is NP-hard or admits a polynomial-time algorithm for all instances remains open.

5.3.1 A fast heuristic algorithm

The challenge of Problem 5.1 comes from avoiding the creation of blocking loops. However, a polynomial-time algorithm can find a minimum set of flip edges that reconstructs a graph G and does not contain 1- and 2-blocking loops but allows longer blocking loops. We define an *edge encoding* of $G = (V, E)$ as a function $f_G : V \times V \to \{0, 1\}$ such that: $f_G(u, v) = 1$ if $\{u, v\}$ is an edge in G and $f_G(u, v) = 0$ otherwise. We omit the subscript on f_G if G is clear from the context.

The following intertwined dynamic programming recurrences find the minimum number of flip edges required for H to construct a given graph G if blocking loops of length ≥ 3 are allowed. First, $S(u, f)$ finds the minimum number of flip edges for the subtree rooted at u and edge encoding f:

$$S(u, f) = S(u_L, f) + S(u_R, f) + A(u_L, u_R, f). \tag{5.3}$$

The expression $A(u, v, f)$ gives the minimum number of flip edges that should be placed between the subtree rooted at u and the subtree rooted at v. This can be computed using the recurrence:

$$A(u,v,f) = \min \begin{cases} A(u_L, v, f) + A(u_R, v, f) \\ A(u, v_L, f) + A(u, v_R, f) \\ 1 + A(u_L, v, \bar{f}) + A(u_R, v, \bar{f}) \\ 1 + A(u, v_L, \bar{f}) + A(u, v_R, \bar{f}). \end{cases} \tag{5.4}$$

In the above, if one of u or v is a leaf but the other is not, the options that look at non-existent children are disallowed.

The function $\bar{f}$ in Eqn. (5.4) is defined as $1 - f$ and thus represents a function such that $\bar{f}(x)$ has opposite parity from $f(x)$ for all x. The A recurrence considers two possible options: (1) We connect u and v with a non-tree edge, this costs us 1 and flips the parity of all edges going between the subtree rooted at u and the subtree rooted at v; or (2) We do not connect u and v with a flip edge. This costs 0 and keeps the parity requirement the same. Regardless of the choice to create an edge, since we are not allowed to have a 2-blocking loop, either (a) we possibly connect u to some descendant of v (and do not connect v to a descendant of u) or (b) we possibly connect v to some descendant of u (and do not connect u to a descendant of v).

The base case for the S recurrence when u is a leaf and the base case for the A recurrence when u and v are leaves are:

$$S(u, f) = 0 \quad \text{and} \quad A(u, v, f) = f(u, v).$$

The minimum number of flip edges needed to make F a history constructing G (allowing blocking loops of ≥ 3) is then given by $\sum_r S(r, d_G) + \sum_{r,q} A(r, q, d_G)$,

where d_G is the edge encoding of G, and the sums are over roots r, q of the trees in the forest F. Standard backtracking can be used to recover the actual minimum edge set. The dynamic program runs in $O(n^2)$ time and space because only two functions f are ever considered: d_G, and $\bar{d_G}$. This yields $\approx n \times n \times 2$ subproblems, each of which can be solved in constant time.

5.3.2 Removing blocking loops

If solution contains blocking loops of length ≥ 3, we can relax the solution by progressively excluding the non-tree edge that participates in the largest number of loops and rerunning the dynamic program. We repeat this until a valid solution is obtained. In the worst case, one may obtain a solution where all non-tree edges are placed at leaves, but in practice long blocking loops do not often arise, and the obtained solutions are close to optimal (see Sec. 5.4.2).

5.3.3 Reconstruction of a common ancestor of two graphs

Given extant networks of several species, in addition to the reconstructed history, we seek a parsimonious estimate for their common ancestor network. Specifically, given extant networks G_1 and G_2, with edge encodings d_1 and d_2, and their duplication forests F_1 and F_2, we want to find an ancestral network $X = (V_X, E_X)$ such that the cost of X evolving into G_1 and G_2 after speciation is minimized. V_X is the set of roots of the homology forest. We assume that the networks of the two species evolved independently after speciation. Therefore, we can use the recurrence above applied to F_1 and F_2 to compute $A_{F_1}(r, q, d_1)$ and $A_{F_2}(r, q, d_2)$ independently for $r, q \in V_X$, and then select edges in X as follows. E_X of X is given by the pairs $r, q \in V_X \times V_X$ for which creating an edge leads to a lower total cost than not creating an edge.

Formally, we place an edge $\{r, q\}$ in E_X if

$$1 + A_{F_1}(r, q, \bar{d}_1) + A_{F_2}(r, q, \bar{d}_2) < A_{F_1}(r, q, d_1) + A_{F_2}(r, q, d_2). \tag{5.5}$$

Rule (5.5) creates an edge in X if doing so causes the cost of parsimonious histories inferred for G_1 and G_2 between the homology groups associated with r and q to be smaller than if no edge was created.

5.3.4 Modifications for self-loops

Self-loops (homodimers) can be accommodated by modifying recurrence (5.3):

$$S'(u, f) = \begin{cases} S'(u_L, f) + S'(u_R, f) + A(u_L, u_R, f) \\ 1 + S'(u_L, \bar{f}) + S'(u_R, \bar{f}) + A(u_L, u_R, \bar{f}). \end{cases} \tag{5.6}$$

The intuition here is that paying cost 1 to create a self-loop on node u creates (or removes) interactions, including self-loops, among all the descendants of u.

5.3.5 Modifications for directed graphs

Finally, the algorithm can be modified to handle evolutionary histories of directed graphs. For this, only the recurrence A need be modified. When computing $A'(u, v, f)$, a non-tree edge can be included from u to v, from v to u, both, or neither. Each of

these cases modifies the function f in a different way. Specifically:

$$A'(u,v,f) = \begin{cases} 0 + A'(u_L, v, f) + A'(u_R, v, f) \\ 1 + A'(u_L, v, \overleftarrow{f}) + A'(u_R, v, \overleftarrow{f}) \\ 1 + A'(u_L, v, \overrightarrow{f}) + A'(u_R, v, \overrightarrow{f}) \\ 2 + A'(u_L, v, \overleftrightarrow{f}) + A'(u_R, v, \overleftrightarrow{f}) \\ \vdots \end{cases},$$

where the vertical ellipsis indicates the symmetric cases involving v_L and v_R, and where $\overrightarrow{f}, \overleftarrow{f}, \overleftrightarrow{f}$ are defined, depending on u and v, as follows:

$$\overrightarrow{f}\,(x,y) = \begin{cases} 1 - f(x,y) & \text{if } x \in \mathrm{ST}(u) \text{and } y \in \mathrm{ST}(v) \\ f(x,y) & \text{otherwise,} \end{cases} \tag{5.7}$$

$$\overleftrightarrow{f}\,(x,y) = \begin{cases} 1 - f(x,y) & \text{if } x \in \mathrm{ST}(u) \text{and } y \in \mathrm{ST}(v) \text{or vice versa} \\ f(x,y) & \text{otherwise,} \end{cases} \tag{5.8}$$

with $\overleftarrow{f}$ defined analogously to $\overrightarrow{f}$. Here, $\mathrm{ST}(u)$ indicates the set of nodes in the subtree rooted at u.

