Biological Applications for de Bruijn Subgraphs and Interval Group Testing

José Augusto Amgarten Quitzau

June 2009

Dissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
(Doctor rerum naturalium)

an der Technischen Fakultät
der Universität Bielefeld

Betreuer:
Prof. Dr. rer. nat. Jens Stoye
Prof. Dr. Ferdinando Cicalese

Thesis defended on Thursday, March 25th, 2010.

Committee:
Prof. Dr. Hermann Ragg (chair),
Dr. Marília Dias Vieira Braga,
Prof. Dr. Jens Stoye (referee),
Prof. Dr. Ferdinando Cicalese (referee).

Gedruckt auf alterungsbeständigem Papier ISO 9706

To my parents and sisters

Preface

As the title suggest, this thesis is divided in two distinct and (almost) unrelated parts. The first, more practical, focuses on biological applications for de Bruijn subgraphs, while the second, more theoretical, presents a study on Interval Group Testing in the presence of erroneous test outcomes. Although the subjects of both parts are from the computational viewpoint completely distinct, their motivations are closely related: while the study of de Bruijn subgraphs was motivated by the identification of unique sequences for a better design of DNA probes, the study of the effect of errors in Interval Group Testing was motivated by the error produced by badly designed probes.

De Bruijn subgraphs have been used now for 10 years for the analysis of genome data, but never experienced big popularity in the field. The main problem with these graphs is the amount of memory needed to store them, since they have one vertex representing each substring of a given length found in the genome. Although almost all applications using de Bruijn graphs transform it in a more compact structure before starting any analyses, methods for directly constructing the compact representation are not found in the literature.

In the first part of this thesis, we present a method for constructing compact representations of de Bruijn subgraphs without passing through the memory expensive step of constructing the graph in its traditional form. We further analyze the use of this compact representation in three different applications: marking repeated sequences in a set of reads, identifying new repeat families in incompletely sequenced genomes, and creating splicing graphs for a collection of transcripts.

In Chapter 1 we give an introduction to graphs, de Bruijn graphs and sequences, and biological applications for de Bruijn subgraphs. In Chapters 2 we present a method for constructing a compact representation for de Bruijn graphs and comment how this representation can be used for marking repetitive sequences in reads. Chapters 3 and

4 are dedicated to a repeat family identification method. Finally Chapter 5 presents a method for clustering transcripts with a common origin in splicing graphs.

The second part of this text is dedicated to Interval Group Testing. Group testing is an approach for reducing the number of tests needed for identifying few elements with some rare property in a large group. An overview on group testing and its application in Genome Research is given in Chapter 6. Group testing finds a lot of applications in many different fields, for instance, in identifying people infected with HIV or syphilis in large populations, finding faulty unities in computer networks, or finding points in mature RNAs where introns were spliced out. The last application motivates a variant called *Interval Group Testing*, which is presented in Chapter 7. The traditional Interval Group Testing cannot deal with experimental errors, which cannot be ignored in a real life application. Therefore in Chapter 8 we present bounds on the number of tests needed for identifying the elements of interest when an upper bound on the number of errors in the tests' outcomes is allowed.

Contents

Part I

De Bruijn Subgraphs

Chapter 1

Introduction

Maybe no academic work should start with such an imprecise information, but I once read somewhere that the houses in Stockholm have no key. At least not this metal piece we turn once or twice to lock our doors. Instead, the houses have a numeric keypad. To open a door in Stockholm, we need a 4-digit number. I do not know whether it is true, but I though about the consequences of such a system. I lived in a relatively big city in southeast Brazil. A city with more than 1,000,000 inhabitants. And the idea that one of the 10,000 possible codes is surely shared by more than 100 people scared me a lot.

Of course there is no reason to be too afraid of having unwanted visitors in our houses, for the robbers still need to discover which key matches our house's lock. And trying all possible codes in sequence is hard work: a naive robber with bad luck would need to press 40,000 digits to open a door. Suppose the robber first tries the combination "1685" and then "1750". In total, he needs to press the 8 keys "16851750". However, since the system has no `enter` key, every digit pressed after the third digit corresponds to a new try. This means that not only the numbers "1685" and "1750" are tested, but also all sequences of four consecutive digits in between: "6851", "8517", and "5175". A smarter robber could think about choosing a sequence of keys where every digit pressed after third digit corresponds to a code not tried before. Since we have 10,000 different combinations, the shortest sequence able to try every single combination could not be shorter than 10,003 digits, which is much better than the previous 40,000. And for the smart bandit, it would be interesting to know if such a sequence exists.

Because of a weird coincidence, a robber with a secret passion for medieval Indian poetry would have an advantage in finding a sequence with minimum length. His inspiration could come from the Sanskrit word:

yamátárájabhánasalagám.

What does it mean? Absolutely nothing. But this word was used by medieval Indian poets as a memory hook to remember all possible sequences of 3 syllables alternating long (accented) and short (unaccented) ones (41, Chapter 8).

The principle is simple: every sequence of 3 syllables appears somewhere in the sequence of vowels *aááááaáaaaá* and can be remembered while someone speaks the word. Computer Scientists may feel more comfortable exchanging *a* for 0 and *á* for 1. A simple inspection shows that the resulting string, 0111010001, contains the 3-digit binary representation for all numbers from 0 to 7 exactly once, and is therefore minimal. Notice that this is exactly how the 10,000 codes should be arranged so that we get an optimal sequence of 10,003 digits.

The robber of Stockholm needs to solve a more complex instance of the same problem. Instead of encoding all 3-digit strings of an alphabet with 2 symbols (*a* and *á*), he needs to encode all 4-digit strings of an alphabet with 10 symbols in a single word. One of the first details we notice while playing with this problem is that the minimum possible length is only reached when every two neighbor codes share 3 digits. More specific, in order to achieve the minimum length, the last 3 digits of the last tried code must correspond to the first 3 digits of the next code, so that one new code is tried at each newly pressed key.

In order to systematically solve the problem, one could think about putting all the 10,000 different codes in a diagram and connect each code to all the possible next ones. For instance, in such a diagram, the codes to be tried after "2007" should be "0070", "0071", ..., and "0079". The structure of this diagram has the following implication: if a word containing all codes as substring has minimal length, then the order of the codes in the sequence describes a trail in this diagram where every code is visited exactly once. The opposite is also true: if a robber is able to find such a trail in this diagram, then he is also able to find a word that minimizes the number of tries to open a door.

In fact, in 1946, N. G. de Bruijn proved that such a word always exists by showing that such a trail always exist. The diagram is thus called *de Bruijn* graph, and the sequence *de Bruijn Sequence*. Although de Bruijn is believed to be the first who systematically studied these graphs, no evidence has been adduced so far. Various descriptions from many different areas can be found in the literature since the end of the 19th century. An elaborate overview is presented in (41).

1.1 Graphs and Strings

In this section we introduce in a simple, and sometimes informal, way the basic knowledge about graphs and strings needed to understanding the first part of this text.

1.1.1 Graphs

A *graph* is a mathematical object defined by two sets V and E, where V is an arbitrary set and E is a set of pairs of elements in V. The elements in V are called the *vertices* of the graph, whereas the pairs in E are called the *edges*. A graph, usually denoted by G, is completely described by these two sets. Often the vertices and edges are graphically represented in a diagram with dots and lines, where dots represent the vertices and the edges are represented by lines connecting the corresponding vertices.

In many applications E is a *multiset*, which means that some pairs may occur more than once in E. In this case, each edge $e \in E$ has an associated *multiplicity*, which corresponds to the number of times the pair e occurs in the multiset E.

Imagine that vertices are geographic places and the edges are ways connecting these places. Any sequence of vertices visited by simply choosing a way (edge) on each place (vertex) and going to the next one is called a *walk* in the graph. A walk that never takes the same way (edge) twice is called a *tour*. If the tour never passes on the same place (vertex) twice, it is called a *path*. Formally, let $v_1, v_2, \ldots, v_\eta$ be vertices in V. If for each v_i, $1 \leq i < \eta$, there is an edge $\{v_i, v_{i+1}\} \in E$, the sequence of vertices $v_1, v_2, \ldots, v_\eta$ is called a walk in the graph. If for each v_i and v_j, $1 \leq i < j < \eta$, $\{v_i, v_{i+1}\} \neq \{v_j, v_{j+1}\}$, the sequence of sets is called a tour in the graph. Finally, if the sequence of vertices $v_1, v_2, \ldots, v_\eta$ is a walk where for each $1 \leq i < j \leq \eta$, $v_i \neq v_j$, the sequence is called a path in the graph.

Many practical applications involve finding tours and paths with special properties in graphs. The first documented practical application of graph theory concerned the problem of deciding whether there is a tour through the city of Königsberg starting on a giving place, passing exactly once on each bridge in the city, and ending on the same place. Leonhard Euler modeled the city as a graph with four places as vertices: North, south, east, and island; and the bridges connecting these places: two connecting the island to the south, two the island to the north, and three connecting the east to each of the other three points, in a total of seven bridges, as shown in Figure 1.1. Euler proved that there is no way to do such a tour in Königsberg, disappointing all people who spent their free time trying to find them. Other graphs, like the de Bruijn graphs, do have such tours, which are called *Eulerian tours*, in honor of the first mathematician who studied them. A graph having Eulerian tours is called an *Eulerian graph.*

Another interesting kind of walk is actually a cycle, which is a path that ends in the same point where it starts. The difference to Eulerian tours is that here we are interested in passing in all vertices (not edges) exactly once. It is easy to see in Figure 1.1 that such a cycle exists in Königsberg: for instance, starting at east, go from east to south, than to the island, and come back passing through the north. Cycles with this property are called *Hamiltonian cycles*, in honor of William Rowan Hamilton, who solved the problem of finding such a cycle in a graph that has the shape of a dodecahedron. Graphs with Hamiltonian cycles are called *Hamiltonian graphs.*

Although the definitions of Eulerian tours and Hamiltonian cycles are very similar, since they only differ in the elements which need to be visited: edges, in Eulerian tours, and vertices, in Hamiltonian cycles, finding Hamiltonian cycles is much more complex than finding Eulerian tours. In fact, while Eulerian tours can be easily found by extending a walk until it passes through all the edges, it is proved that there is no better way of finding Hamiltonian cycles than trying any possible way to pass through all the vertices.

Even though Hamiltonian cycles are hard to find, there are cases where Eulerian cycles in a graph can be used to find Hamiltonian cycles in another. Consider a graph G with vertex set V and edge multiset E, $G = (V, E)$. We define the *line graph* of G, $L(G) = (V_L, E_L)$, as follows:

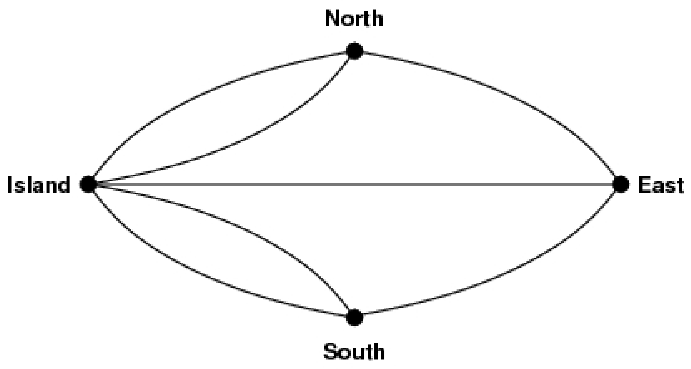

Figure 1.1: Graph representing the bridges of Königsberg. The city quarters, represented by the set of vertices $V = \{\text{North}, \text{South}, \text{East}, \text{Island}\}$, are connected by seven bridges, represented by the multiset of edges $E = \{\{\text{North, Island}\}, \{\text{North, Island}\}, \{\text{South, Island}\}, \{\text{South, Island}\}, \{\text{North, East}\}, \{\text{Island, East}\}, \{\text{South, East}\}\}$.

$$\begin{aligned} V_L &= E, \\ E_L &= \{\{\{u,v\},\{v,w\}\} \mid u,v,w \in V\}. \end{aligned}$$

The graph $L(G)$ is therefore the graph that has the edges of G as **vertices** and two vertices are connected if and only if the edges they represent have a common vertex in G. Notice that every walk on G that passes through at least three vertices can be mapped in a walk on $L(G)$, since every group of three consecutive vertices in the walk in G correspond to an edge in $L(G)$. Since an Eulerian cycle in G passes visits all edges in G, the corresponding cycle in $L(G)$ passes through all the vertices of the line graph. Therefore there is a direct correspondence between Hamiltonian cycles in $L(G)$ and Eulerian cycles in G.

In graphs like the one shown in Figure 1.1 the order of the vertices in an edge is not important: in the Königsberg example, if north and south are connected by a bridge, one can go both from north to south and in the opposite direction through the same bridge. We can think about modeling the case of one-way bridges and say that the edges {North, South} and {South, North} are distinct, where the first edge models the case when it is possible to go from north to south, and the second models the opposite direction. These graphs are called *directed graphs*, and the diagram proposed for organizing the codes to be tried by a robber in Stockholm is an example of them:

while the pair of codes $\{1234, 2345\}$ can be efficiently tried with only 5 digits, the sequence $\{2345, 1234\}$ needs 8 digits.

Edges in directed graphs are sometimes called *arcs*, and their multiset can be denoted by A to reinforce the fact that the graph is directed. In an arc $\{u, v\}$, the vertex u is called the *predecessor* of v and the vertex v the *successor* of u. The number of arcs connecting a vertex to its predecessors is called the vertex *indegree*, while the *outdegree* of a vertex is the number of arcs connecting it to its successors. If a vertex has no predecessor, it is called a *source*, while a vertex with no successor is called a *sink*. Definitions like walk, tours, and paths can be easily applied to directed graphs with the additional restriction that consecutive pairs in walks must respect the order of the vertices in the arcs. Also Eulerian tours and Hamiltonian cycles can be defined on directed graphs.

1.1.2 Alphabets and Strings

An alphabet is a finite set of symbols usually denoted by Σ. Each alphabet has a length $\sigma = |\Sigma|$, which corresponds to the number of symbols in the alphabet. The symbols of an alphabet, also called *characters*, can be used to construct sequences called *strings*[1]. Strings also have a length, which is the number of symbols they contain, counted with repetitions. An *empty* string is a string with no characters and is usually denoted by ε. Strings are sometimes called *tuples*. An l-tuple is a string of length l. The set of all strings of a given length l over an alphabet Σ is denoted by Σ^l, which is also the set of all l-tuples over Σ.

To give some examples, consider the alphabet of accented and unaccented vowels $\Sigma =$ {*a*, *á*} used in the introduction of this part. The sequence of vowels $s =$ *aáááaáaaaá* is a string of length $|s| = 10$ over this alphabet. Moreover, the set of all strings of length 3 on this alphabet is $\Sigma^3 =$ {*aáá*, *ááá*, *ááa*, *áaá*, *aáa*, *áaa*, *aaa*, *aaá*, }.

The characters in a string are numbered from 1, the leftmost character, to the length of the string. Given a string s, we denote by $s[i, j]$, $1 \leq i \leq j \leq |s|$ the string formed by the characters of s from the character i, inclusive, to the character j, inclusive.

[1]In this work we use the terms *sequence* and *string* as synonyms. Although some authors establish clear differences between strings and sequences, specially in case of substrings and subsequences, these differences are not explored in this work.

Each string $s[i, j]$ is called a *substring* of s. In the previous example, $s[1, 3] = a\acute{a}\acute{a}$ is a substring of $s = a\acute{a}\acute{a}\acute{a}a\acute{a}aaa\acute{a}$. Notice that in the previous example the set Σ^3 is the set of all substrings of s of length 3, which makes of s a de Bruijn sequence, as we will see in Section 1.2.

The set of all substrings of a given length l of a string is called its *l-dimensional spectrum*, or simply *spectrum* when the value of l is clear. For instance, if $\Sigma = \{a, \acute{a}\}$, Σ^3 is the 3-dimensional spectrum of $a\acute{a}\acute{a}\acute{a}a\acute{a}aaa\acute{a}$.