The heuristic can be extended to handle different costs for edge addition and edge deletion by changing the constants in the recurrences to be functions dependent on f.

5.4 Results

5.4.1 Generating plausible simulated histories

We use a *degree-dependent model* (DDM) to simulate an evolutionary path from a putative ancestral network to its extant state. The model simulates node duplication, node deletion, independent edge gain, and independent edge loss with given probabilities $\mathrm{P_{ndup}}$, $\mathrm{P_{nloss}}$, $\mathrm{P_{egain}}$ and $\mathrm{P_{eloss}}$, respectively. The nodes or edges involved in a modification are chosen probabilistically based on their degrees (as in [94]) according to the following expressions:

$$\mathrm{P}(u \mid \text{node duplication}) \propto 1/k_u \qquad \mathrm{P}(u \mid \text{node loss}) \propto 1/k_u \tag{5.9}$$

$$\mathrm{P}((u,v) \mid \text{edge gain}) \propto k^o_u \qquad \mathrm{P}((u,v) \mid \text{edge loss}) \propto 1/k^o_u, \tag{5.10}$$

where k^o_u is the out-degree of a node u, and k_u is the total degree. At each time step, the distribution of possible modifications to the graph is calculated as $\mathrm{P}(\text{modification}) = \mathrm{P_{operation}P}(\text{object} \mid \text{operation})$. Nodes with out-degree of 0 are removed. We also consider a *degree-independent model* (DIM) in which the four conditional probabilities in Eqns. (5.9) and (5.10) are all equal.

This model is theoretically capable of producing evolutionary trajectories between any two networks while incorporating preferential attachment to the source node and random uniform choice of the target node. Furthermore, a node is chosen for duplication or loss in inverse proportion to its degree — a proxy for a selection effect based on relative impact of the event on the network.

Varying parameters $\mathrm{P_{ndup}}$, $\mathrm{P_{nloss}}$, $\mathrm{P_{egain}}$ and $\mathrm{P_{eloss}}$ can produce a wide variety of densities and sizes. For biologically plausible settings — those with high $\mathrm{P_{ndup}}$, low $\mathrm{P_{nloss}}$, and moderate to high $\mathrm{P_{egain}}$ and $\mathrm{P_{eloss}}$ — we observe that the constructed networks have an exponential in-degree distribution and scale-free out-degree distri-

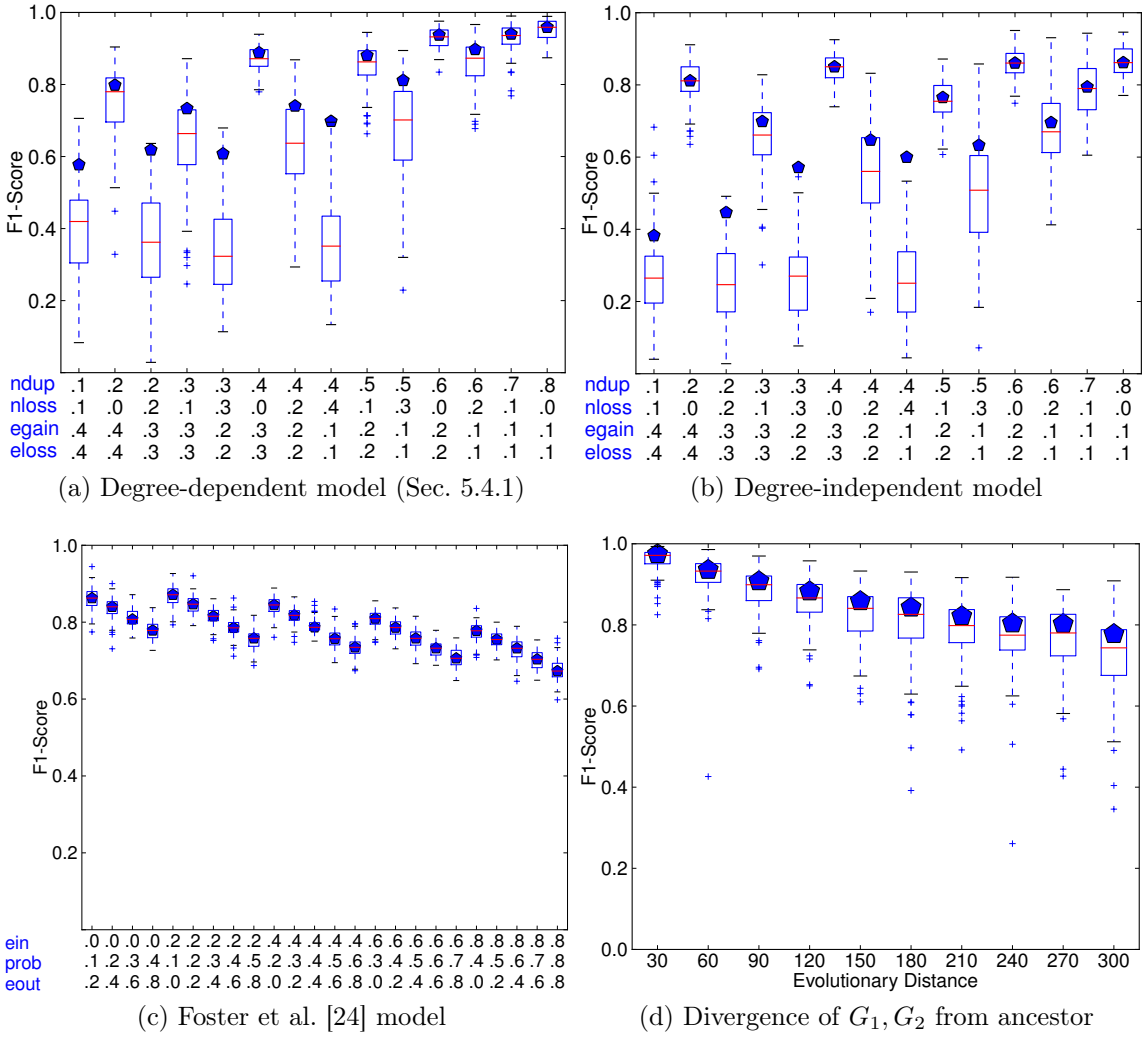
bution (exponent on average 1.79) [58]. This is in agreement with scale-free exponents of various real networks such as *S. cerevisiae* [47], *E. coli* [25] and *C. elegans* [16], which are between 1.73 and 1.99.

We also consider a regulatory evolution model by Foster et al. [24], which is based on gene duplication, with incoming and outgoing edges kept after duplication (P_{inkeep} and $P_{outkeep}$ probabilities respectively). New edges are added with probability $P_{innovation}$.

In all of the network evolution models, we started with a random connected seed graph that has 10 nodes and 25 edges. We evolved it to X by 200 operations where speciation happens and then both G_1 and G_2 evolve from X by additional 200 operations each. To ensure the ancestral graph was biologically reasonable, instances were kept only if the ancestral graph X had an in-degree that fit an exponential distribution with parameter between 1.0 and 1.2 or an out-degree that was scale-free with parameter between 1.8 and 2.2.

5.4.2 Reconstructing histories

Optimality of loop breaking. The greedy procedure to break blocking loops produces histories that are very close to optimal. We generated 1400 networks using the DDM model using the range of parameters on the x-axis of Fig. 5.3a. In the vast majority of cases (1325 out of 1400), either no loop breaking is required, or the solution discovered after greedily breaking all loops has the same cost as the original solution. In these cases, therefore, the method returned a provably maximally parsimonious set of edge modification events. In the remaining 75 cases (5.4%), greedily removing blocking loops increased the number of edge modifications by between 1 and 10 ($< 2\%$ of the initial number of edge modification events). Since the initial solution provides a lower bound on the optimal, we can verify that the greedy procedure always found a solution



(a) Degree-dependent model (Sec. 5.4.1)

(b) Degree-independent model

(c) Foster et al. [24] model

(d) Divergence of G_1, G_2 from ancestor

Figure 5.3: (a-c) Effect of model parameters on reconstruction accuracy under three different models. "Prob" in (c) is the probability of edge gain. (d) Effect of evolutionary distance (number of network modification operations) on the quality of the ancestral network reconstruction. In both plots, boxes show 1st and 3rd quartile over 100 networks with median indicated by a line. Pentagons show the median if edges incident to nodes lost in both lineages are not considered.

within 2% of the optimal (and perhaps even better). Thus, it seems that in practice, while blocking loops occur, the greedy procedure does a good job of eliminating them without increasing the number of events significantly.

Effect of growth model and its parameters. Modeling the evolutionary dynamics of a regulatory network is still an active topic of research. We therefore experiment with

three different network models (Sec. 5.4.1). Despite their differences, high precision and recall (measured as the F1 score) can be obtained for all of them for many choices of their parameters (Fig. 5.3a-c). Very good performance can be achieved under the general model presented above whether degree distributions are taken into account (Fig. 5.3a) or not (Fig. 5.3b) when selecting nodes and edges to modify. In these cases, for most parameter choices, precision is close to 1.0, meaning every edge predicted to be in the ancestor, in fact, was. Recall is often lower. The Foster et al. [24] model, with its heavy reliance on duplication events and lack of node loss events, tends to be the simplest under which to reconstruct the ancestral graph (Fig. 5.3c).

The largest factor leading to poorer performance is lower recall caused by gene losses. If all descendants of a gene are lost in both extant networks, it is not possible to reconstruct edges incident to it. If these edges are excluded from the computation of recall, the F1 score often improves dramatically. Median F1 scores are shown as pentagons in Fig. 5.3.

Robustness to evolutionary divergence. Naturally, the ability to recover the ancestral network degrades as time passes and the extant networks diverge. However, the degradation is slow (Fig. 5.3d, using the degree-dependent model with parameters fixed at $P_{ndup} = 0.35$, $P_{nloss} = 0.05$, $P_{egain} = 0.3$, and $P_{eloss} = 0.3$). When the distance is small, we are almost always able to recover the ancestral network well, as illustrated by the high F1-scores and small interquartile ranges in Figure 5.3d. Even when the distance between the ancestral and extant networks is large (300) compared to the average ancestral network size (55), we obtain an F1-score of 0.72 (0.77 when homology groups lost in both lineages are not considered).

5.5 Conclusion

We have presented a novel framework for representing network histories involving gene duplications, gene loss, and edge gain and loss for both directed and undirected graphs. A combinatorial characterization for valid histories was given. We have shown that a fast heuristic can recover optimal histories in a large majority of instances. We further provide evidence that, even with a probabilistic, weighted, generative model of network growth, a parsimony approach can recover accurate histories.

Chapter 6

Conclusion

Assembling genomes is more and more common, thanks to the reduction in sequencing cost and high-throughput technologies. The need for reliable and fast genome assembly software is ever more pressing, especially for large input size, such as mammalian genome size or larger. In the first part, we presented two projects in genome assembly which were motivated by my experience assembling two large genomes, the cow *Bos taurus* and the domestic turkey *Meleagris gallopavo*.

First, Jellyfish increases the speed and memory efficiency of counting k-mers, in particular in sequencing reads. This allows larger datasets to be handled in a reasonable time by properly using the multi-core architecture of current computers. Moreover, k-mer frequencies have many applications in biology beyond genome assembly.

Jellyfish suffers from a few limitations. Most importantly, the length k of the mers is currently limited to 31. Some assemblers are now using larger values for k (e.g. ALLPATHS-LG recommends using 96-mers with reads of length 150 bases). This is an implementation detail — nothing in Jellyfish's design prevents the use of longer keys. Future development of Jellyfish will lift that restriction.

In the second project, the Chromosome Builder explores ways to combine marker maps and mate pairs information to improve the accuracy of scaffolding. Accurate scaffolds and proper placement on the chromosome is important for the analysis of

genomes. For example, to find genes which are not contained in one contig, or to compare the evolution of the genomes of related species.

In the second part of this thesis, we considered different problems in bioinformatics. Namely the seeded complex detection problem and the ancestral network reconstruction problem. Both of these problems, and many others in genome assembly, are conveniently expressed in the language of graph theory. Understanding how the structure of natural networks appear and what this structure means for the organism could potentially answer many questions about evolution.