1.2 De Bruijn Graphs and Sequences

Formally, given an alphabet Σ of size $\sigma = |\Sigma|$ and a length $d > 0$, a word on Σ is called *de Bruijn sequence* if it contains all possible strings of length d on Σ exactly once as substrings. The d-dimensional *de Bruijn graph* $G = (V, A)$ on Σ is defined by its sets of vertices and edges:

$$\begin{aligned} V &= \Sigma^d, \\ A &= \{\{u, v\} \mid u, v \in V \text{ and } u[i+1] = v[i], 1 \leq i < d\}. \end{aligned}$$

In words, it is the directed graph that has all possible strings of length d over the alphabet as vertices and an arc from vertex u to vertex v if, by deleting the first character of u and the last character of v, we get the same string. Strings of length at least d over the same alphabet describe walks on the d-dimensional de Bruijn graph. In particular, a de Bruijn sequence of dimension d describes a Hamiltonian path[2] in the d-dimensional de Bruijn graph.

Take a vertex v in the d-dimensional de Bruijn graph and consider its prefix v' and suffix w', both of length $(d-1)$. Since v' is a prefix of a string of length d and w' is the suffix of the same string, the suffix of length $(d-2)$ of v' is equal to the prefix of w'. Therefore the pair $\{v', w'\}$ corresponds to exactly one edge in the $(d-1)$-dimensional de Bruijn graph on the same alphabet. Moreover, an edge $\{u, v\}$ in the d-dimensional de Bruijn graph can be mapped into the pair of edges $\{\{u', v'\}, \{v', w'\}\}$ in the $(d-1)$-dimensional de Bruijn graph. As a result, the d-dimensional de Bruijn graph is always

[2]A *Hamiltonian path* is a sequence of vertices where every two consecutive vertices are connected by an edge and every vertex in the graph is found in the sequence exactly once.

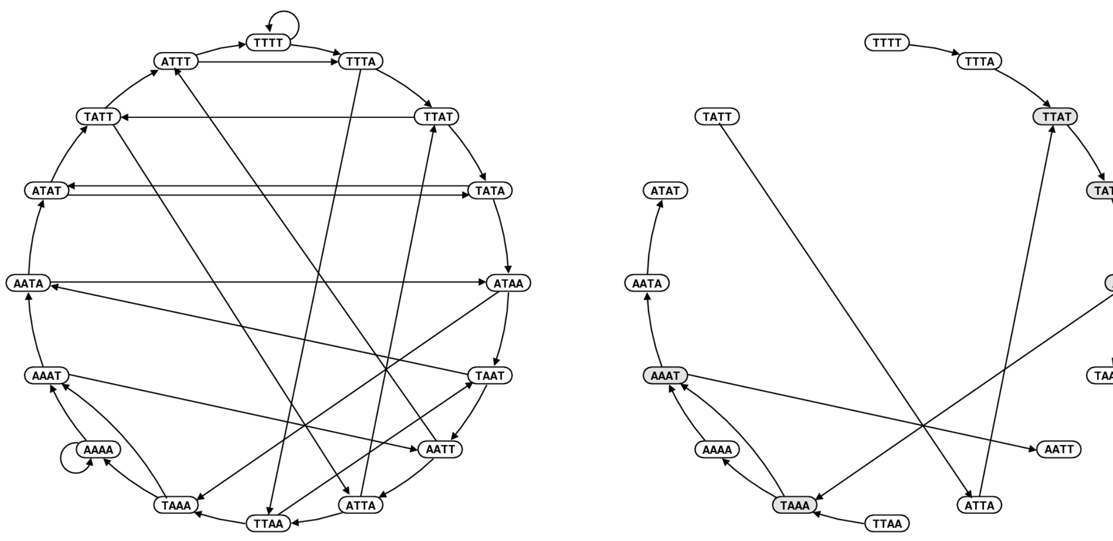

Figure 1.2: The 4-dimensional de Bruijn graph for the alphabet $\Sigma = \{A, T\}$ (left) and the 4-dimensional de Bruijn subgraph associated to the set {`TTTTATAAT`, `TATTATAAATT`, `TTAAAATAT`} (right). On the associated de Bruijn graph, `TTAT` is an example for a junction and `ATAA` for a bifurcation.

the line graph of the $(d-1)$-dimensional de Bruijn graph on the same alphabet. It is easy to verify that for all vertex v' in the $(d-1)$-dimensional de Bruijn graph it holds that indegree(v') = outdegree$(v') = \sigma$, where σ is the alphabet size. This condition is necessary and sufficient to prove that the $(d-1)$-dimensional de Bruijn graph is Eulerian. In Section 1.1.1 we showed a mapping between Eulerian cycles in a graph and Hamiltonian cycles in its line graph. Therefore, each Eulerian cycle in the $(d-1)$-dimensional de Bruijn graph corresponds to an Hamiltonian cycle in the d-dimensional de Bruijn graph, which shows that d-dimensional de Bruijn graphs are Hamiltonian, and there is always a de Bruijn sequence of dimension d.

Given a set of strings, we define the associated d-dimensional *de Bruijn subgraph* as the subgraph of the d-dimensional de Bruijn graph that contains all the walks described by these strings and no extra vertex or arc. A vertex in an associated de Bruijn subgraph is called a *junction* when it has in-degree greater than 1. A vertex with out-degree greater than 1 is called *bifurcation.* Figure 1.2 shows examples of a de Bruijn graph and a de Bruijn subgraph associated to a collection of strings. Note that different sets may have the same associated de Bruijn subgraph. For instance, the set {`TATTATAAT`, `TTTTATAAAATAT`, `TTAAATT`} also has the associated graph shown in

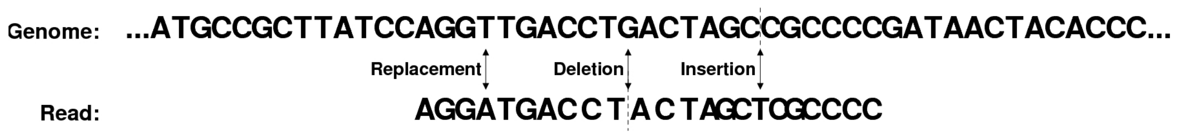

Figure 1.3: The replacement, deletion, and insertion edit-operations.

Figure 1.2 (right).

1.3 De Bruijn Graphs and Bioinformatics

Thirty years after revealing the first sequence of nucleotides within a DNA molecule, we are still not able to read an entire genome at once. Instead we read small contiguous portions of DNA molecules called *reads.* This is a common limitation shared by every sequencing technique. In order to discover a whole genome sequence, we need to solve the *genome assembly problem.*

Definition 1.1. Genome Assembly Problem. *Given a collection of (imperfect) substrings of a string $\mathcal{G}$, determine $\mathcal{G}$.*

Here the genome assembly problem is defined for genomes with only a single chromosome. Although we know that organisms often have more chromosomes, because we human beings all have, this simplification is realistic, since the physical separation of chromosomes is possible, and most genome projects focus on each chromosome individually. Clearly, the substrings in our case are reads, and $\mathcal{G}$ is the genome sequence. With "imperfect" we mean that the substrings in the collection do not necessarily match one of $\mathcal{G}$'s substrings, but may differ from them by a certain number of *edit-operations* shown in Figure 1.3 and described below:

Replacement: One character in the genome has been replaced by a different character in the read.

Deletion: One character in the genome is not found (deleted) in the read.

Insertion: In the read, one additional character occurs which does not exist in the genome.

Assembling the genome is the first task of the majority of genome exploring projects. Therefore, it is not surprising that the first use of de Bruijn graphs in Bioinformatics was the Eulerian path approach to sequence assembly proposed by Idury and Waterman (25) and extended by Pevzner, Tang and Waterman (35). Despite the success achieved by the resulting Euler assembler in assembling bacterial genomes, de Bruijn graphs are not used further for other biological applications. We found many extensions of the de Bruijn graph based assembly approach in recent literature (6; 8; 9; 10; 46), but they usually focus either on improvements in error correction methods or in adapting the original method to new sequencing data. Other works present graphs that slightly remind of de Bruijn graphs, but miss their main feature, namely, the unique representation of tuples of a given size (34; 38). In the next sections we present biological applications based on de Bruijn graphs.

1.3.1 Variations on a de Bruijn Graph Assembler

Theme: Idury and Waterman

Probably the first biological application using an underlying de Bruijn graph was the algorithm for DNA sequence assembly devised by Idury and Waterman (25). The algorithm receives a read collection $\mathcal{S}$ and a positive integer l as input. Initially, all substrings of size l found in the reads are used to construct an $(l-1)$-dimensional de Bruijn subgraph[3]. After that, an Eulerian tour is found in this graph. This tour is supposed to spell the genome sequence. At the end, the reads in $\mathcal{S}$ are aligned to the genome sequence, so that biologists may visually inspect the assembled sequence and evaluate its quality.

Algorithm 1 Idury-Waterman DNA Sequence Assembly Algorithm

```
1: function ASSEMBLY($\mathcal{S}, l$)
2:     Obtain the union of spectra of all fragments and their reverse complements.
3:     Construct the de Bruijn graph on $(l-1)$-tuples for the $l$-tuples generated in line 2.
4:     Perform a variant of Eulerian Tour and infer the sequence
5:     Align the fragments to the sequences produced in the line 4
6: end function
```

[3]Because the set of such substrings is called *spectrum*, the authors call this graph *spectrum graph*.

To deal with sequencing errors, Algorithm 1 does not find an Eulerian tour, but uses heuristics while trying to find the path corresponding to the genome in the tangled graph constructed with the fragments' spectra. Notice that there is no clear optimality criterion in this approach: the algorithm heuristically decides which nodes should be included in the path and which should be skipped. The analogy to Eulerian tours is limited to the ideal error-free case: the tour found by the algorithm in real data is not even expected to be Eulerian.

Variation I: The Euler Assembler

In 2001, Pevzner *et al.* successfully applied Idury and Waterman's ideas to bacterial genomes. The main difference between their Euler assembler (36) and the previously described de Bruijn based assembler is that in Euler the assembling is clearly divided in four distinct steps:

Elimination of erroneous tuples. In Idury and Waterman's assembler, this operation is done together with the "Eulerian" tour construction. In this variant, the elimination of erroneous tuples takes place before the graph construction. Pevzner *et al.* analyze the tuples' multiset found in the reads' spectra and separate the tuples in two groups: *solid* and *weak*, where solid means that the tuple appears in the multiset with multiplicity greater than a given parameter and weak means that the tuple appears with multiplicity up to the parameter. Assuming that the solid tuples should be correct, they are used as a draft of the target genome's spectrum. This draft is in turn used to correct other tuples. The result is a shrinkage of around 97% in the number of read errors, concordant with an error rate reduction from 4.8% to 0.11%.

De Bruijn graph construction. The associated de Bruijn subgraph for the given collection $\mathcal{S}$ of reads is constructed.

Graph simplification. Long induced paths are substituted by single nodes.

Sequence determination. The assembled sequence is output.

As shown in the next sections, most of the de Bruijn graph based assembly methods rely on these four basic steps. In Idury and Waterman's approach, there is an extra fifth step where the reads in $\mathcal{S}$ are aligned to the genome sequence. According to the authors, this is an important step for the assembly validation by biologists, since by visualizing the alignment, they can clearly see how the reads fit in the assembler output.

The Euler assembler notably overcomes the assemblers based on the traditional overlap-layout-consensus paradigm when assembling prokaryotic genomes, but there is no reason to expect that this is also true for eukaryotes. There are many reasons to believe that the step from prokaryotic to eukaryotic genomes may be more difficult for de Bruijn based approaches than for traditional ones:

- The high number of repetitive elements in eukaryotic genomes makes the resulting de Bruijn subgraph much more tangled than the ones found in bacterial genomes. As a result, the number of possible Eulerian tours is bigger, which makes the task of deciding which one corresponds to the target genome more difficult.
- Tandem repeated sequences have similar tuples – sometimes even so similar that a few errors may transform one tuple in another also found in the genome spectrum. Most of the error correction procedures are based on the copy number of the tuples, which is the number of occurrences of a certain tuple in the read collection spectrum. If a sequencing error transforms a tuple into another one of the genome spectrum, the copy numbers for both tuples are no longer precise. As a result, error correction procedures based on copy numbers may fail in these cases.
- Although the memory used to represent de Bruijn subgraphs grows nicely asymptotically, the amount of memory used in practice to represent bacterial genomes is almost prohibitively large. Moreover, eukaryotic genomes are orders of magnitude larger and also much more complex. With a larger genome we are forced to choose between a de Bruijn graph of larger dimension, which prevents different regions from matching by chance, and a graph of smaller dimension, which saves memory, but is surely much more complex and complicates the identification of the correct path.

The authors seem to ignore these facts when they write:

> Our main result is the reduction of the fragment assembly problem to a variation of the classical Eulerian path problem (36, page 257).

The mentioned reduction does not imply any simplification of the genome assembly problem. Finding an Eulerian path in a graph is indeed a computationally easy task. The problem is that biologists do not want "ONE" Eulerian path, but "THE" Eulerian path corresponding to the genome sequence. If we consider that a graph $G = (V, E)$ has

$$C \prod_{v \in V} (\text{outdegree}(v) - 1)!$$

alternative Eulerian circuits, where C is a value that depends on the graph structure[4] (2, Section 9.4), we see that finding the walk in the underlying de Bruijn subgraph corresponding to the genome sequence may be a hard task. Of course the fact that reads describe small trails of the walk may help finding the original one, but recent results of Medvedev *et al.* (31) show that Eulerian path based approaches are not simpler than approaches based on the overlap-layout-consensus paradigm.

Variations II and III: Euler-DB and Euler-SF

Three variations of the basic Euler-assembler were published at the time the original software was presented (33), two of them being described in slightly more detail four months later (35). They all incorporate extra information about the reads in order to simplify the sequence determination step:

Double-barreled data. By slightly changing the sequencing technique, it is possible to obtain pairs of reads whose relative distance in the genome may be estimated. These pairs are called *mate pairs* and can be viewed as larger reads of the form "left known sequence – gap of estimated length – right known sequence". This information was incorporated in the Euler-DB assembler, which tries to find a path in the underlying graph which binds a left to the corresponding right known sequence. If a path with the estimated length is found, the mate pair is exchanged by a *mate-read*, which consists of the whole sequence spelled by the path beginning in the left read, going through the found path, and ending at the right read. Pevzner *et al.* reported that 81% of the used *Neisseria meningitidis* mate-pairs could be transformed into mate-reads.

[4]To be more precise, C is any cofactor of the Laplacian matrix of G.

Scaffolding. Only mate-pairs which correspond to the same connected component in the de Bruijn subgraph have a chance to be transformed into a mate-read. In some cases, the sequences of a mate pair end up in two different components. In these cases, although the gap in the mate-pair cannot be filled, valuable information about the order of the components' sequences is obtained. In Euler-SF, the information of separated mate-pairs is used to connect sinks to sources in the graph with artificially inserted edges, so that an Eulerian path passing through all the components may be found. At the end, the genome sequence is not entirely known, but the order of its fragments can be inferred by the separated mate-pairs.

Variation IV: Playing with short reads

Nowadays there is a number of alternatives to the traditional Sanger sequencing method – most of them producing a much larger amount of shorter reads. In 2004, just after the appearance of the first sequences generated by one of these alternative methods in GenBank, Chaisson, Pevzner, and Tang (9) presented a first analysis of the impact of short reads in de Bruijn subgraph based assemblers. The main contribution of this work is a formal algorithmic solution for the error correction problem solved in Euler. The presented results are again limited to bacterial chromosomes, and unfortunately heuristic approximations are used instead of the exact dynamic programming solution proposed.