Algorithmic questions lead to the study of exactly k-edge-connected graphs. Although some subsets of this class of graphs (e.g. Harary graphs or k-connected k-regular graphs) have been studied before, we described exactly connected for the first time and gave a complete characterization for $k = 1, 2, 3$. Unsurprisingly, the characterization for $k = 3$ is much more complex than for $k = 1, 2$, and the characterization for larger values of k has so far remained elusive. Finding a general synthesis for all k, as it was done for k-regular graphs, is a possible extension of this work.

If the biological networks of extant species are not easily studied, the networks of ancestral species are, literally, not accessible. Nevertheless, knowing such an ancestral network and its precise evolution could be invaluable. On the other hand, it is possible to infer a "most likely" biological network of the ancestor of one, or many related, species. Our work on the reconstruction of network evolution finds the "most likely" ancestral network under the assumption of parsimony, that is the fewest number of events are preferred.

In practice, our algorithm found optimal or near optimal solutions in polynomial time, but the exact complexity class of this problem remains unknown. I postulate the problem of determining an optimal proper history (i.e. without any blocking loop) is NP-hard.

Bibliography

[1] M. Aldana, E. Balleza, S. Kauffman, and O. Resendiz. Robustness and evolvability in genetic regulatory networks. *J. Theor. Biol.*, 245(3):433–448, 2007.

[2] D. Barnette. A construction of 3-connected-graphs. *Israel Journal of Mathematics*, 86:397–407, 1994.

[3] David R Bentley. Whole-genome re-sequencing. *Current Opinion in Genetics & Development*, 16(6):545 – 552, 2006. Genomes and evolution.

[4] B. Bollobás. *Extremal Graph Theory*. Dover Publications, June 2004.

[5] E. Borenstein and M. W. Feldman. Topological signatures of species interactions in metabolic networks. *J. Comput. Biol.*, 16(2):191–200, 2009.

[6] E. Borenstein, M. Kupiec, M. W. Feldman, and E. Ruppin. Large-scale reconstruction and phylogenetic analysis of metabolic environments. *Proc. Natl. Acad. Sci. USA*, 105(38):14482–14487, 2008.

[7] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome Res*, 18(5):810–20, May 2008.

[8] Davide Campagna, Chiara Romualdi, et al. RAP: a new computer program for de novo identification of repeated sequences in whole genomes. *Bioinformatics*, 21(5):582–588, 2005.

[9] Susan Celniker, David Wheeler, Brent Kronmiller, Joseph Carlson, Aaron Halpern, Sandeep Patel, Mark Adams, Mark Champe, Shannon Dugan, Erwin Frise, Ann Hodgson, Reed George, Roger Hoskins, Todd Laverty, Donna Muzny, Catherine Nelson, Joanne Pacleb, Soo Park, Barret Pfeiffer, Stephen Richards, Erica Sodergren, Robert Svirskas, Paul Tabor, Kenneth Wan, Mark Stapleton, Granger Sutton, Craig Venter, George Weinstock, Steven Scherer, Eugene Myers, Richard Gibbs, and Gerald Rubin. Finishing a whole-genome shotgun: Release 3 of the drosophila melanogaster euchromatic genome sequence. *Genome Biology*, 3(12):research0079.1–0079.14, 2002.

[10] G. Chaty and M. Chein. Minimally 2-edge connected graphs. *Journal of Graph Theory*, 3:15–22, 1979.

[11] Chimpanzee Sequencing and Analysis Consortium. Initial sequence of the chimpanzee genome and comparison with the human genome. *Nature*, 437(7055):69–87, Sep 2005.

[12] Fan Chung, Linyuan Lu, T Gregory Dewey, and David J Galas. Duplication models for biological networks. *J. Comp. Biol.*, 10(5):677–687, January 2003.

[13] Rami A Dalloul, Julie A Long, Aleksey V Zimin, Luqman Aslam, Kathryn Beal, Le Ann Blomberg, Pascal Bouffard, David W Burt, Oswald Crasta, Richard P M A Crooijmans, Kristal Cooper, Roger A Coulombe, Supriyo De, Mary E Delany, Jerry B Dodgson, Jennifer J Dong, Clive Evans, Karin M Frederickson, Paul Flicek, Liliana Florea, Otto Folkerts, Martien A M Groenen, Tim T Harkins, Javier Herrero, Steve Hoffmann, Hendrik-Jan Megens, Andrew Jiang, Pieter de Jong, Pete Kaiser, Heebal Kim, Kyu-Won Kim, Sungwon Kim, David Langenberger, Mi-Kyung Lee, Taeheon Lee, Shrinivasrao Mane, Guillaume Marçais, Manja Marz, Audrey P McElroy, Thero Modise, Mikhail Nefedov, Cédric Notredame, Ian R Paton, William S Payne, Geo Pertea, Dennis Prickett, Daniela Puiu, Dan Qioa, Emanuele Raineri, Magali Ruffier, Steven L Salzberg, Michael C Schatz, Chantel Scheuring, Carl J Schmidt, Steven Schroeder, Stephen M J Searle, Edward J Smith, Jacqueline Smith, Tad S Sonstegard, Peter F Stadler, Hakim Tafer, Zhijian Jake Tu, Curtis P Van Tassell, Albert J Vilella, Kelly P Williams, James A Yorke, Liqing Zhang, Hong-Bin Zhang, Xiaojun Zhang, Yang Zhang, and Kent M Reed. Multi-platform next-generation sequencing of the domestic turkey (*Meleagris gallopavo*): genome assembly and analysis. *PLoS Biol*, 8(9), 2010.

[14] R. W. Dawes. Minimally 3-connected graphs. *Journal of combinatorial theory, Series B*, 40:159–168, 1986.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[16] B. Deplancke et al. A gene-centered *C. elegans* protein-DNA interaction network. *Cell*, 125(6):1193–1205, June 2006.

[17] Reinhard Diestel. *Graph Theory*. Springer-Verlag Heidelberg, New York, electronic edition 2005 edition, 2005.

[18] G. Ding and P. Chen. Generating r-regular graphs. *Discrete Applied mathematics*, 129:239–343, 2003.

[19] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[20] Janusz Dutkowski and Jerzy Tiuryn. Identification of functional modules from conserved ancestral protein-protein interactions. *Bioinformatics*, 23(13):i149–i158, 2007.

[21] Robert C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[22] Yoshimi Egawa, Katsumi Inoue, and Ken-ichi Kawarabayashi. Nonseparating induced cycles consisting of contractible edges in k-connected graphs. *Electronic Notes in Discrete Matematics*, 11:253–264, July 2002.

[23] J. Flannick, A. Novak, B. S. Srinivasan, H. H. McAdams, and S. Batzoglou. Graemlin: general and robust alignment of multiple large interaction networks. *Genome Res.*, 16(9):1169–1181, 2006.

[24] D. V. Foster, S. A. Kauffman, and J. E. S. Socolar. Network growth models and genetic regulatory networks. *Phys. Rev. E*, 73(3):031912, Mar 2006.

[25] Socorro Gama-Castro et al. RegulonDB (version 6.0): gene regulation model of escherichia coli k-12 beyond transcription, active (experimental) annotated promoters and textpresso navigation. *Nucleic Acids Research*, 36(suppl 1):D120–D124, 2008.

[26] H. Gao, J.F. Groote, and W.H. Hesselink. Almost wait-free resizable hashtables. In *Proceeding of the 18th International Parallel and Distributed Processing Symposium*, page 50a, April 2004.

[27] T. A. Gibson and Goldberg D. S. Reverse engineering the evolution of protein interaction networks. *Pac. Symp. Biocomput.*, pages 190–202, 2009.

[28] 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. *Proc Natl Acad Sci U S A*, 108(4):1513–8, Jan 2011.

[29] L. A. Goddyn and J. van den Heuvel. Removable circuits in multigraphs. *Journal of combinatorial theory, Series B*, 71:130–143, 1997.

[30] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications*. Discrete Mathematics and Its Applications. 2006.

[31] R. Halin. Studies on minimally n-connected graphs. In *Combinatorial mathematics and its applications*, pages 129–136, April 1967.

[32] R. Halin. A theorem on n-connected graphs. *Journal of Combinatorial Theory*, 7:150–154, 1969.

[33] Paul Havlak, Rui Chen, K. James Durbin, Amy Egan, Yanru Ren, Xing-Zhi Song, George M. Weinstock, and Richard A. Gibbs. The Atlas Genome Assembly System. *Genome Research*, 14(4):721–732, 2004.

[34] John Healy, Elizabeth E Thomas, Jacob T Schwartz, and Michael Wigler. Annotating large genomes with exact word matches. *Genome Res*, 13(10):2306–15, Oct 2003.

[35] D. A. Holton, B. Manvel, and B. D. McKay. Hamiltonian cycles in cubic 3-connected bipartite planar graphs. *Journal of combinatorial theory, Series B*, 38:279–297, 1985.

[36] I Ispolatov, P L Krapivsky, and A Yuryev. Duplication-divergence model of protein interaction network. *Phys. Rev. E*, 71(6 Pt 1):061911, June 2005.

[37] Takashi Ito, Tomoko Chiba, Ritsuko Ozawa, Mikio Yoshida, Masahira Hattori, and Yoshiyuki Sakaki. A comprehensive two-hybrid analysis to explore the yeast protein interactome. *Proceedings of the National Academy of Sciences*, 98(8):4569–4574, 2001.

[38] David B Jaffe, Jonathan Butler, et al. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome Res*, 13(1):91–6, Jan 2003.

[39] E. L. Johnson. A proof of four-coloring of the edges of a cubic graph. *American Math Monthly*, 73:52–55, 1966.

[40] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[41] J. Kececioglu and E. Myers. Combinatorial algorithms for dna sequence assembly. *Algorithmica*, 13:7–51, 1995. 10.1007/BF01188580.

[42] David R Kelley, Michael C Schatz, and Steven L Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol*, 11(11):R116, 2010.

[43] A. Kreimer, E. Borenstein, U. Gophna, and E. Ruppin. The evolution of modularity in bacterial metabolic networks. *Proc. Natl. Acad. Sci. USA*, 105(19):6976–6981, 2008.

[44] M. Kriesell. A constructive characterization of 3-connected triangle-free graphs. *Journal of combinatorial theory, Series B*, 97:358–370, 2007.

[45] Stefan Kurtz, Apurva Narechania, Joshua C Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9:517, 2008.

[46] Edya Ladan-mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. In *In Proceedings of the 18th International Symposium on Distributed Computing, LNCS 3274*, pages 117–131. Springer, 2004.

[47] Tong Ihn Lee et al. Transcriptional regulatory networks in saccharomyces cerevisiae. *Science*, 298(5594):799–804, 2002.

[48] A Lefebvre, T Lecroq, H Dauchel, and J Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–26, Feb 2003.

[49] E. D. Levy and J. B. Pereira-Leal. Evolution and dynamics of protein interactions and networks. *Curr. Opin. Struct. Biol.*, 18(3):349–357, 2008.

[50] Harris A Lewin, Denis M Larkin, Joan Pontius, and Stephen J. O'Brien. Every genome sequence needs a good map. *Genome Research*, 2009.

[51] Ruiqiang Li, Wei Fan, Geng Tian, et al. The sequence and de novo assembly of the giant panda genome. *Nature*, 463(7279):311–317, 01 2010.

[52] 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 Research*, 20(2):265–272, 2010.

[53] R. D. Lick. Critically and minimally n-connected graphs. *Lecture notes in mathematics: The Many Facets of Graph Theory*, 110:199–205, 1969.

[54] Yue Liu, Xiang Qin, Xing-Zhi Henry Song, Huaiyang Jiang, Yufeng Shen, K James Durbin, Sigbjørn Lien, Matthew Peter Kent, Marte Sodeland, Yanru Ren, Lan Zhang, Erica Sodergren, Paul Havlak, Kim C Worley, George M Weinstock, and Richard A Gibbs. Bos taurus genome assembly. *BMC Genomics*, 10:180, 2009.

[55] W. Mader. Minimale n-fach zusammenhangende graphen mit maximaler kantenzal. *Arch. Math.*, 23:219–224, 1972.

[56] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[57] 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. Godwin, 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 microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005. 10.1038/nature03959.

[58] Sergei Maslov and Kim Sneppen. Computational architecture of the yeast regulatory network. *Phys Biol*, 2(4):S94–100, 2005.

[59] Lakshmi K. Matukumalli, Cynthia T. Lawley, Robert D. Schnabel, Jeremy F. Taylor, Mark F. Allan, Michael P. Heaton, Jeff O'Connell, Stephen S. Moore,

Timothy P. L. Smith, Tad S. Sonstegard, and Curtis P. Van Tassell. Development and characterization of a high density snp genotyping assay for cattle. *PLoS ONE*, 4(4):e5350, 04 2009.

[60] D. Matula. k-components, cluster, and slicings in graphs. *SIAM J. Appl. Math.*, 22(3):459–480, 1972.

[61] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceeding of PODC '96*, 1996.

[62] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM.