The even shorter reads produced by the Illumina-Solexa sequencers are studied by Butler *et al.* (8). They focused on microreads to design their "ALLPATHS" assembler, and used "30-base simulated reads modeled after real Illumina-Solexa reads" to test it. In their simulation, the reads covering the target genome around 80 times, are paired. The assembly procedure is never described in an algorithmic-like fashion and is very similar to the original de Bruijn assembler procedure by Idury and Waterman, though less precise. In fact, sections like *Algorithmic ingredients for unpaired-read assembly* and *Algorithmic ingredients for paired-read assembly* give a strong "cookbook" feeling to the reader. Like in Idury and Waterman's assembler, the de Bruijn subgraph associated to the given read set is built and processed in the hope of getting a simpler graph able to describe the target genome. The read pair information is used to simplify the graph in a bootstrap fashion: first the pairs with smaller distance are processed in the search for the path that connects them and at the same time better approximates the target genome; once instance is solved for closer pairs, the assembler processes more distant ones. Analyzing a pair involves finding all paths in the graph that connect its both ends. It is not difficult to see that low complexity regions may lead to combinatorial explosion. In such cases, the assembler gives up assembling the corresponding region. As a result, the method fails in assembling more complex genomes. This can be seen in the results presented by the authors (8, Table 3): in the human genome assembly, only 0.2% of the genome is covered by contigs with more than 10,000 bases.

The "Velvet" method for short read assembly, presented by Zerbino and Birney (46), uses a structure which is similar to sequence graphs to assemble small genomes of maximally 5 million nucleotides. They show that, although the direct assembly of an entire genome based on short reads may be hard, or even impossible, the conventional idea of physically breaking the DNA molecule into smaller pieces, obtaining a high quality assembly of the small portions, and merging them into an overall good assembly of the whole genome, can be feasible.

Variation V: DNA sequencing with nanopores

Bokhari and Sauer (6) addressed the other extreme of new generation sequencing strategies. Sequencing with nanopores is still far away from being reality. On the other hand, if and when a sequencing machine can be built based on this principle, sequence assembling may become a trivial task.

The principle of nanopore sequencing is simple: A membrane separates two vessels filled with a fluid, one of them containing single stranded DNA molecules. The membrane has several nanopores through which the DNA molecules may pass. The pores are so narrow that the bases are forced to pass one at a time. An electrical potential is applied to the pores. When a base passes through one of them, it causes a change in the electrical potential. This change can be measured, and used to identify the base. In the ideal case, one could read a complete DNA molecule in one pass. However, secondary structures of single stranded DNA, like hairpins, may hinder the molecule flow through the pore, and eventually break it. Therefore, this method is expected to produce reads of average length 10,000 base pairs.

An amazing particularity of nanopore sequencing is that it is not limited to the traditional 5' to 3' end orientation: since polymerases play no role in the sequencing process, and the pores are unable to distinguish between the DNA extremities, some molecules may be read from the 3' to the 5' end. As a result, both complementarity and reverse complementarity are possible, giving rise to 4 different versions of the same molecule. This could be a problem if the reads were not as long as they are. Since the reads are thousands of bases long, the underlying graph dimension may be much larger than in usual de Bruijn graph based applications. As a result, the probability of having a tuple appearing twice in the genome by chance is close to zero; even in two different forms like the sequence and its complement, or the sequence and its reverse complement. The approach proposed by Bokhari and Sauer takes advantage of that: they use an underlying graph with large dimension to force the resulting graph to be formed by four disjoint paths: one for each variant of the target genome. In the case they succeed in finding such a graph, they output the sequences spelled by the four paths as the assembled genomes; otherwise their assembler assumes that the data quality is too poor to allow the genome assembly and reports a failure. Errors are treated in a similar way to the assembler suggested by Idury and Waterman (25): lowly covered nodes are considered to represent sequencing errors and are hence removed from the graph.

The biggest problem in such an approach is evident: even if the assumption that the data produced by such a sequencing method is of very high quality is reasonable, the graph dimension needed to avoid loops caused by exact repeats may be so large that the probability of having a correct tuple is close to zero. In this case, even a huge coverage cannot guarantee that the whole genome will be covered by error free tuples. Moreover, even when the genome is covered by error-free tuples, it may be impossible to separate them from erroneous tuples.

1.3.2 Euler-CN: Counting Copy Numbers

The assemblers Euler-DB and Euler-SF presented in the previous section only adapt previously existing techniques to a de Bruijn graph based genome assembling approach. However, a by-product of this family of Euler assemblers is able to do something new: estimating the number of copies of sequences in a genome before solving the assembly problem.

The approach is simple, but efficient. Pevzner *et al.* (33) defined, for each vertex v in a directed graph $G(V, E, w)$, the value $div(v)$ which is the sum of multiplicities of edges entering v minus the sum of multiplicities of edges leaving it. The edge multiplicities are given by the function w. A graph is called *balanced* if $div(v) = 0$ for all $v \in V$ that are neither sinks nor sources. A flow $f : E \to \mathbb{N}$, with $f(e) \geq w(e)$ for all $e \in E$, such that the graph $G(V, E, f)$ is balanced. The copy number of a vertex is the sum of multiplicities of edges entering (or leaving) the node in $G(V, E, f)$. The principle of parsimony suggests that a flow that minimizes

$$\sum_{e \in E} f(e)$$

should give the real copy number for each vertex, or at least a good approximation of it.

In (33), the authors suggest a reduction to the problem of finding a minimal flow in a network with lower capacity bounds. The reduction consists of adding an artificial vertex to the graph and creating edges going from each sink to the new vertex, and from the new vertex to each source. This approach was used to assign copy numbers to edges in a de Bruijn subgraph built with artificially created reads of the genome of the bacterium *Mycoplasma genitalium*. The authors report the assignment of multiplicities to 3 edges and comment the correctness of one of them. Unfortunately no stronger proof of the concept is provided in the original paper.

Chapter 2

Representing Sparse de Bruijn Subgraphs

Sequence associated de Bruijn subgraphs have a nice asymptotic behavior: their maximum number of vertices increases linearly with the size of the input, and even decreases with the dimension of the graph. The main problem is that, although these graphs scale very well with the sequence set size, graphs corresponding to genomes as small as bacterial genomes are already huge. On the other hand, de Bruijn graphs are by definition sparse (16, Chapter 7). Even in applications where smaller dimensions are required (20), their number of edges in the DNA world is not greater than four times the number of vertices. Their subgraphs are surely sparser. In a typical sequence analysis application (35; 47), the probability of having a vertex with maximum in- or outdegree is very low. Therefore the graph construction in such applications is usually followed by a step where long branch-free paths are collapsed to single vertices (Section 1.3.1).

To represent sparse de Bruijn subgraphs, we use an indexed structure that we call *d-dimensional sequence graph*, or simply *sequence graph*, shown in Figure 2.1. Like a d-dimensional de Bruijn subgraph, every d-tuple over the given alphabet is represented by at most one vertex. As well as that, a sequence graph may contain an arc (u, v) only if the $d-1$ suffix of u is identical to the $d-1$ prefix of v. The main difference between sequence graphs and de Bruijn graphs is that vertices in a sequence graph are not limited to the size d, but may have any size between d and $|\Sigma|^d + d - 1$, which is the maximum size of a d-dimensional de Bruijn sequence[1]. This allows the representation of non-branching paths in a single vertex.

To connect the sequence graph to the concept of de Bruijn subgraphs, we use an index (see Figure 2.1 (right)) mapping every d-tuple to the vertex in which it is found. Remember that vertices may have size greater than d, therefore the representation of a tuple may start anywhere in the middle of a vertex. In order to precisely identify a tuple, not only the vertex, the offset of the d-tuple is given by another index (see circles in Figure 2.1). We also extend the neighborhood concepts from vertices to tuples. Consider two tuples a and b in a sequence graph. We call b the *successor* of a if either a is the suffix of a vertex u, b is the prefix of a vertex v, and the graph contains the arc (u, v); or if there is a vertex v such that $a = v[i \dots i + d - 1]$ and $b = v[i + 1 \dots i + d]$, for a non-negative integer i. We call a the *predecessor* of b in this case, and a and b are called *neighbors*. Note that there may exist vertices u with suffix a and v with prefix b without a and b being neighbors. See, for example, the vertices `TGAGTA` and `TAAGATGCAATATTGTG` in Figure 2.1.

Notice that there is no rule forcing non-branching paths to be represented by single nodes. Depending on how the sequence graphs are built, they may be exactly like their corresponding de Bruijn graphs. In order to compress non-branching paths during the graph construction, some care needs to be taken. Apart from the inclusion of vertices, there are two operations that can be applied on the set of vertices: cutting and merging. They are described in the following sections.

2.1 The Cut Operation

When a sequence is included in a sequence graph, the tuples found only in the sequence need to be added to the graph as new vertices. The tuples already represented by the graph may be in the middle of a vertex. Since tuples that are neighbors in the sequence have to be neighbors in the graph as well, and arcs only connect the last tuple of a

[1]For an intuition about the maximum size of a d-dimensional de Bruijn sequence, remember that de Bruijn sequences can be obtained by superimposing the vertices of d-dimensional de Bruijn graphs in the order they appear in a Hamiltonian path. Since each vertex overlaps with its predecessor in all but the last symbol, after the d symbols of the first visited vertex, each of the remaining $|\Sigma|^d - 1$ vertices adds exactly one new symbol to the sequence, resulting in a sequence with $d + |\Sigma|^d - 1$ symbols.

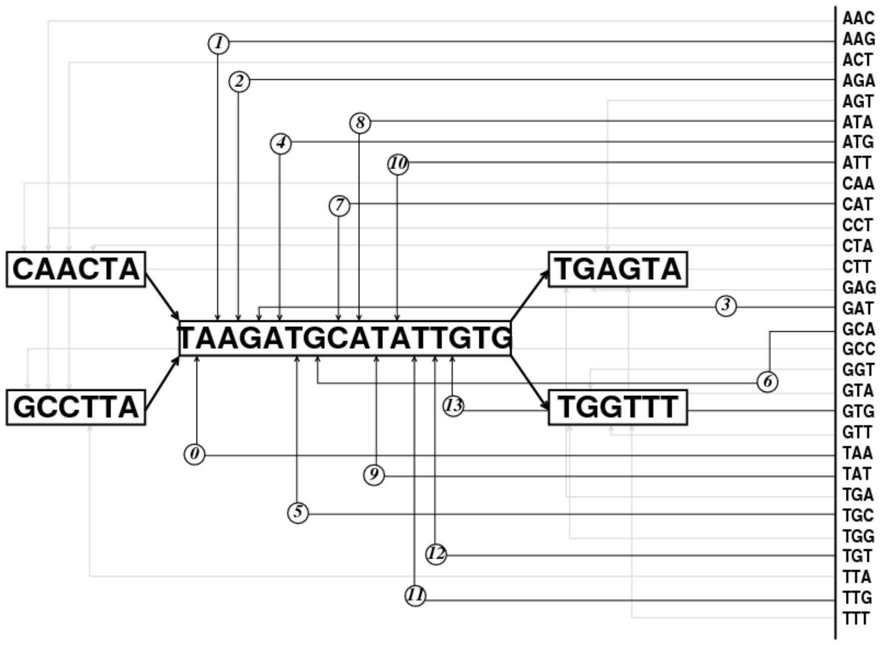

Figure 2.1: Sequence graph corresponding to a 3-dimensional de Bruijn subgraph on the alphabet $\Sigma = \{A, C, G, T\}$. Connectors to the vertex `TAAGATGCATATTGTG` are shown as black arrows with offsets, all other connectors are shown in gray.

vertex to the first of another, in order to represent the sequence correctly, vertices need sometimes to be cut.

The cut operation transforms a single vertex into two neighboring vertices. It is illustrated in Figure 2.2 and presented in pseudo-code in the next page. During the cut, a new vertex is created, and the vertex prefix is transferred to it. In addition, every incident edge to the cut vertex is transferred to the new one. A cut does not change the set of sequences represented by the graph, since no new tuple of size d or greater is created, and the new edge binds two tuples that were neighbors before.

As Figure 2.2 shows, connectors to the cut part are out-of-date after the operation: they should point to the new vertex u after the cut. Updating these connectors would imply a computational cost of $\mathcal{O}(\log|\Sigma|^d) = \mathcal{O}(d \log|\Sigma|)$ for every tuple in the spectrum of the cut part, since all connectors must first be found in the index. To avoid the index search, we postpone the connector update to the next usage of the connector. The update is done by a link to one of the incoming vertices. We call this link the followMe link. It works like an Ariadne's thread, marking the path to the tuples that were once represented by the prefix of the actual vertex.

Algorithm 2 Cut Procedure

```
1: procedure Cut($v, cutPoint$)
2:     create a new vertex $u$
3:     label($u$) ← label($v$)[$0 \ldots cutPoint - 1 + dimension$]
4:     delete the first $cutPoint$ characters of label($v$)
5:     for each edge $(u', v)$ do
6:         remove the edge $(u', v)$
7:         create the edge $(u', u)$
8:     end for
9:     followMe($u$) ← followMe($v$)
10:    starting-point($u$) ← starting-point($v$)
11:    starting-point($v$) ← starting-point($v$) + $cutPoint$
12:    create the edge $(u, v)$
13:    followMe($v$) ← $u$
14:    for each $s \in$ sequence-set($v$) do
15:        put $s$ in sequence-set($u$)
16:    end for
17: end procedure
```

A connector knows that an update is needed thanks to a starting point associated to each vertex. Every time a vertex is cut, its starting point increases by the size of the prefix the vertex loses. As a result, the difference between the connector offset and the vertex starting point is only non-negative if the tuple linked by the connector is still represented by the vertex after the cut. In the case an update is needed, the correct vertex is localized by following the trace left by the followMe links. The connector to the tuple “TAA” in Figure 2.2 shows how it works: after the cut, the offset of the tail is set to 4, and the corresponding difference is -4. This causes the connector to follow the link in the direction to the head of the original vertex. At the head, the difference becomes 0, which shows that the correct vertex was reached.

Many of the connectors may never be used. The ones that are used can only be accessed via the index, and any operation that uses them must pay the computational cost of searching for them. When we create the followMe link, we combine two searches in one. The computational cost of finding a connector is payed by the operations that need to access vertices via the index, and must do the search anyway. Therefore avoiding connector updates reduces the cost of each update to $\mathcal{O}(1)$.

In many applications it is necessary to store a set of sequences where the vertex tuples are found. Since vertices are cut with the intention of inserting a sequence in only

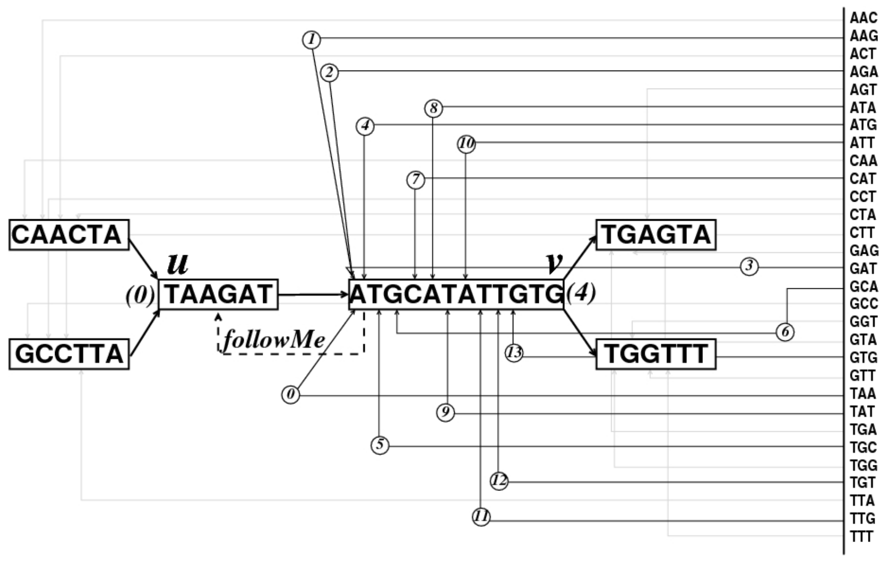

Figure 2.2: Example of the cut operation applied to position 4 of the central vertex in the graph of Figure 2.1. Numbers in parentheses correspond to the vertex starting points after the cut.

one of its parts, the sequence set of each part must be independent after the cut operation. In lines 14 to 16 of Algorithm 2, we duplicate the sequence set of v. This can be done in $\Theta(|\text{sequence-set}(v)|)$ time. Since both the number of connectors to update and the number of characters to transfer to the new vertex are bounded by the size of the vertex, the time needed for a cut is $\mathcal{O}(|v| + |\text{sequence-set}(v)|)$. In general, $|\text{sequence-set}(v)|$ may be as large as $|\mathcal{S}|$, the number of sequences inserted in the graph. In later applications following, we expect to have much smaller sequence sets, since the genome coverage is small.