[63] M. Middendorf, E. Ziv, and C. H. Wiggins. Inferring network mechanisms: the *Drosophila melanogaster* protein interaction network. *Proc. Natl. Acad. Sci. USA*, 102(9):3192–3197, 2005.

[64] 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):2818–24, Dec 2008.

[65] Jason R Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–27, Jun 2010.

[66] B. G. Mirkin, T. I. Fenner, M. Y. Galperin, and E. V. Koonin. Algorithms for computing parsimonious evolutionary scenarios for genome evolution, the last universal common ancestor and dominance of horizontal gene transfer in the evolution of prokaryotes. *BMC Evol. Biol.*, 3:2, 2003.

[67] A Mithani, GM Preston, and J Hein. A stochastic model for the evolution of metabolic networks with neighbor dependence. *Bioinformatics*, 25(12):1528–1535, 2009.

[68] Mouse Genome Sequencing Consortium, Robert H Waterston, Kerstin Lindblad-Toh, Ewan Birney, Jane Rogers, Josep F Abril, Pankaj Agarwal, Richa Agarwala, Rachel Ainscough, Marina Alexandersson, Peter An, Stylianos E Antonarakis, John Attwood, Robert Baertsch, Jonathon Bailey, Karen Barlow, Stephan Beck, Eric Berry, Bruce Birren, Toby Bloom, Peer Bork, Marc Botcherby, Nicolas Bray, Michael R Brent, Daniel G Brown, Stephen D Brown, Carol Bult, John Burton, Jonathan Butler, Robert D Campbell, Piero Carninci, Simon Cawley, Francesca Chiaromonte, Asif T Chinwalla, Deanna M Church, Michele Clamp, Christopher Clee, Francis S Collins, Lisa L Cook, Richard R Copley, Alan Coulson, Olivier Couronne, James Cuff, Val Curwen, Tim Cutts, Mark Daly, Robert David, Joy Davies, Kimberly D Delehaunty, Justin Deri, Emmanouil T Dermitzakis, Colin Dewey, Nicholas J Dickens, Mark Diekhans,

Sheila Dodge, Inna Dubchak, Diane M Dunn, Sean R Eddy, Laura Elnitski, Richard D Emes, Pallavi Eswara, Eduardo Eyras, Adam Felsenfeld, Ginger A Fewell, Paul Flicek, Karen Foley, Wayne N Frankel, Lucinda A Fulton, Robert S Fulton, Terrence S Furey, Diane Gage, Richard A Gibbs, Gustavo Glusman, Sante Gnerre, Nick Goldman, Leo Goodstadt, Darren Grafham, Tina A Graves, Eric D Green, Simon Gregory, Roderic Guigó, Mark Guyer, Ross C Hardison, David Haussler, Yoshihide Hayashizaki, LaDeana W Hillier, Angela Hinrichs, Wratko Hlavina, Timothy Holzer, Fan Hsu, Axin Hua, Tim Hubbard, Adrienne Hunt, Ian Jackson, David B Jaffe, L Steven Johnson, Matthew Jones, Thomas A Jones, Ann Joy, Michael Kamal, Elinor K Karlsson, Donna Karolchik, Arkadiusz Kasprzyk, Jun Kawai, Evan Keibler, Cristyn Kells, W James Kent, Andrew Kirby, Diana L Kolbe, Ian Korf, Raju S Kucherlapati, Edward J Kulbokas, David Kulp, Tom Landers, J P Leger, Steven Leonard, Ivica Letunic, Rosie Levine, Jia Li, Ming Li, Christine Lloyd, Susan Lucas, Bin Ma, Donna R Maglott, Elaine R Mardis, Lucy Matthews, Evan Mauceli, John H Mayer, Megan McCarthy, W Richard McCombie, Stuart McLaren, Kirsten McLay, John D McPherson, Jim Meldrim, Beverley Meredith, Jill P Mesirov, Webb Miller, Tracie L Miner, Emmanuel Mongin, Kate T Montgomery, Michael Morgan, Richard Mott, James C Mullikin, Donna M Muzny, William E Nash, Joanne O Nelson, Michael N Nhan, Robert Nicol, Zemin Ning, Chad Nusbaum, Michael J O'Connor, Yasushi Okazaki, Karen Oliver, Emma Overton-Larty, Lior Pachter, Genís Parra, Kymberlie H Pepin, Jane Peterson, Pavel Pevzner, Robert Plumb, Craig S Pohl, Alex Poliakov, Tracy C Ponce, Chris P Ponting, Simon Potter, Michael Quail, Alexandre Reymond, Bruce A Roe, Krishna M Roskin, Edward M Rubin, Alistair G Rust, Ralph Santos, Victor Sapojnikov, Brian Schultz, Jörg Schultz, Matthias S Schwartz, Scott Schwartz, Carol Scott, Steven Seaman, Steve Searle, Ted Sharpe, Andrew Sheridan, Ratna Shownkeen, Sarah Sims, Jonathan B Singer, Guy Slater, Arian Smit, Douglas R Smith, Brian Spencer, Arne Stabenau, Nicole Stange-Thomann, Charles Sugnet, Mikita Suyama, Glenn Tesler, Johanna Thompson, David Torrents, Evanne Trevaskis, John Tromp, Catherine Ucla, Abel Ureta-Vidal, Jade P Vinson, Andrew C Von Niederhausern, Claire M Wade, Melanie Wall, Ryan J Weber, Robert B Weiss, Michael C Wendl, Anthony P West, Kris Wetterstrand, Raymond Wheeler, Simon Whelan, Jamey Wierzbowski, David Willey, Sophie Williams, Richard K Wilson, Eitan Winter, Kim C Worley, Dudley Wyman, Shan Yang, Shiaw-Pyng Yang, Evgeny M Zdobnov, Michael C Zody, and Eric S Lander. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420(6915):520–62, Dec 2002.

[69] E W Myers, G G Sutton, A L Delcher, et al. A whole-genome assembly of Drosophila. *Science*, 287(5461):2196–204, Mar 2000.

[70] S. Navlakha and C. Kingsford. Network archaeology: Uncovering ancient networks from present-day interactions. *PLoS Comput. Biol.*, 7(4):e1001119, 2011.

[71] Oystein Ore. *The Four-Color Problem*, volume 27. Academic Press Inc., 1967.

[72] L. Pachter. An introduction to reconstructing ancestral genomes. In *Proc. Symp. in Applied Mathematics*, volume 64, pages 1–20, 2007.