2.2 The Merge Operation

If two vertices $u, v \in V$ have identical sequence sets and are separated only by the arc (u, v), they may be merged. The merge operation is the inverse of the cut operation. As the name suggests, it removes (u, v) by merging its vertices into a single vertex. This is done by transferring the information from v to u while updating the edges and connectors, so that the vertex v may be removed from the graph afterwards. The operation is presented in pseudo-code in Algorithm 3.

Since the merge operation aims at the complete removal of a vertex, and all connectors pointing to the removed vertex have to be updated, this operation is asymptotically more time consuming than a cut. Since each connector pointing to the second vertex must be found in the index, each merge operation takes $\mathcal{O}(|v|\, d \log |\Sigma|)$ time.

Algorithm 3 Merge Procedure

```
1: procedure MERGE($u, v$)
2:     label($u$) ← label($u$) + label($v$)[$dimension \ldots$ size($v$)]
3:     for each edge $(v, v')$ do
4:         remove the edge $(v, v')$
5:         create the edge $(u, v')$
6:     end for
7:     for each tuple $t \in$ d-tuples($v$) do
8:         update connector($t$)
9:     end for
10:    discard the vertex $v$
11: end procedure
```

2.3 Implementation Details

We assume that the index is implemented by a balanced tree table (13, Chapter 12). The index maps d-tuples to simple data structures called *connectors*. A *connector* is a pair formed by a pointer to a vertex and an integer. The integer is the key to find the d-tuple representation in the vertex: summing up this number to the vertex' starting point either gives the d-tuple offset in the vertex, or indicates that the connector is out-of-date. In the second case, the connector can be easily updated by following the $followMe$ links until the sum becomes non-negative.

In many applications, it is necessary to store in each vertex the sequences which are represented by it, as well as the number of occurrences of the vertex label in each sequence. We assume that sequences can be uniquely identified by an integer. And each vertex has a sequence multiset, where pairs <sequence identifier,multiplicity> are stored in balanced binary search trees (13, Chapter 12). If s is the number of sequences inserted into the graph, these trees have at most s vertices, and insertions and searches can be done in $\mathcal{O}(\log s)$ time.

Vertex adjacencies are stored in arrays of size $|\Sigma|$. Both the edges going from and to the vertices are stored, so that any path in the graph is given by a doubly linked list.

2.4 Finding Repetitive Sequences in DNA Molecules

The main challenge in using de Bruijn subgraphs as a starting point for sequence assembling is that, even for high dimensions, the subgraph associated to a collection of reads is very tangled. Fortunately, finding exact repeats is much simpler than assembling a genome, since we do not need to untangle graphs, but only to identify the tangled parts. Sequence graphs allow a compact representation of sparse de Bruijn subgraphs, but the representation efficiency depends on how cuts and merges are done. In this section, we show how to insert sequences into an initially empty sequence graph in such a way that the number of vertices is minimized at the same time that vertices corresponding to repeats in the inserted sequences are identified.

When identifying vertices corresponding to repetitive regions, we make use of the following variables:

marked(v): Boolean denoting whether vertex v is marked. A vertex is marked when it is part of a repeated region.

sequence-set(v): The set of sequences that has label(v) as substring.

After each sequence insertion, we want the following invariants to hold:

- The value of marked(v) is `true` if and only if label(v) is a repetitive sequence.
- If (u, v) is an edge, then either the out-degree of u or the in-degree of v is greater than 1, or sequence-set(u) $\neq$ sequence-set(v).

2.4.1 Special Operations

Other minor functions and procedures are used to transform sequence graphs or access their vertices. These are:

cutLeft(c). This procedure cuts the vertex linked to the connector c exactly at the position the connector points to. Consequently, the resulting vertex begins with the tuple indicated by c.

cutRight(c). Similar to cutLeft, this procedure cuts the vertex to which the connector c points, creating a vertex where the tuple is at the end of the vertex pointed to by the followMe(node(c)) after the cut.

getConnectors($s, path, i, j$). Let s be a string, *path* be an array of connectors of length $|\text{spectrum}(s)|$, and $0 \leq i \leq j < |\text{spectrum}(s)|$. This procedure acts on the content of $path[i \ldots j]$ in the following way:

- Let $R = \text{spectrum}(s)[i' \ldots j']$, $i \leq i' \leq j' \leq j$, be a maximal region such that there is a vertex n in the sequence graph representing all the d-tuples in R, and in which for every pair of tuples a, b such that b is a successor of a in R, b is a successor of a in n. After the execution of getConnectors, $path[i'] = \text{connector}(\text{spectrum}(s)[i'])$, $path[j'] = \text{connector}(\text{spectrum}(s)[j'])$, and $path[k] = \text{null}$, for $i' < k < j'$.
- For every index i such that the d-tuple $\text{spectrum}(s)[i]$ is not represented in the sequence graph, create a new connector $c = \text{connector}(\text{spectrum}(s)[i])$, and let $path[i] = c$.

The changes on *path* after calling getConnectors is schematically represented by Figure 2.3.

getVertices($path$). This function returns a list of vertices describing the same path in the graph given by the array of connectors *path*. It returns a list containing the vertices corresponding to the connectors in the array *path* in the same order as the connectors. If a series of consecutive connectors correspond to consecutive tuples in a vertex, the vertex is included only once in the list, so that the correspondence between the path defined by the list of connectors and the path given by the vertex order holds. This operation assumes that the operation getConnectors updated the array of connectors *path*, so that it looks like the scheme in Figure 2.3.

contiguous($path, i, j$). Let *path* be an array of connectors. This boolean function returns `true` if the connectors in $path[i \ldots j]$ correspond to a substring of a vertex.

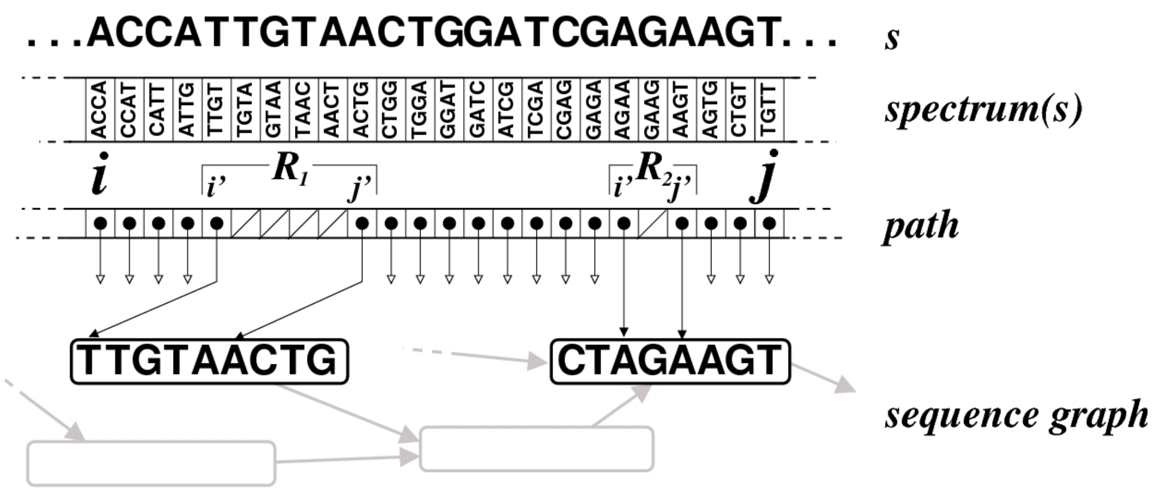
...ACCATTGTAACTGGATCGAGAAGT...
s
ACCA CCAT CATT ATTG TTGT TGTA GTAA TAAC AACT ACTG CTGG TGGA GGAT GATC ATCG TCGA CGAG GAGA AGAA GAAG AAGT AGTG CTGT TGTT
spectrum(s)
i
i'
R1
j'
i'
R2
j'
j
path
TTGTAACTG
CTAGAAGT
sequence graph

Figure 2.3: Array *path* after the procedure calling GETCONNECTORS$(s, path, i, j)$. New connectors are represented by white headed arrows. The old connectors are divided in two regions, R_1 and R_2.

Both cut operations clearly have the same running time as the CUT operation described in Algorithm 2. GETCONNECTORS only uses the index to localize the desired connectors, therefore it runs in $\mathcal{O}((j-i)d\log|\Sigma|)$ time. The function GETVERTICES simply accesses the vertices of a collection of connectors, updating the connectors. However, the computational cost for updating a connector has already been "payed" by the cut operation that caused this cost. Thus, we consider that the connector update is made in constant time, and the function GETVERTICES runs in $\mathcal{O}(|path|)$ time. CONTIGUOUS can clearly be calculated in $\mathcal{O}(j-i)$ time.

2.4.2 Sequence Insertion

The insertion of a string, shown in pseudo-code in Algorithm 4, is done in three phases, which are described below.

Phase One: Changes in the Set of Vertices

In the first phase, the graph is prepared for the sequence insertion. In this phase, existing vertices are cut in order to represent common substrings that the new sequence shares with already inserted sequences. At the same time, new vertices are inserted into the graph, so that the unique new parts can be represented as well. This phase is preceded by a short initialization step, where the connectors to the sequence tuples are either found or created and inserted into the index. At the end of the initialization step, we have an array of connectors which looks like the one shown in Figure 2.3. This structure guides the creation of new vertices, as well as the adaptation of old vertices to the new sequence. Vertices are created or adapted while the sequence spectrum is analyzed from *left* to *right*, that is, from the first to the last d-tuple.

New vertices Lines 7 to 17 of Algorithm 4 detect and fill places where new vertices are needed. The necessity of a new vertex is identified by the presence of *empty connectors*, which are connectors that are not associated to any vertex. Empty connectors are represented by white headed arrows in Figure 2.3. When the leftmost in a series of empty connectors is found, a vertex is created (line 9) and labeled with the first $(d-1)$ symbols of the corresponding d-tuple (line 10). Neighboring empty connectors are iteratively found, and the necessary updates both in the new vertex and in the connectors are done (lines 11 to 16). This step continues until the first non-empty connector or the end of the string is reached.

Old vertices Any vertex connected to a non-empty connector is an *old vertex*. Even newly created vertices can be considered old if they contain tuples which are duplicated in the sequence to insert. The importance of distinguishing between new and old vertices is that only old vertices may be repetitive. This fact is used in phase three to mark repetitive vertices.

If only a part of an old vertex matches the new sequence, the vertex must be cut. Any pair of neighboring d-tuples in the new sequence may force a vertex cut. It is only necessary that at least one of the two tuples is already represented, and that the tuples are not neighbors in the graph. A cut is necessary every time at least one of the cases described below is observed. (Figure 2.4 illustrates these cases.)

Case A: Let a and b be two tuples in the string to be inserted, such that a is the predecessor of b. Suppose b is already represented in the graph by the vertex n. If a is not the predecessor of b in n and b is not the leftmost tuple of n, then n has to be cut into two vertices n_1 and n_2, creating the arc (n_1, n_2), in such a way that b is the leftmost tuple of n_2.

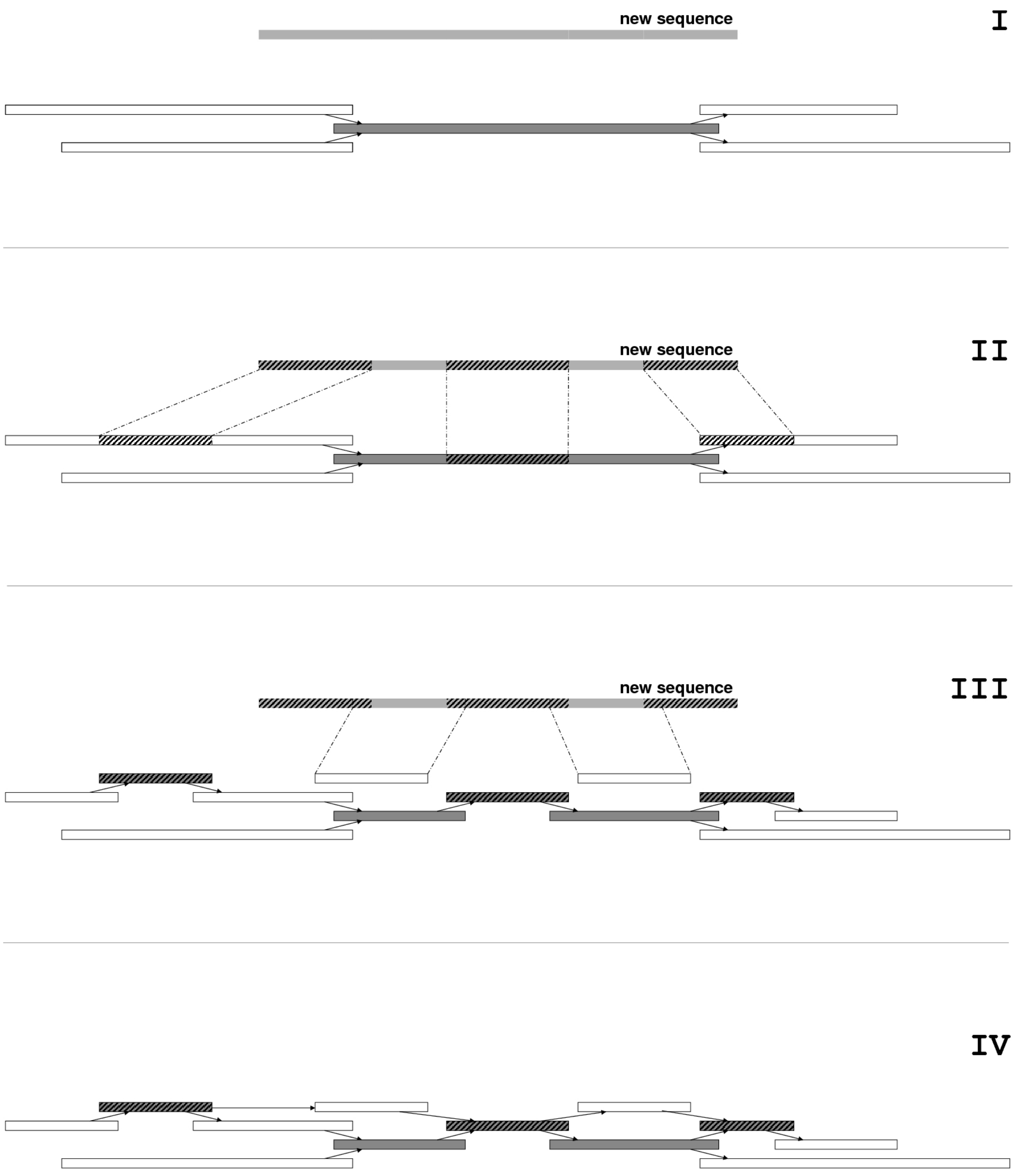

Figure 2.4: Snapshots of a de Bruijn subgraph in three different moments of a sequence insertion. In **I**, we see the graph before the insertion. In **II**, the contiguous regions already contained in the graph are identified. After the identification of contiguous regions, new vertices are inserted and the necessary cuts are done, like shown in **III**. Finally, **IV** shows the graph after the cration of the necessary edges, with the new repeated regions marked.

Case B: Let a and b be two tuples in the string to be inserted, such that a is the predecessor of b. Suppose a is already represented in the graph by the vertex n. If b is not the successor of a in n and a is not the rightmost tuple of n, then n has to be cut into two vertices, n_1 and n_2, creating the arc (n_1, n_2), in such a way that a is the rightmost tuple of n_1.

Case C: Let a be the leftmost tuple in the string to be inserted. Suppose a is already represented in the graph in the vertex n. If a is not the leftmost tuple of n, then n has be cut into two vertices, n_1 and n_2, creating the arc (n_1, n_2), in such a way that a is the leftmost tuple of n_2.

Case D: Let a be the rightmost tuple in the string to be inserted. Suppose a is already represented in the graph in the vertex n. If a is not the rightmost tuple of n, then n has to be cut into two vertices, n_1 and n_2, creating the arc (n_1, n_2), in such a way that a is the rightmost tuple of n_1.

In Algorithm 4, we use the boolean function CONTIGUOUS to identify the longest substring that matches an existing vertex (line 20). Once this substring is found, we are able to analyze the extremities of the corresponding vertex region, so that the necessary cuts may be done (lines 21 and 22).

Phase Two: Walk Connection

The set of vertices may have been modified by cuts in Phase One. Therefore Phase Two also starts with two initialization steps: the update of the connectors in $cPath$ (line 26), and the conversion of $cPath$ into the corresponding vector of vertices $vPath$ (line 27). Phase Two begins with the connection of the vertices in $vPath$ (line 28) and the insertion of the sequence identifier in each of the vertices' sequence sets (line 29).

If vertices may be merged, this is done in lines 30 to 32. It is only possible to merge two vertices if they have the same set of sequences and are neighbors. Vertices with the same set of sequences are either created in the same insertion or are two parts of a cut vertex. If they are created during the same sequence insertion, they are not neighbors by construction. If they are parts of a cut vertex, they can only remain with the same set of sequences if the rightmost portion of the vertex corresponds to the beginning of the sequence, and the leftmost portion corresponds to the end of the

Algorithm 4 Sequence Insertion

```
 1: procedure InsertSequence(s)
 2:     $index \leftarrow 0$
 3:     $last \leftarrow |s| - d + 1$
 4:     $cPath$ is an empty array of connectors of size $last$
 5:     getConnectors($s, cPath, 0, last$)                              ▷ Phase One
 6:     while $index < last$ do
 7:         if node($cPath[index]$) = null then                          ▷ new vertex
 8:             $start \leftarrow index$
 9:             create a new vertex $n$
10:             label($n$) ← d-tuples($s$)[$start$][$0 \ldots d-1$]
11:             repeat
12:                 node($cPath[index]$) ← $n$
13:                 offset($cPath[index]$) ← $index - start$
14:                 append the last symbol of d-tuples($s$)[$index$] to label($n$)
15:                 $index \leftarrow index + 1$
16:             until $index = last$ or node($cPath[index]$) $\neq$ null
17:         end if
18:         if $index < last$ then                                       ▷ old vertex
19:             $start \leftarrow index$
20:             $index \leftarrow \min(x : x \geq start \wedge \neg$contiguous$(cPath, start, x))$
21:             if Case A or Case C then cutLeft($cPath[start]$)
22:             if Case B or Case D then cutRight($cPath[index - 1]$)
23:             mark node($cPath[start]$) as $old$
24:         end if
25:     end while
26:     getConnectors($s, cPath, 0, last$)                              ▷ Phase Two
27:     $vPath \leftarrow$ getVertices($cPath$)
28:     for each pair of neighbors $u, v$ in $vPath$, create the edge $(u, v)$
29:     for each vertex $n \in vPath$, insert $s$ in sequence-set($n$)
30:     if sequence-set($vPath[0]$) = sequence-set(followMe($vPath[0]$)) then
31:         merge(followMe($vPath[0]$), $vPath[0]$)
32:     end if
33:     for each maximal contiguous region old vertices $vPath[i \ldots j]$ do ▷ Phase Three
34:         $R \leftarrow \emptyset$
35:         for each $v$ in $vPath[i \ldots j]$,
36:                 and its left and right neighbors, $u$ and $w$, both possibly null do
37:             $U \leftarrow \bigcup_{u' \in \mathrm{in}(v) \setminus \{u\}}$ sequence-set$(u') \cup \bigcup_{w' \in \mathrm{in}(v) \setminus \{w\}}$ sequence-set$(w')$
38:             $R \leftarrow R \cup$ (sequence-set($v$) $\cap U$)
39:         end for
40:         for each unmarked vertex $n \in vPath[i \ldots j]$ do
41:             if sequence-set($n$) $\cap R \neq \emptyset$ then
42:                 marked($n$) ←TRUE
43:             end if
44:         end for
45:     end for
46: end procedure
```

sequence. Any other disposition would create either a difference between the sequence sets or a prohibitive degree greater than one. Therefore, the only vertices that may be merged after a sequence insertion are the *first vertex in the sequence walk and its previous vertex.*

Phase Three: Repeat Identification

After Phase Two, there is a walk representing the new sequence in the graph, and this walk is stored by the array $vPath$. This last phase identifies vertices in this walk which correspond to repetitive sequences. We call these *repetitive vertices*. Each repetitive vertex was found in the graph during phase one and marked as *old* in line 23. Algorithm 4 differentiates repetitive vertices from common old vertices by comparing each sequence set to the set of sequences that either enter the new sequence's walk through a junction or leave it through a bifurcation.

In the third phase, every maximal contiguous region formed only by old vertices is separately analyzed (lines 33 to 45). For each vertex v in such a region, we find the set of sequences which access v through an alternative walk (line 37). These are all the sequences belonging at the same time to the sequence set of v and the sequence set of the neighbors of v which are not in the newly created walk. In Algorithm 4, these sequences are cumulated in a set R (line 38). Finally, each vertex containing one of the sequences in R is for sure a repeat and may be marked (lines 40 to 44).

Note that every time the walk of a new sequence passes through an old vertex, the vertex may be a repetitive vertex. However, two walks sharing vertices are not always evidence of repeat. When the end of a walk exactly corresponds to the beginning of another one, the common part between them corresponds to a portion of the genome which was sequenced twice. The same is true if one walk is completely contained in another one.

2.4.3 Running Time

In this section, l is the length of the inserted sequence, s denotes the number of sequences inserted so far, and L refers to the maximum length among the s sequences.

Phase One takes $\mathcal{O}(lL + ld)$ time. In the worst case, the $\mathcal{O}(l)$ tuples in the sequence are individually analyzed, and all operations, but the cut, can be executed in constant time. Notice that every l-tuple may cause at most 2 cuts, both bounded by the maximum vertex size, which is bounded by the length of the longest sequence already inserted. At the same time, each tuple may force a search in the index at most two times (in lines 5 and 26). Each search in the index takes $\mathcal{O}(\log |V|) = \mathcal{O}(\log |\Sigma|^d) = \mathcal{O}(d \log |\Sigma|)$ time. Since in our case the alphabet size is constant, each search requires $\mathcal{O}(d)$ time.

In phase Two, the running time is dominated by the insertions of sequences into the vertices' sequence sets (iterative statement in line 29), and by the merge in line 31. The update is done in $\mathcal{O}(l \log s)$ time, since insertions in balanced search trees with size at most s require $\mathcal{O}(\log s)$ time, and at most l insertions are done. Since the only vertices that may be merged are the first and the last vertices of the new sequence walk, merges are done in $\mathcal{O}(l \log l)$ time. Summing up, phase two is executed in $\mathcal{O}(l \log s + l \log l)$ time. Notice that observing the factor $l \log l$ is unlikely, given that situations where the merge is allowed are rare.

The third phase is executed in $\mathcal{O}(ls \log s)$ time. The union in line 37, together with the intersection in line 38 (in parentheses), requires traversing at most $2(|\Sigma| - 1) + 1$ trees, with at most s vertices each. This cannot result in a set with more than s distinct elements. The union with R is done by inserting the elements of the resulting set in R. The whole computation is done in $\mathcal{O}(s \log s)$ time, and at most once for each tuple, giving the proposed running time bound. The overall running time of a sequence insertion is therefore bounded by

$$\mathcal{O}(lL + ld + ls \log s).$$

Remarks on the Running Time Analysis. The reader might have noticed that the time for duplicating the sequence set in the cut operations was not added to the running time of phase one. Since the cuts are responsible for the lL factor in the overall insertion running time, one could argue that the upper bound should be larger. In fact, the time for duplicating the set is covered by the upper bound for the sequence sets update in line 29. To understand why, consider cutting a vertex v with σ sequences in its sequence set. Since v can be cut, it must represent $\tau > 1$ tuples. And the σ

sequences in its sequence set show that for σ times we computed $\tau - 1$ times an unused cost of $\mathcal{O}(\log s)$. We have therefore a "credit" of $\mathcal{O}(\tau\sigma \log s)$, whereas at the time of the cut, the sequence set duplication can be done in $\mathcal{O}(\sigma) = \mathcal{O}(\tau\sigma \log s)$. Therefore we may ignore the duplication in the overall running time.

Chapter 3

Repeat Family Identification in Incompletely Sequenced Genomes

In the later 1980's, scientists had the first contact with genome sequences of higher-order organisms. At that time, they were amazed by the amount of "junk" in these sequences. Examining this junk in the following decades, they discovered that these portions of the genome were less useless than they first suspected. In fact, there is a myriad of active elements between coding sequences, some of them being able to replicate themselves, acting like virus DNA. In fact, they are believed to be the vestiges of virus infections in ancestral species. In prokaryotes, these virus-like sequences are called *insertion sequences*, whereas in eukaryotes they are referred to as *mobile elements*. The most frequently found mobile elements are the *transposons* and *retrotransposons*. In the case we don't want to distinguish between prokaryote and eukaryote genomes, we call them *repetitive elements*.

Although repetitive elements do not refer to active parts of the genome, since they encode only proteins which are related to their own replication, they are able to change the genome in many ways. It is known that pairs of insertion sequences sometimes act together and duplicate not only themselves, but the whole sequence between them (30). Also when mobile elements work alone, the position where the new copy is inserted may belong to important regions in the genome, like active genes. In fact, insertions of mobile elements are observed in several genetic disorders, like Duchenne muscular dystrophy, type 2 retinitis pigmentosa, β-thalassemia, or chronic granulomatous disease (37).

A maybe less noble, but important motivation to study repetitive elements is the waste of time and money they cause in genomic research. Finishing a whole eukaryotic genome sequencing project is neither cheap nor fast, and the study of specific regions in these huge genomes still depends on specific primer design. But even when using very specific primers, PCR experiments may be worthless if the sequence to which the primer was designed appears thousands of times in the whole genome. In many cases, however, there may be enough sequenced information available to give an overview of the repetitive elements in the genome. Finding these elements in an incompletely sequenced, unfinished genome is the aim of the work presented in this chapter.

Strategies for *de novo* repeat identification usually assume that two similar sequences in a given collection cannot be different copies of the same locus of the genome. Then, alignments with quality above a certain threshold are assumed to provide evidence of a repeat family. We overcome this limitation and accept any kind of sequence set as input, including sets with several copies of the same locus. Therefore, we do not align the sequences, like the traditional approaches (4), but partially assemble them using a de Bruijn subgraph.

The differentiation between *repetitive sequences*, that have two or more identical copies in the genome, and *unique sequences* can be done in a sequence graph during the graph construction (Chapter 2). In this and the next chapter we aim at separating reads according to the repeat families they cover, without previously knowing the appropriate families.

3.1 Repeat Families in Sequence Graphs

The length of repetitive elements may vary from the few bases of short tandem repeats to the thousands of bases of long transposons. Exact repeats with length greater than the underlying graph's dimension can be easily identified, since identical parts shared by the copies are represented by the same node both in the sequence graph and in the de Bruijn subgraph. Repetitive elements which are shorter than the underlying graph's dimension can be detected in special cases. For instance, the exhaustive, uninterrupted succession of almost perfect copies in tandem repeats is able to create tangled patterns in the graph, although they may be much smaller than the graph dimension. In these cases, the large number and perfection of copies is responsible for

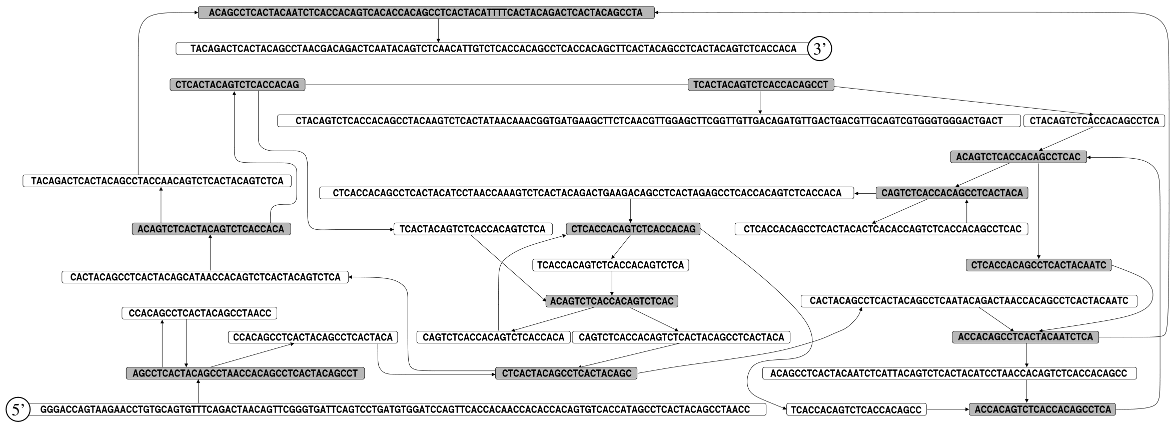
ACAGCCTCACTACAATCTCACCACAGTCACACCACAGCCTCACTACATTTTCACTACAGACTCACTACAGCCTA
TACAGACTCACTACAGCCTAACGACAGACTCAATACAGTCTCAACATTGTCTCACCACAGCCTCACCACAGCTTCACTACAGCCTCACTACAGTCTCACCACA
3'
CTCACTACAGTCTCACCACAG
TCACTACAGTCTCACCACAGCCT
CTACAGTCTCACCACAGCCTACAAGTCTCACTATAACAAACGGTGATGAAGCTTCTCAACGTTGGAGCTTCGGTTGTTGACAGATGTTGACTGACGTTGCAGTCGTGGGTGGGACTGACT
CTACAGTCTCACCACAGCCTCA
ACAGTCTCACCACAGCCTCAC
TACAGACTCACTACAGCCTACCAACAGTCTCACTACAGTCTCA
CTCACCACAGCCTCACTACATCCTAACCAAAGTCTCACTACAGACTGAAGACAGCCTCACTAGAGCCTCACCACAGTCTCACCACA
CAGTCTCACCACAGCCTCACTACA
ACAGTCTCACTACAGTCTCACCACA
TCACTACAGTCTCACCACAGTCTCA
CTCACCACAGTCTCACCACAG
CTCACCACAGCCTCACTACACTCACACCAGTCTCACCACAGCCTCAC
TCACCACAGTCTCACCACAGTCTCA
CTCACCACAGCCTCACTACAATC
CACTACAGCCTCACTACAGCATAACCACAGTCTCACTACAGTCTCA
ACAGTCTCACCACAGTCTCAC
CACTACAGCCTCACTACAGCCTCAATACAGACTAACCACAGCCTCACTACAATC
CCACAGCCTCACTACAGCCTAACC
CCACAGCCTCACTACAGCCTCACTACA
CAGTCTCACCACAGTCTCACCACA
CAGTCTCACCACAGTCTCACTACAGCCTCACTACA
ACCACAGCCTCACTACAATCTCA
AGCCTCACTACAGCCTAACCACAGCCTCACTACAGCCT
CTCACTACAGCCTCACTACAGC
ACAGCCTCACTACAATCTCATTACAGTCTCACTACATCCTAACCACAGTCTCACCACAGCC
5'
GGGACCAGTAAGAACCTGTGCAGTGTTTCAGACTAACAGTTCGGGTGATTCAGTCCTGATGTGGATCCAGTTCACCACAACCACACCACAGTGTCACCATAGCCTCACTACAGCCTAACC
TCACCACAGTCTCACCACAGCC
ACCACAGTCTCACCACAGCCTCA

Figure 3.1: Entangled 21-dimensional sequence graph. This complex cyclic structure is caused by 52.5 imperfect copies of the sequence "AGCCTCACTAC" in the genome of the fish *Fugu rubripes*. The repetitive region starts on the vertex marked with the symbol $5'$ and ends on the vertex marked with the symbol $3'$.

the rising of larger perfect matches. Figure 3.1 shows a sequence found in the genome of the fish *Fugu rubripes* where 52.5[1] consecutive imperfect copies of an 11 nucleotides long sequence entangle a 21-dimensional sequence graph.