[73] Philipp Pagel, Stefan Kovac, Matthias Oesterheld, Barbara Brauner, Irmtraud Dunger-Kaltenbach, Goar Frishman, Corinna Montrone, Pekka Mark, Volker Stümpflen, Hans-Werner Mewes, Andreas Ruepp, and Dmitrij Frishman. The mips mammalian protein–protein interaction database. *Bioinformatics*, 21(6):832–834, 2005.

[74] Romualdo Pastor-Satorras, Eric Smith, and Ricard Sole. Evolving protein interaction networks from gene duplication. *J. Theor. Biol.*, 222:199–210, 2003.

[75] J. B. Pereira-Leal, E. D. Levy, C. Kamp, and S. A. Teichmann. Evolution of protein complexes by duplication of homomeric interactions. *Genome Biol.*, 8(4):R51, 2007.

[76] P A Pevzner, H Tang, and M S Waterman. An eulerian path approach to dna fragment assembly. *Proc Natl Acad Sci U S A*, 98(17):9748–53, Aug 2001.

[77] John W. Pinney, Grigoris D. Amoutzias, Magnus Rattray, and David L. Robertson. Reconstruction of ancestral protein interaction networks for the bZIP transcription factors. *Proc. Natl. Acad. Sci. USA*, 104(51):20449–20453, 2007.

[78] Michael D. Plummer. On minimal blocks. *Transactions of the American Mathematical Society*, 134(1):85–94, 1968.

[79] M. Pop, D. S. Kosack, and S. L. Salzberg. Hierarchical scaffolding with bambus. *Genome Research*, 14:149–159, 2004.

[80] Mihai Pop. Genome assembly reborn: recent computational challenges. *Brief Bioinform*, 10(4):354–66, Jul 2009.

[81] C. Purcell and T. Harris. Non-blocking hashtables with open addressing. Technical Report 639, University of Cambridge, September 2005.

[82] Dana Randall. Efficient generation of random nonsingular matrices. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1991.

[83] Michael Roberts, Brian R Hunt, James A Yorke, Randall A Bolanos, and Arthur L Delcher. A preprocessor for shotgun assembly of large genomes. *J Comput Biol*, 11(4):734–52, 2004.

[84] Gabriel Robins and Alexander Zelikovsky. Improved steiner tree approximation in graphs. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, SODA '00, pages 770–779, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[85] Steven L. Salzberg and James A. Yorke. Beware of mis-assembled genomes. *Bioinformatics*, 21(24):4320–4321, 2005.

[86] Michael C. Schatz, Arthur L. Delcher, and Steven L. Salzberg. Assembly of large genomes using second-generation sequencing. *Genome Res*, (9):1165–73, Sep 2010.

[87] H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, MIT, 1954.

[88] O. Shalev and N. Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, 2006.

[89] H Shizuya, B Birren, U J Kim, V Mancino, T Slepak, Y Tachiiri, and M Simon. Cloning and stable maintenance of 300-kilobase-pair fragments of human dna in escherichia coli using an f-factor-based vector. *Proc Natl Acad Sci U S A*, 89(18):8794–7, Sep 1992.

[90] Suzanne S. Sindi, Brian R. Hunt, and James A. Yorke. Duplication count distributions in DNA sequences. *Phys. Rev. E*, 78(6):061912, Dec 2008.

[91] Rohit Singh, Jinbo Xu, and Bonnie Berger. Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Proc. Intl. Conf. on Research in Computational Molecular Biology (RECOMB)*, pages 16–31, 2007.

[92] Warren Snelling, Readman Chiu, Jacqueline Schein, Matthew Hobbs, Colette Abbey, David Adelson, Jan Aerts, Gary Bennett, Ian Bosdet, Mekki Boussaha, Rudiger Brauning, Alexandre Caetano, Marcos Costa, Allan Crawford, Brian Dalrymple, Andre Eggen, Annelie Everts-van der Wind, Sandrine Floriot, Mathieu Gautier, Clare Gill, Ronnie Green, Robert Holt, Oliver Jann, Steven Jones, Steven Kappes, John Keele, Pieter de Jong, Denis Larkin, Harris Lewin, and John McEwan. A physical map of the bovine genome. *Genome Biology*, 8(8):R165, 2007.

[93] E. Steinitz and H. Rademacher. *Vorlesungen über die Theorie de Polyeder unter Einschluss der Elemente der Topologie.* Springer, Berlin, 1934.

[94] A. J. Stewart, R. M. Seymour, and A. Pomiankowski. Degree dependence in rates of transcription factor evolution explains the unusual structure of transcription networks. *Proc. Biol. Sci.*, 276(1666):2493–2501, 2009.

[95] R. Rivest T. Cormen, C. Leiserson. *Introduction to algorithms*, chapter 12. MIT Press, 1990.

[96] Sarah A Teichmann and M Madan Babu. Gene regulatory network growth by duplication. *Nat. Genetics*, 36(5):492–6, May 2004.

[97] Carsten Thomassen. Non-separating cycles in k-connected graphs. *Journal of Graph Theory*, 5:351–354, 1981.

[98] Carsten Thomassen and Bjarne Toft. Non-separating induced cycles in graphs. *Journal of combinatorial theory, Series B*, 31:199–224, 1981.

[99] W.T. Tutte. *Connectivity in Graphs*. Number 15 in Mathematical Expositions. University of Toronto Press, 1966.

[100] Peter Uetz, Loic Giot, Gerard Cagney, Traci A. Mansfield, Richard S. Judson, James R. Knight, Daniel Lockshon, Vaibhav Narayan, Maithreyan Srinivasan, Pascale Pochart, Alia Qureshi-Emili, Ying Li, Brian Godwin, Diana Conover, Theodore Kalbfleisch, Govindan Vijayadamodar, Meijia Yang, Mark Johnston, Stanley Fields, and Jonathan M. Rothberg. A comprehensive analysis of protein-protein interactions in saccharomyces cerevisiae. *Nature*, 403(6770):623–627, 2000. 10.1038/35001009.