In the case of interspersed repeats, their replication mechanism allows the appearance of copies which are physically far away from each other in the DNA molecule. On the other hand, the sequence of each imperfect copy is usually unique. Apart from the usual short reverse repeats in their extremities, the sequence inside mobile elements often lacks long exact repeats. Therefore, the portion of a sequence graph corresponding to a repeat family is much better organized than the tangled tandem repeat regions. Often, the sequence graphs of repeat families are acyclic graphs, like shown in Figure 3.2. This can be used as a starting point for repeat identification.

3.2 Repeat Family Detection Through Connected Components

We first study prokaryotic genomes, since they are less complex and hence easier to understand. A typical bacterial genome is not bigger than six or seven million base

[1]Fractional copy numbers are common in tandem repeats and mean that the last copy corresponds to a prefix of the repeated sequence.

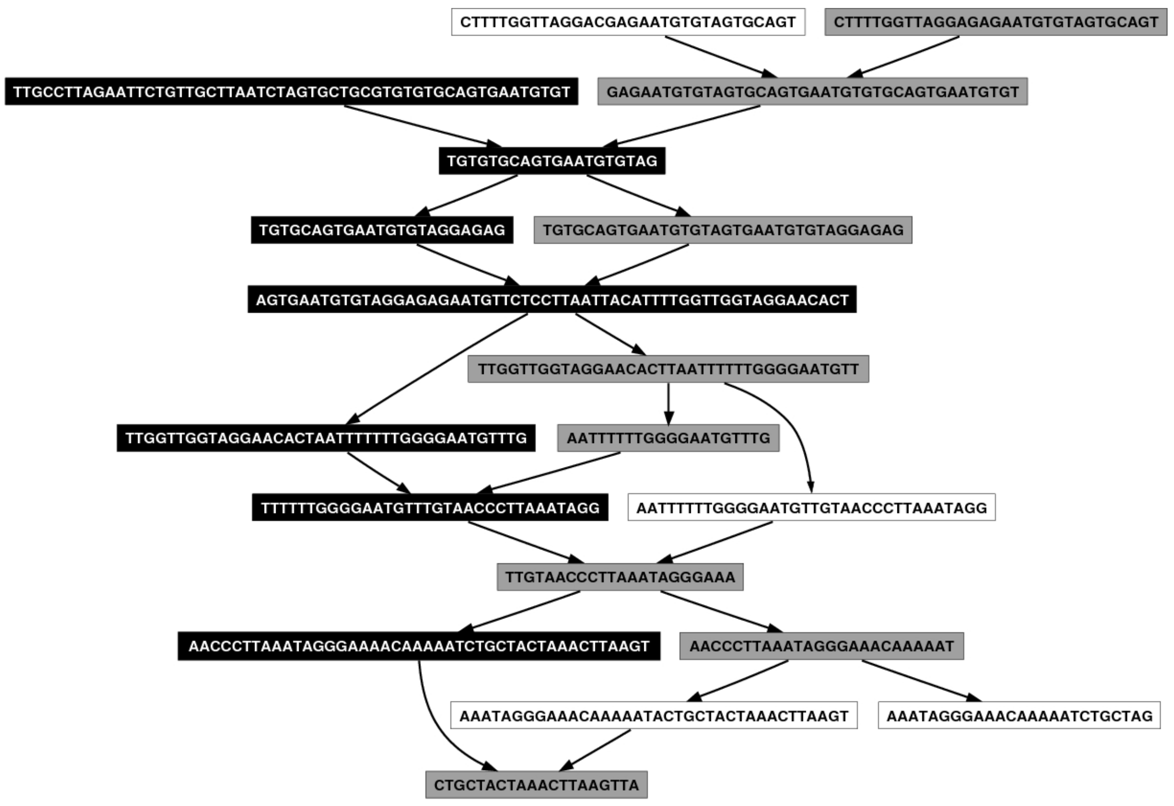
CTTTTGGTTAGGACGAGAATGTGTAGTGCAGT
CTTTTGGTTAGGAGAGAATGTGTAGTGCAGT
TTGCCTTAGAATTCTGTTGCTTAATCTAGTGCTGCGTGTGTGCAGTGAATGTGT
GAGAATGTGTAGTGCAGTGAATGTGTGCAGTGAATGTGT
TGTGTGCAGTGAATGTGTAG
TGTGCAGTGAATGTGTAGGAGAG
TGTGCAGTGAATGTGTAGTGAATGTGTAGGAGAG
AGTGAATGTGTAGGAGAGAATGTTCTCCTTAATTACATTTTGGTTGGTAGGAACACT
TTGGTTGGTAGGAACACTTAATTTTTTGGGGAATGTT
TTGGTTGGTAGGAACACTAATTTTTTTGGGGAATGTTTG
AATTTTTTGGGGAATGTTTG
TTTTTTGGGGAATGTTTGTAACCCTTAAATAGG
AATTTTTTGGGGAATGTTGTAACCCTTAAATAGG
TTGTAACCCTTAAATAGGGAAA
AACCCTTAAATAGGGAAAACAAAAATCTGCTACTAAACTTAAGT
AACCCTTAAATAGGGAAACAAAAAT
AAATAGGGAAACAAAAATACTGCTACTAAACTTAAGT
AAATAGGGAAACAAAAATCTGCTAG
CTGCTACTAAACTTAAGTTA

Figure 3.2: Component of a sequence graph for a set of 454-reads of the plant *Beta vulgaris*. The vertices in black contain sequences of a mariner transposon. Other vertices contain sequence of unclassified reads. Repeated sequences are marked in grey. Notice that the component is a directed acyclic graph.

pairs, and contains roughly the same number of l-tuples, when l is relatively small. The number of possible strings of length l for an alphabet of size 4, by contrast, is already huge for small values of l. For typical sequence analysis applications, like approximate string matching, the value of l is chosen large enough to allow the assumption that very few tuples appear twice in the genome just by chance.

The typical number of insertion sequence families in a single bacterial genome is quite small. The average number of different families in a single genome found in (30) is 2.79. The copy number of a family in a single genome is also not big. Although the number of copies can be as big as 14, like the number of copies belonging to the family IS1 in Mycoplasma, the average number of distinct elements of the same family is 2.27. Therefore, assuming that copies are uniformly spread along the genome sequence, we may expect repeats to be separated by quite long non-repetitive sequences. This may be also true for some eukaryotes, like *Arabidopsis thaliana* which has 10% of its genome composed by mobile elements (19), while other eukaryotes have a much more complicated genome structure.

Nodes corresponding to repetitive sequences may be discovered and marked during the sequence graph construction, as explained in Chapter 2. Nodes corresponding to unique sequences either represent larger sequences from the unique parts of the genome, or are the result of small dissimilarities between elements of the same repeat family. In the second case, unique nodes should not be larger than a repeat family element, since we expect entities of the same repeat family to be similar enough to share perfect matches. On the other hand, unique sequences between repeat copies can be much longer.

The sequence graph for a genome must therefore be composed of clusters of small repetitive nodes, interconnected by longer ones representing single sequences. As a result, the deletion of long unique nodes may decompose the graph into a few connected components, containing one or more repeat families. Based on this simple principle, we devised a method for separating repeat families in a genome. The procedure is described in Algorithm 5. The input is a set $\mathcal{S}$ of reads of some genome and a length threshold value l. First, we build the sequence graph for this set of sequences.

Originally, nodes with different sequence sets cannot be merged (see Chapter 2). As a result, every read end coincides with a node end, which leads in many cases to branch free paths in the sequence graph. Here we ignore this restriction and merge nodes in

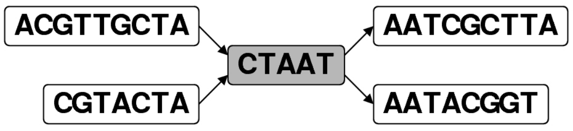

Figure 3.3: Some errors may transform a tuple into another tuple of the genome spectrum. When this happens, we observe a small repetitive node surrounded by unrelated unique sequences.

branch free paths, as long as they are either both marked as repeats, or both unmarked, even if their sequence sets are not identical. The resulting graph may contain long single nodes, exceeding the length threshold l. These nodes are consequently removed from the graph. As a result, node pairs that could not be merged before may now be merged. Notice that, after merging these nodes, no other node can be merged or deleted.

The resulting graph is already a collection of separated connected components. However, some of them may be the result of unrelated small perfect matches. These repeats created by chance are easy to identify. They are in components with few nodes (typically not more than 5), with a single, short repeated node in the center, like shown in Figure 3.3. We call these components *small components*. The small components are removed from the graph as well, leaving only components corresponding to larger families.

Algorithm 5 Connected Components

```
1: function IsolateComponents($\mathcal{S}, l$)
2:     Build the sequence graph for $\mathcal{S}$
3:     Merge all possible pairs of nodes
4:     Remove all single nodes of length $\geq l$
5:     Merge all possible pairs of nodes
6:     Remove all small components
7:     return the resulting connected components
8: end function
```

3.3 Proof of Concept

We implemented this approach in the Java programming language and tested it with artificially created chromosomes. Unfortunately, we cannot measure the quality of our approach by comparing it to already published *de novo* repeat identification methods, since we use a collection of reads as input instead of a complete chromosome. Actually, we do not even assume to have enough reads to cover the entire genome. Therefore we make use of artificial data sets to verify the reasonability of this approach. For sure artificial data have the disadvantage of not representing a real life situation, but they also have the following advantages:

- The number of families is known.
- The number and sequence of distinct family elements is known.
- No other repetitive elements are there, so that we can focus on the simulated insertion sequences.

For a proof of concept, we applied the connected component strategy to artificially created chromosomes with different numbers of repeat families. Each simulated chromosome has a total length of 1 million base pairs and is composed by two kinds of sequences:

Background Sequence: The background sequence corresponds to the unique genome sequence. In our tests, we used 19-dimensional de Bruijn subsequences as background, which means that the background sequences do not contain any duplicated substring of length 19.

Repeat Families: The repeat families are collections of similar sequences, called the *family members*. They originate from a 19-dimensional de Bruijn subsequence, called the family's *base sequence*. The base sequence is used to start creating other family members in an incremental tree-like fashion: for creating a new member, we randomly choose one family member and imperfectly duplicate it by simulating insertions, deletions and replacements. Each newly created sequence differs from its original in 6% of the nucleotides on average. This agrees with real cases, like the Alu family in the human genome, where the sequences diverge by up to 12% from other elements in the family (26). The number of members in a family is called the *family size*.

The inserted repeat families were of size 2, 4, 16, and 256. In our tests, an artificial chromosome can have either 0 or 2 families of each size. All possible combinations of family sizes were used, giving a total of 15 chromosome configurations. For each configuration, we created 15 different chromosomes and read sets with coverage 0.25, 0.5, 0.75, and 1, respectively, simulating partially finished sequencing projects. The length of the artificially created reads follow a Poisson distribution with mean 250.

3.3.1 Results

We used the sets of reads as input to the connected component based repeat family detector. The result was a collection of connected components for each artificial data set. The sequences of the family members were used to mark the vertices where repeat families are found, which creates an association between families and connected components. Based on this association we analyzed the number of components in which a family is split, the number of different families found in each components, and the percent of families that are represented by components in the method output. These numbers are presented in Table 3.1.

Ideally we would be able to isolate each family in a different component, so that each connected component contains sequences of a single family, and each family is completely contained in a single component. Table 3.1 shows a different reality. In the left column ("Components per Family"), we see that families are usually split into more than three components. However, each component usually contains sequences of a single family, which is shown by the center column ("Families per Component"). This means that although the families are split, they are at least not so dispersed that their separation is impossible.

In the rightmost column ("Discovered Families (%)"), we see how much of the inserted families could be detected by the method. The fact that we were never able to identify all families in the odd rows is expected. These are cases where the chromosomes have some family of size two. In such cases, depending on the underlying sequence graph's dimension and the difference between the family members, it can happen that these two family members do not share any tuple, or the number of shared tuples is so small that they end up being discarded as small components. In Chapter 4 we discuss a strategy for keeping such small components by identifying non-repetitive sequences which are similar enough to be considered member of the same family even though they do not share any common subsequence larger than the graph dimension.

Analyzing the data in Table 3.1, we come to the conclusion that the worst problem of this approach is the splitting of families into up to 5 components. In order to improve the method's efficiency we need to understand why the splitting happens, and if it is a problem at all. We classify the splittings into two types:

Horizontal Splitting: Since the repetitive elements are not essential parts of the active genome, they are usually not under evolutionary pressure. As a result, once a new copy of a repeat family is created, it starts diverging from its original sequence. Depending on divergence time and speed, these sequences can accumulate so many mutations that they do not share any tuple. If they keep replicating themselves, the new copies of the first sequence may cluster in a component, whereas new copies of the second one cluster in another one. In this case, although all the sequences have a common history, they give rise to two different components in the resulting graph. We call this separation of a family into two disconnected groups *Horizontal Splitting.*

Vertical Splitting: In many cases there are regions in the genome that are more susceptible to change than others. If there is such a region in the sequence of a repeat family, the component containing the prefixes of the family members may be connected to the component containing its suffixes only by long unique nodes. If these nodes are long enough, they are discarded in Step 4 of Algorithm 5, resulting in two components for the same family: one containing its prefixes, and one with the suffixes. We call this decomposition of a family into its prefixes and suffixes *Vertical Splitting.*

Differentiating between horizontal and vertical splitting gives more information than just a name for effects of mutation events in a graph. There is a basic difference between these two kinds of component splitting: vertical splitting necessarily shows a method's weakness, whereas horizontal splitting may be just a negative effect of having too much information about the evolutionary history of the data.

Biologists group repeat families by structure and similarity of their sequences. They do not discard the hypothesis that two distinct families could have a common origin. However, if family members cumulated enough dissimilarities along the time to be classified as different families, they should not be merged into a single family. This is not reflected in our simulations: we group all the sequences originated from the same

Table 3.1: Summary of the experiments with artificial data. Each row corresponds to one of the 15 types of data sets. On the left, we see the average number of different components containing sequences of the same family. In the middle, the average numbers of different families found in a single component are displayed. On the right, we see the percentage of inserted families which could be found in the graph after eliminating long nodes.

Components per Family				**Families per Component**				**Discovered Families (%)**			
0.25	**0.50**	**0.75**	**1.00**	**0.25**	**0.50**	**0.75**	**1.00**	**0.25**	**0.50**	**0.75**	**1.00**
6.33	5.57	4.41	5.68	1.00	1.00	1.00	1.00	20	40	70	73
4.50	3.50	4.33	4.70	1.00	1.00	1.00	1.00	63	97	97	100
5.26	4.78	4.82	4.32	1.00	1.00	1.00	1.00	43	70	85	92
4.27	4.37	4.18	4.42	1.00	1.00	1.00	1.00	100	100	100	100
3.93	4.48	4.31	4.12	1.00	1.00	1.00	1.02	58	78	87	93
4.41	4.01	4.31	4.47	1.00	1.01	1.00	1.02	83	95	100	100
4.31	4.83	4.25	4.20	1.01	1.01	1.02	1.05	64	82	91	94
4.08	3.80	3.89	4.44	1.93	1.93	1.93	1.93	100	100	100	100
4.68	4.56	3.98	4.51	1.78	1.93	1.61	1.51	65	70	80	90
4.38	4.93	4.45	4.72	1.78	1.50	1.58	1.91	85	100	100	100
4.88	4.36	4.28	4.14	1.64	1.55	1.40	1.48	60	77	89	96
4.83	4.69	5.08	4.23	2.18	2.89	2.98	3.31	100	100	100	100
4.48	4.89	4.81	4.50	2.29	2.19	2.36	2.28	72	89	90	98
4.28	4.62	4.63	4.24	1.66	1.82	2.35	2.50	89	98	99	100
4.26	4.51	4.91	4.98	1.60	1.66	1.78	1.87	68	93	97	98

base sequence into the same family, disregarding the fact that evolution may separate them into two or more families.

Vertical splitting can be an indicator of a method's weakness: if the sequences are so similar that their prefix and suffix cluster in different components, the method should avoid removing the nodes by binding these components.

Chapter 4

Combining Vertices to Improve Sensitivity

In the previous chapter we presented an approach for grouping sequences of the same repeat family in a single component of a sequence graph. We finished the chapter presenting two causes for the splitting of repeat families in independent connected components. We also discussed that horizontal splittings are caused by the natural accumulation of discrepancies inside a family of sequences that separately evolve from a common ancestor, and argued that separations due to horizontal splitting do not imply a problem of the approach. On the other hand, vertical splitting have a significant impact on the quality of the family separation strategy. In this chapter we present an strategy for reducing vertical splitting in a sequence graph.

Let v be a bifurcation in a d-dimensional sequence graph, and let t and u be two successors of v. We call t and u *parallel* vertices. By the definition of sequence graphs, all pairs of parallel vertices coincide in the first $d-1$ characters and differ at least on the d^{th}. They only appear in graphs representing repetitive sequences, and are found in two situations, according to their relative positions in repeat families:

Inside repetitive regions, where sequencing errors or point mutations in different copies separate small portions of sequences that were originally identical. Figure 4.1 (Box 1) shows the pair of parallel vertices `TGCGATC` and `TGCAATC`. Notice that the vertices differ only in their central base. If this base was the same, both strings would be represented by a single vertex, and we would observe a single

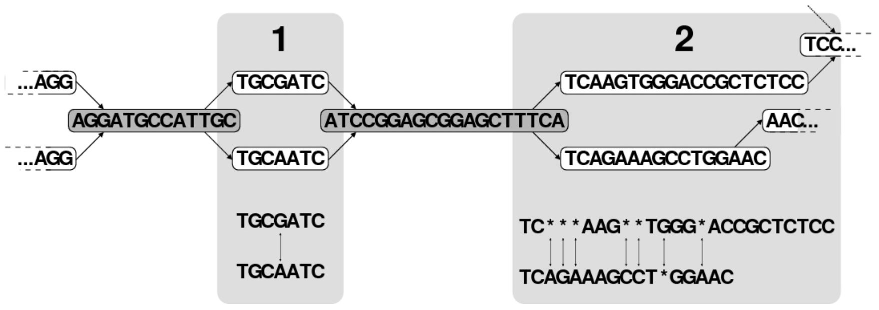

Figure 4.1: Part of a 4-dimensional sequence graph showing two examples of parallel vertices. In box 1 we see two vertices inside a repetitive region, while box 2 shows vertices outside a repetitive region. Under each pair of vertices we see best alignment between the vertices' prefixes.

long repetitive vertex instead of the two shown in the figure. Therefore we say that the central base *separates* these parallel vertices.

Outside repetitive regions, at the extremity of family members, on the border between the vertices representing repetitive regions and the ones representing evolutionarily uncorrelated sequences. We find an example of such a parallel pair in Figure 4.1 (Box 2), where 7 deletions in a 17 bases long alignment indicate that the sequences in the vertices' prefixes probably are not evolutionarily correlated.

Notice that in both pairs shown in Figure 4.1, the vertices correspond to unique sequences. According to the approach for identifying repeat families presented in Section 3.2, all these vertices could be deleted from the graph. The deletion of vertices outside repetitive regions is not only desired, but the core of the approach. On the other hand, by deleting vertices inside repetitive regions, we create undesired vertical splits in our graph. To avoid this, we identify parallel vertices inside repetitive regions, combine them into a single vertex, and mark the combined vertex as a repetitive region.

As the alignments in Figure 4.1 suggest, we differentiate the two kinds of parallel vertices based on the dissimilarity between the sequences inside them. We define a threshold for the dissimilarity between the sequences and consider that any pair of parallel vertices with dissimilarity under the threshold is inside a repetitive sequence. This threshold, τ, is a scale factor of the minimum number of sequence editions necessary to separate the parallel vertices. For instance, if $\tau = 1.0$, only vertices separated by the minimum number of operations are supposed to be inside repetitive region. If

$\tau = 1.5$, parallel vertices separated by up to 50% more editions than the minimum are still considered inside a repetitive region. In an extremal case, if $\tau \geq d$, none of the pairs of parallel vertices will be considered outside a repetitive region. The number of editions allowed for a pair of vertices, $\varepsilon(l, d)$, is given by

$$\varepsilon(l, d) = \tau \cdot \mu(l, d),$$

where $\mu(l, d)$ is the minimum number of edit-operations needed to separate two initially identical sequences of length l in a sequence graph. This number is given by

$$\mu(l, d) = \left\lceil \frac{l - d + 1}{d} \right\rceil .$$

By *combining* two vertices we mean replacing them by a single vertex whose sequence is the consensus between them. The procedure is shown in Algorithm 6 and represented in Figure 4.2. In the most general case, the two vertices to be combined, v_1 and v_2, are of different length. We assume w.l.o.g. that v_1 is longer than v_2.

Algorithm 6 Combine

```
 1: procedure COMBINE($v_1, v_2, \tau$)
 2:     Let $v_1$ be the longer of the two vertices
 3:     Align the sequences of $v_1$ and $v_2$, creating a semi-global alignment of length $l$
 4:     if the alignment score is smaller than $\tau \cdot \mu(l, d)$ then
 5:         Cut the vertex $v_1$ at the end of the aligned prefix
 6:         Let $\overline{v_1}$ be the left portion of the cut vertex $v_1$
 7:         Create a new vertex $v$ with the consensus of $\overline{v_1}$ and $v_2$
 8:         Bind the vertices in the neighborhood of $\overline{v_1}$ and $v_2$ to $v$
 9:         Remove $\overline{v_1}$ and $v_2$
10:     end if
11: end procedure
```

In the first step, the vertex prefixes are aligned. We use a semi-global alignment algorithm for that (40, Section 3.2.3). The semi-global alignment gives the best alignment between the shorter sequence and the prefix of the longer one. We assume that the original sequence has the same length as the alignment, and use the alignment length to calculate $\varepsilon(l, d)$. When the prefix alignment scores less than $\varepsilon(l, d)$, we assume that the parallel vertices are inside a repetitive region, and proceed with the combine operation. The longer of the two vertices, v_1, is cut at the point where its aligned

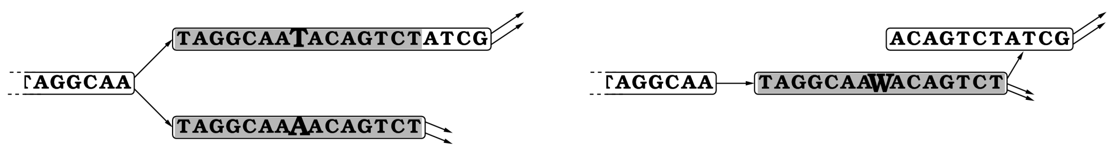

Figure 4.2: The *combine* operation. The two shaded vertices on the left are combined, and result in the shaded vertex on the right. Two vertices are only combined when the edit distance between their prefixes is below a certain threshold. The new vertex label contains the consensus sequence.

prefix ends. The vertex corresponding to v_1's prefix and the vertex v_2 are replaced by a new vertex v, which represents the alignment consensus. The new vertex v is finally connected to the neighborhood of the replaced vertices.

The distance matrix used in the alignment algorithm, $\mathcal{M}$, is based on the IUPAC standard one-letter code for nucleotides, which has not only symbols for nucleotides, but also for all possible sets of them. Let s and t be two letters in the IUPAC standard code, and S and T the corresponding sets. Then the value of $\mathcal{M}_{st}$, the score for matching s with t, is given by

$$\mathcal{M}_{st} = \begin{cases} 0 & \text{if } S \subseteq T \text{ or } T \subseteq S, \text{ and } S \neq \emptyset \neq T \\ \max\left(|S \setminus T|, |T \setminus S|\right) & \text{otherwise} \end{cases}$$

Notice that for symbols that represent a single nucleotide (`A`, `C`, `G`, `T`), the score is simply 0 for a match and 1 for a mismatch. For matches involving at least one symbol representing a set of nucleotides, like `W` = {`A`, `T`}, the score is 0 if one set contains the other; otherwise it is the minimum number of replacements and deletions needed to transform one set into the other. For instance, the score for aligning `W` with `G` is 2, since we need to replace one of the elements of `W` by `G`, and delete the remaining one; on the other hand, the score for aligning `W` with `T` is 0, since `W` contains `T`.

The consensus is created based on the alignment between v_1's prefix and v_2: each position in the alignment corresponds to the union of the IUPAC symbols in the corresponding positions in both vertices. In the example in Figure 4.2, the symbols in all the positions but the central one represent the same sets, and are kept in the combined vertex. The symbol at the central position of the new vertex, `W` = {`A`,`T`}, is the union of the symbols in the single discrepant position in the vertices.

In general, combining two vertices does not reduce the graph size, since one vertex is split and two are merged. In practice, a series of combinations may reduce tangled subgraphs to simple paths, which may be finally merged into a single longer vertex. Especially for more complex data-sets sequenced at low genome coverage, this procedure gives a considerable advantage over the simple connected component approach, as shown by the results in Section 4.1.

4.1 Effectiveness in Bacterial Genomes

In order to evaluate our more advanced algorithm, we created a data-set with real bacterial genome sequences and their known insertion sequences. The bacterial chromosomes were obtained from the NCBI Website[1], while the corresponding insertion sequences were obtained from the insertion sequence database *IS Finder*[2]. We created 30 read sets covering 25, 50, 75, and 100 percent of the genome on average. Each of the read sets was used seven times as input for the *combine* method, each time with a different combine scale factor $t \in \{1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0\}$. Like in Section 3.3 we associated the resulting connected components to the repeat families found in each of them. In opposite to the experiments with artificially created sequences, here we do not know the complete set of insertion sequences in the chromosomes, which makes it impossible to decide whether unmarked components correspond to unknown insertion sequence families. Therefore we focus our analysis exclusively on the associated components.

We base our analysis on the three values used in Section 3.3.1: the fraction of known families found in the set of components, shown in Table 4.1, the average number of distinct families in a single component, presented in Table 4.2, and the average number of components in which a family is split, shown in Table 4.3. The tables in this section summarize the data obtained using 100% genome coverage. The tables for 25%, 50%, 75%, and 100% coverage can be found in Appendix A.

In Table 4.1 we find the percentage of known families identified in the components. The data values show an increasing percentage of discovered families with increasing combine factor. This could mislead the reader to believe that the largest possible

[1]NCBI: `http://www.ncbi.nlm.nih.gov`
[2]IS Finder: `http://www-is.biotoul.fr/is.html`

Table 4.1: Average percentage of known families which were discovered at 100% coverage.

Combine Factor (τ)	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**
Bifidobacterium longum	49	72	71	81	81	83	82
Burkholderia xenovorans	50	84	84	88	88	90	90
Colwellia psychrerythraea	20	33	33	33	33	33	33
Desulfitobacterium hafniense	80	95	95	100	100	100	100
Desulfovibrio desulfuricans	50	93	93	97	97	100	100
Escherichia coli	43	75	73	85	85	92	92
Geobacter uraniumreducens	46	72	72	72	72	80	80
Gloeobacter violaceus	63	85	85	88	88	98	98
Granulibacter bethesdensis	7	30	27	53	50	70	73
Haloarcula marismortui	13	47	44	75	69	90	91
Halobacterium sp-plasmid pNRC100	39	57	47	56	51	61	59
Legionella pneumophila	32	30	33	33	40	30	30
Legionella pneumophila-Philadelphia 1	38	70	63	87	83	93	83
Methanosarcina acetivorans	95	99	99	100	100	100	100
Methylococcus capsulatus	44	82	83	96	96	98	98
Nitrosospira multiformis	70	97	93	100	100	100	100
Photobacterium profundum	100	100	100	100	100	100	100
Pseudomonas syringae	97	100	100	100	100	100	100
Pyrococcus furiosus	53	72	69	84	80	94	94
Ralstonia solanacearum	63	88	88	95	95	98	98
Rhodopirellula baltica	89	100	100	100	100	100	100
Roseobacter denitrificans	50	90	87	100	100	100	100
Salinibacter ruber	97	100	100	100	100	100	100
Shewanella oneidensis	27	29	29	38	41	33	35
Sulfolobus solfataricus	99	99	99	100	100	100	100

combine factor would always give the best result. However, using large factors has disadvantages, as we will discuss in the analysis of the next two tables. As the genomes of *Halobacterium sp-plasmid pNRC100*, *Legionella pneumophila-Philadelphia 1*, and *Shewanella oneidensis* show, the percentage of discovered families can decay if the combine factor is too large. This happens because large thresholds allow the combination of unrelated parallel vertices. In extreme cases, the resulting graph is poorly informative and full of vertices with symbols representing groups of nucleotides.

The side effects of using larger combine factors is better seen in Table 4.2, where the average number of different families in a single component is presented. Ideally this number should be as close to 1.0 as possible, since we aim at separating families into *individual* components. The data shows that too large values do the opposite. Specially in genomes like *Ralstonia solanacearum* and *Sulfolobus solfataricus* the use of the combine operation interferes drastically with the ability of separating families. However, larger factors affect different genomes in different ways: in *Gloeobacter violaceus*, using a combine threshold 1.5 increases the percentage of found families from 63% to 85%.

The reader may remember that the aim of the combining operation is to avoid the

Table 4.2: Average number of different families found in the same components at 100% Coverage.

Combine Factor (τ)	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**
Bifidobacterium longum	1.2	1.2	1.2	1.2	1.2	1.3	1.3
Burkholderia xenovorans	1.0	1.5	1.5	2.1	2.1	2.4	2.4
Colwellia psychrerythraea	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfitobacterium hafniense	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfovibrio desulfuricans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Escherichia coli	1.0	1.1	1.0	1.1	1.1	1.1	1.1
Geobacter uraniumreducens	1.4	1.4	1.4	1.4	1.4	1.5	1.6
Gloeobacter violaceus	1.0	1.1	1.1	1.4	1.4	1.7	1.7
Granulibacter bethesdensis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Haloarcula marismortui	1.0	1.0	1.1	1.5	1.6	2.0	1.9
Halobacterium sp-plasmid pNRC100	1.1	1.2	1.2	1.3	1.5	1.4	1.5
Legionella pneumophila	1.0	1.2	1.2	1.4	1.3	1.2	1.2
Legionella pneumophila-Philadelphia 1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Methanosarcina acetivorans	1.6	1.5	1.5	2.1	2.2	3.6	3.6
Methylococcus capsulatus	1.0	1.0	1.0	1.1	1.1	1.5	1.5
Nitrosospira multiformis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Photobacterium profundum	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Pseudomonas syringae	1.4	2.0	2.0	2.2	2.2	3.6	3.6
Pyrococcus furiosus	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Ralstonia solanacearum	1.4	3.2	3.5	4.4	4.5	6.9	6.9
Rhodopirellula baltica	1.3	1.3	1.4	1.4	1.4	1.7	1.7
Roseobacter denitrificans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Salinibacter ruber	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Shewanella oneidensis	1.5	2.2	2.2	2.3	2.3	2.7	3.0
Sulfolobus solfataricus	2.1	2.9	3.2	4.5	4.7	5.4	5.4

deletion of vertices corresponding to unique sequences inside repetitive regions. With the combine operation we target the vertical splitting and hoped for a less split graph. In this sense, the data in Table 4.3, the average number of components in which a family is split, brings good and bad news. For the genomes of *Burkholderia xenovorans*, *Gloeobacter violaceus*, *Haloarcula marismortui*, *Halobacterium sp-plasmid pNRC100*, *Methanosarcina acetivorans*, *Rhodopirellula baltica*, *Salinibacter ruber*, *Shewanella oneidensis*, and *Sulfolobus solfataricus* the pattern observed in the data is exactly the expected one: with larger factors the number of components per family is reduced, which means that vertices inside repetitive regions are kept and avoid unwanted component splittings. For many other bacteria we notice that the number of components either increases with increasing combine factor or increases up to a certain point before it starts decreasing. This shows that for some genomes the combine operation avoids the complete deletion of some components up to a certain combine threshold. Only after this point the desired effect of the combine operation is observed, namely the reduction of the number of components.

Table 4.3: Average number of components per repeat family in a sequence graph (100% coverage).