[101] J. Craig Venter, Mark D. Adams, Eugene W. Myers, Peter W. Li, Richard J. Mural, Granger G. Sutton, Hamilton O. Smith, Mark Yandell, Cheryl A. Evans, Robert A. Holt, Jeannine D. Gocayne, Peter Amanatides, Richard M. Ballew, Daniel H. Huson, Jennifer Russo Wortman, Qing Zhang, Chinnappa D. Kodira, Xiangqun H. Zheng, Lin Chen, Marian Skupski, Gangadharan Subramanian, Paul D. Thomas, Jinghui Zhang, George L. Gabor Miklos, Catherine Nelson, Samuel Broder, Andrew G. Clark, Joe Nadeau, Victor A. McKusick, Norton Zinder, Arnold J. Levine, Richard J. Roberts, Mel Simon, Carolyn Slayman, Michael Hunkapiller, Randall Bolanos, Arthur Delcher, Ian Dew, Daniel Fasulo, Michael Flanigan, Liliana Florea, Aaron Halpern, Sridhar Hannenhalli, Saul Kravitz, Samuel Levy, Clark Mobarry, Knut Reinert, Karin Remington, Jane Abu-Threideh, Ellen Beasley, Kendra Biddick, Vivien Bonazzi, Rhonda Brandon, Michele Cargill, Ishwar Chandramouliswaran, Rosane Charlab, Kabir Chaturvedi, Zuoming Deng, Valentina Di Francesco, Patrick Dunn, Karen Eilbeck, Carlos Evangelista, Andrei E. Gabrielian, Weiniu Gan, Wangmao Ge, Fangcheng Gong, Zhiping Gu, Ping Guan, Thomas J. Heiman, Maureen E. Higgins, Rui-Ru Ji, Zhaoxi Ke, Karen A. Ketchum, Zhongwu Lai, Yiding Lei, Zhenya Li, Jiayin Li, Yong Liang, Xiaoying Lin, Fu Lu, Gennady V. Merkulov, Natalia Milshina, Helen M. Moore, Ashwinikumar K Naik, Vaibhav A. Narayan, Beena Neelam, Deborah Nusskern, Douglas B. Rusch, Steven Salzberg, Wei Shao, Bixiong Shue, Jingtao Sun, Zhen Yuan Wang, Aihui Wang, Xin Wang, Jian Wang, Ming-Hui Wei, Ron Wides, Chunlin Xiao, Chunhua Yan, Alison Yao, Jane Ye, Ming Zhan, Weiqing Zhang, Hongyu Zhang, Qi Zhao, Liansheng Zheng, Fei Zhong, Wenyan Zhong, Shiaoping C. Zhu, Shaying Zhao, Dennis Gilbert, Suzanna Baumhueter, Gene Spier, Christine Carter, Anibal Cravchik, Trevor Woodage, Feroze Ali, Huijin An, Aderonke Awe, Danita Baldwin, Holly Baden, Mary Barnstead, Ian Barrow, Karen Beeson, Dana Busam, Amy Carver, Angela Center, Ming Lai Cheng, Liz Curry, Steve Danaher, Lionel Davenport, Raymond Desilets, Susanne Dietz, Kristina Dodson, Lisa Doup, Steven Ferriera, Neha Garg, Andres Gluecksmann, Brit Hart, Jason Haynes, Charles Haynes, Cheryl Heiner, Suzanne Hladun, Damon Hostin, Jarrett Houck, Timothy Howland, Chinyere Ibegwam, Jeffery Johnson, Francis Kalush, Lesley

Kline, Shashi Koduru, Amy Love, Felecia Mann, David May, Steven McCawley, Tina McIntosh, Ivy McMullen, Mee Moy, Linda Moy, Brian Murphy, Keith Nelson, Cynthia Pfannkoch, Eric Pratts, Vinita Puri, Hina Qureshi, Matthew Reardon, Robert Rodriguez, Yu-Hui Rogers, Deanna Romblad, Bob Ruhfel, Richard Scott, Cynthia Sitter, Michelle Smallwood, Erin Stewart, Renee Strong, Ellen Suh, Reginald Thomas, Ni Ni Tint, Sukyee Tse, Claire Vech, Gary Wang, Jeremy Wetter, Sherita Williams, Monica Williams, Sandra Windsor, Emily Winn-Deen, Keriellen Wolfe, Jayshree Zaveri, Karena Zaveri, Josep F. Abril, Roderic Guigó, Michael J. Campbell, Kimmen V. Sjolander, Brian Karlak, Anish Kejariwal, Huaiyu Mi, Betty Lazareva, Thomas Hatton, Apurva Narechania, Karen Diemer, Anushya Muruganujan, Nan Guo, Shinji Sato, Vineet Bafna, Sorin Istrail, Ross Lippert, Russell Schwartz, Brian Walenz, Shibu Yooseph, David Allen, Anand Basu, James Baxendale, Louis Blick, Marcelo Caminha, John Carnes-Stine, Parris Caulk, Yen-Hui Chiang, My Coyne, Carl Dahlke, Anne Deslattes Mays, Maria Dombroski, Michael Donnelly, Dale Ely, Shiva Esparham, Carl Fosler, Harold Gire, Stephen Glanowski, Kenneth Glasser, Anna Glodek, Mark Gorokhov, Ken Graham, Barry Gropman, Michael Harris, Jeremy Heil, Scott Henderson, Jeffrey Hoover, Donald Jennings, Catherine Jordan, James Jordan, John Kasha, Leonid Kagan, Cheryl Kraft, Alexander Levitsky, Mark Lewis, Xiangjun Liu, John Lopez, Daniel Ma, William Majoros, Joe McDaniel, Sean Murphy, Matthew Newman, Trung Nguyen, Ngoc Nguyen, Marc Nodell, Sue Pan, Jim Peck, Marshall Peterson, William Rowe, Robert Sanders, John Scott, Michael Simpson, Thomas Smith, Arlan Sprague, Timothy Stockwell, Russell Turner, Eli Venter, Mei Wang, Meiyuan Wen, David Wu, Mitchell Wu, Ashley Xia, Ali Zandieh, and Xiaohong Zhu. The Sequence of the Human Genome. *Science*, 291(5507):1304–1351, 2001.

[102] A. Wagner. Evolution of gene networks by gene duplications: A mathematical model and its implications on genome organization. *Proceedings of the National Academy of Sciences*, 91(10):4387–4391, May 1994.

[103] Xiuwei Zhang and Bernard Moret. Refining transcriptional regulatory networks using network evolutionary models and gene histories. *Alg. Mol. Biol.*, 5(1):1, 2010.

[104] Xiuwei Zhang and Bernard M. Moret. Boosting the performance of inference algorithms for transcriptional regulatory networks using a phylogenetic approach. In *Proc. Intl. Workshop on Algorithms in Bioinformatics (WABI)*, pages 245–258, 2008.

[105] Aleksey V Zimin, Arthur L Delcher, Liliana Florea, David R Kelley, Michael C Schatz, Daniela Puiu, Finnian Hanrahan, Geo Pertea, Curtis P Van Tassell, Tad S Sonstegard, Guillaume Marçais, Michael Roberts, Poorani Subramanian, James A Yorke, and Steven L Salzberg. A whole-genome assembly of the domestic cow, bos taurus. *Genome Biol*, 10(4):R42, 2009.