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	2.5	3.4	3.1	3.7	3.6	3.8	3.7
Burkholderia xenovorans	2.6	2.0	2.0	1.5	1.5	1.3	1.3
Colwellia psychrerythraea	3.1	3.1	3.1	3.1	2.9	3.1	3.1
Desulfitobacterium hafniense	3.3	4.1	3.7	3.9	3.7	3.3	3.3
Desulfovibrio desulfuricans	2.3	2.8	2.5	3.6	3.3	3.3	3.1
Escherichia coli	2.2	3.0	3.0	3.1	3.1	3.1	3.1
Geobacter uraniumreducens	1.8	2.2	2.2	2.5	2.5	2.2	2.1
Gloeobacter violaceus	3.4	3.6	3.5	2.0	2.0	1.3	1.3
Granulibacter bethesdensis	2.0	2.0	2.0	2.1	2.1	2.1	2.2
Haloarcula marismortui	2.1	2.0	2.0	1.6	1.6	1.6	1.6
Halobacterium sp-plasmid pNRC100	2.4	2.8	2.5	1.7	1.5	1.6	1.5
Legionella pneumophila	2.4	2.6	2.9	2.3	2.2	1.6	1.6
Legionella pneumophila-Philadelphia 1	2.5	3.8	3.2	4.1	3.8	4.1	4.1
Methanosarcina acetivorans	2.9	2.9	2.8	1.9	1.9	1.5	1.4
Methylococcus capsulatus	2.2	3.1	2.9	3.6	3.4	2.8	2.7
Nitrosospira multiformis	3.1	4.4	4.2	5.4	5.1	3.5	3.5
Photobacterium profundum	1.7	1.9	1.8	1.8	1.8	1.9	1.8
Pseudomonas syringae	2.1	1.9	1.9	1.6	1.6	1.2	1.2
Pyrococcus furiosus	1.7	1.7	1.6	1.8	1.7	1.8	1.8
Ralstonia solanacearum	2.4	1.4	1.4	1.2	1.2	1.1	1.1
Rhodopirellula baltica	2.2	2.6	2.5	2.4	2.4	2.0	2.0
Roseobacter denitrificans	2.3	3.3	3.1	3.8	3.8	3.6	3.5
Salinibacter ruber	3.0	2.1	2.0	1.1	1.0	1.0	1.0
Shewanella oneidensis	2.0	1.3	1.3	1.4	1.4	1.3	1.3
Sulfolobus solfataricus	2.1	1.6	1.5	1.3	1.3	1.2	1.2

4.2 Tests in Eukaryotic Short Sequences

We also applied the sequence graph based approach to paired end sequences from the genome of the plant *Beta vulgaris*. The aim of the analysis presented in this section is to provide an overview of the sequence graph for this kind of read set, test the scalability of sequence graphs when dealing with large amounts of data, and identify known repeat families in the components isolated by the sequence graph approach.

The reads we used are short reads obtained with the 454 sequencing method. This approach keeps paired ends by physically binding them with the help of a *linker*, as shown in Figure 4.3. A linker is a small DNA molecule of known sequence which is bound to both extremities of a read, creating a circular DNA molecule like the one shown in Figure 4.3. The circular DNA is cleaved before sequencing, resulting into two linear DNA molecules: one containing the central part of the read, which is discarded, and one containing the read extremities bound by the linker, which is sequenced. Linker sequences remain during the whole sequencing process. As a result, reads obtained by this method contain a linker sequence in the middle, which has to

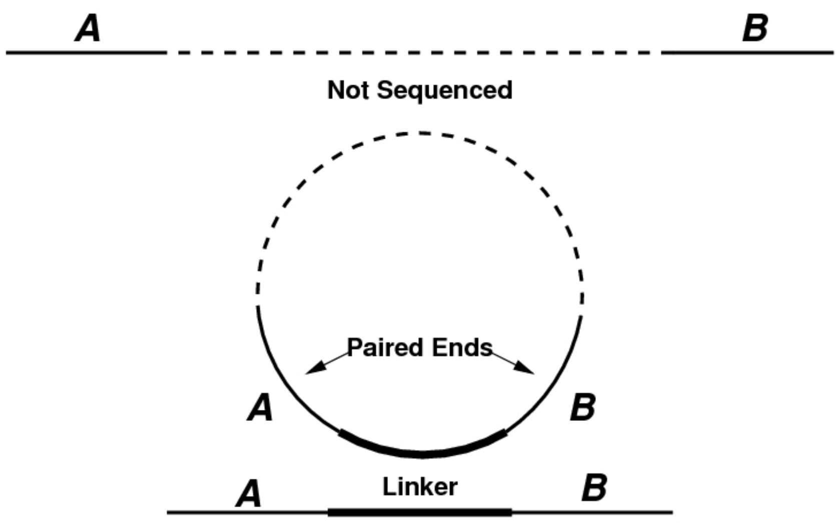

Figure 4.3: Sequencing 454 paired ends. Both ends A and B of a read (top) are bound together by a small sequence, the *linker*, which holds the pair together after the cleavage of the central part of the read (dashed line). Once the central part is removed, both read ends and the linker are sequenced.

be removed.

The linker is not the only artifact in 454 paired end reads. A short prefix $TCAG$ is also included in every read, and has to be removed before analysis. We trimmed all reads before the analysis, removing sequences included during the sequencing process and low quality regions. Details about the trimming procedure are given in the next section and the results of applying the family identification approach to this data are presented in Section 4.2.2.

4.2.1 Trimming 454 Paired End Sequences

As mentioned in the previous chapter, paired end 454-sequences have three main artifacts: TCAG prefixes, linker sequences, and low quality regions. Since the read region containing the prefix and the linker do not overlap, we removed these artifacts first. After the linker removal we identified and removed low quality regions in each of the read extremities. More details about the identification and removal of each artifact are given below:

TCAG Prefixes: TCAG prefixes are easy to identify: if present, they are always in the first 4 positions in the sequence. Each read prefix was verified, and TCAG prefixes were trimmed.

Linker sequences: Linker sequences are well known, but since they are larger than TCAG prefixes, low quality bases and sequencing errors prohibit the trimming by simple exact sequence comparison, as we do with the prefixes. For the removal of linkers we first align the linker with the read sequence, without penalizing gaps in prefixes and suffixes, and trim the read region aligned to the linker. To avoid randomly cutting reads that don't have the linker sequence, trimming is only done if the number of mismatches and indel operations in the aligned region is smaller than 10% of the alignment length.

Low quality sequences: The linker sequence is usually in the center of the read. Therefore removing it usually divides the read in two parts. Each read part obtained after the linker removal was checked for low quality regions. During the low quality trimming we focus on bases with quality value of at least a given value τ. The quality trimming is done in three phases:

1. **Quality conversion.** We subtract τ from each quality value, so that low quality bases get a negative value.
2. **High quality region identification.** After converting the quality values, we identify the region in the read that maximizes the sum of base qualities. In case of ties, we choose the longest region closest to the read prefix. The identification of all maximal scoring regions in a read can be done in linear time (39).
3. **Low quality trimming.** Once the best region is identified, we trim the read prefix and suffix not belonging to it.

4.2.2 A Low Coverage Eukaryotic Sequence Graph

A total of 2,884,945 reads with average length 99.87 remained after trimming and discarding sequences shorter than 19 bases. This amount is both for de Bruijn and sequence graphs too big. The results presented in this section come from the analyses of a 19-dimensional sequence graph built with 200,000 randomly chosen sequences from one of the eight files which form this data-set. The main problem with short read data is not the genome size, but the amount of sequences. Notice that the beginning and the end of sequences usually forces either the creation of a vertex or the cut of an existing one, therefore inserting sequences in sequence graphs creates new vertices and enlarges the sequence sets both in new and in pre-existing vertices.

We started the analysis by creating a 19-dimensional sequence graph and inserting the 200,000 sequences into it. The resulting graph was post-processed with the following operations:

Merge. We first merged all possible pairs of neighbor vertices. For this merging we only considered the in-degree and out-degree of the vertices and the fact that merged vertices must be either both inside or both outside a repeated region. The sequence set criterion[1] was ignored in this merging. At the end, 65,066 vertices were merged in this post-processing phase.

Small Components. As defined in Section 3.2, small components have a few non-repetitive vertices connected to a short repetitive vertex. We eliminated all such components in which the repetitive vertex was shorter than 38 bases, removing 4,960 vertices.

Singletons. Singletons are non-repetitive vertices that are not connected to any other vertex. We identified and removed 45,501 of these vertices.

Combine. We combined parallel vertices using a combine factor $\tau = 2$. This combine operation was applied 286,663 times.

Removal of Long Non-Repetitive Vertices. We deleted all non-repetitive vertices with length greater than 150 bases. This phase removed 17,921 vertices from the graph.

Final Merging. After the first post-processing phases, there may be pairs of neighbor vertices for which merging was not allowed in the first phase, but is allowed now. In fact, 348,018 vertices could be merged after the previous combinations and vertex removals.

The remaining graph has 13,823 components. We ignored the 8,237 components with less than 9 vertices and focused on components with a number of vertices between 9

[1]In Chapter 2, the sequence set in the merged vertices must be identical.

and 120. Only one of the components has more than 120 vertices. This component, with 1,348,038 vertices, is by far the biggest component in the graph. Components like this are commonly found in sequence graphs even after the post-processing vertex removals and usually contain sequences of different repeat families that could not be separated by the method. In fact, we marked in the sequence graph the tuples found in known repetitive elements of the *Beta vulgaris* genome[2], and identified in this component tuples from all different classes of repeat families in the database.

From the remaining 5,585 components we discarded the 167 components which were not acyclic and 26 components where we found tuples of known *Beta vulgaris* repeat families. The sequences corresponding to all possible paths in the 5,392 remaining acyclic components were stored in individual files and compared against a database of known plant repetitive elements. For the database search we used the BLAST[3] alignment tool. We selected the alignments with e-value smaller than 0.01, focusing only on database hits that can be really homologous to the graph sequences. Significant hits to a repeat database entries were found in sequences of 241 components.

4.3 Conclusions

In this chapter we presented an operation called *combine*, which is able to simplify sequence graphs and avoid the deletion of non repetitive vertices inside repetitive regions. Tests in bacterial genomes show that the de Bruijn based method for repeat family identification is able to recover more information using less genome information when using this operation (results on Table 4.1). We also notice that the gain is limited to small values for the combine factor, since less restrictive combinations tend to cluster unrelated sequences and facilitate the occurrence of components containing different families (results on Table 4.2). Finally we conclude that the optimal combine factor value, as well as the effectiveness of this operation, is genome dependent (results on Table 4.3).

After applying the repeat identification method to short reads of an eukaryotic genome, we realize that sequence graphs, like de Bruijn graphs, have problems when scaling

[2]This database of known repeat families was provide by the Chair of Genome Research of the Bielefeld University's Department of Biology

[3]http://blast.ncbi.nlm.nih.gov/

from prokaryotic to eukaryotic genomes. To begin with, the number of reads that can be fitted in a sequence graph is small in comparison to the amount of data that modern short read technologies can produce. Moreover, combining parallel vertices and deleting long single vertices is still not enough to separate most of the repetitive sequences, what is shown by the largest graph component in Section 4.2.2. However, the amount of directed acyclic components found outside this component shows that the sequence graph based repeat family identification can be used as a starting point, helping biologist in finding the first members of a repeat family. Here the low coverage forced by the graph size supports the argument that most of the directed acyclic components correspond to repeat families, since the low coverage guarantees a low probability of having two copies of the same genome locus in the dataset. The percentage of components with a repeat database hit, 4.47%, though low is consistent with the results of other approaches for finding repeats using the same dataset.

Chapter 5

De Bruijn Subgraphs and Alternative Splicing

In the beginning of this century, the humanity faced a quite sensitive existential question: in terms of number of genes, we may be just a little bit more complex than a worm or a fruit fly, and even simpler than rice. The first predicted number of genes in the human genome, 100,000 genes, was far away from reality. The media tried to rise a little bit the disappointing more realistic prediction of 30,000 genes, suggesting 40,000 (42), but the actual number is not supposed to be much greater than 27,000.

The idea that all the internal processes in a clearly more complex organism may be controlled by a much smaller pool of genes seems to be a huge paradox, but the explanation for this apparent mystery can be literally found in Einstein's brain: like in his small brain (15; 1), the structural complexity of our genome is much more important than its size. At least in the case of our genome, it is easy to understand where complexity helps: a modification of RNA molecules, called *splicing*, can create different variants of a single gene.

For long time, we believed that the whole process of creating a protein from the genomic code consisted of two steps:

Transcription, when the genomic DNA is copied in the form of an RNA molecule called *messenger RNA*, or shortly *mRNA*.

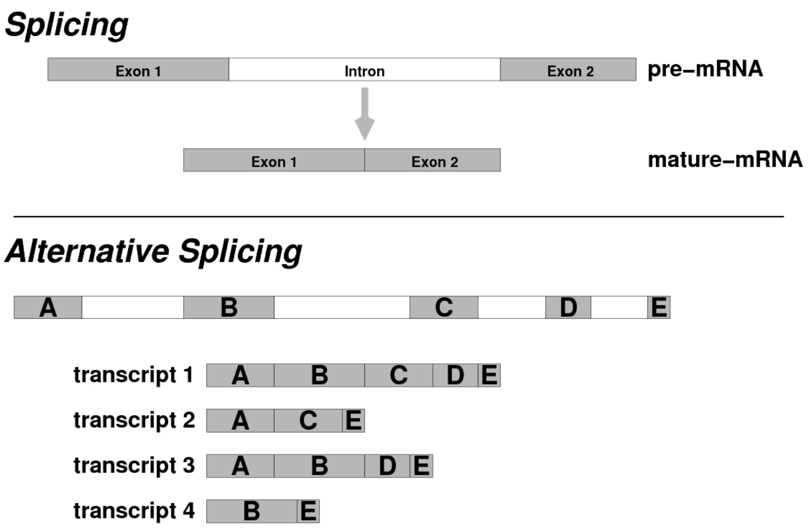

Figure 5.1: On the top, the schematic representation of the process of splicing: the intron in the pre-mRNA is spliced out, and the resulting mature mRNA is composed only by the remaining exons. On the bottom, a pre-mRNA with five exons and four of the possible transcripts. In *transcript 1*, all the exons are preserved. Exons "B" and "D" are spliced out together with their adjacent introns in *transcript 2*; the same happens with exon "C" in *transcript 3*, and exons "A", "C", and "D" in *transcript 4*.

Translation, when the mRNA molecule is "decoded" by an RNA-protein complex called *ribosome*, which is able to translate the genetic code carried by the mRNA into a protein.

In 1977, Berget and colleagues (5) observed an mRNA that loses a piece at some moment between transcription and translation. The process of losing parts was called *splicing*, and is schematically shown in the top part of Figure 5.1. The lost parts of the mRNA molecule are called *introns*, while the remaining parts are called *exons*. The mRNA molecule before splicing is called *pre-mRNA*, while the spliced molecule is called *mature mRNA*.

Later analyses showed that splicing is a frequent post-transcriptional modification. Moreover, many mRNA molecules have two or more introns, which gives to them an interesting combinatorial property: eventually, some exons may be discarded together with their surrounding introns, resulting in different mature mRNA variants from the same original copy. This process is known as *alternative splicing*, and the resulting mature mRNA molecules are also called *transcripts*. (See bottom part of Figure 5.1)

Notice that an mRNA molecule with n exons can give rise to up to $2^n - 1$ different transcripts. Therefore, the first expected diversity of 100,000 transcripts in the human genome can be achieved with less than 1,000 genes, each of them with 7 exons. Nevertheless, in nature, not all mRNA molecules are spliced; those who are may not have alternative transcripts; and those who have are seldom found in every possible variant. By any means, splicing explains how 27,000 coding regions can be enough to produce around 100,000 different kinds of proteins.

5.1 The Splicing Graph

The usual string representation of a set of transcripts does not provide a good visualization of the correlations between exons in the different transcripts. In order to represent sets of transcripts in a more informative way, Heber and colleagues defined the *splicing graph* (21). Their informal splicing graph definition does not always match the output of their algorithm for the graph construction. In this section we give a formal definition for splicing graphs and argue why the graph they construct is not the graph they define.

Definition 5.1. Splicing Graph. *Let $S = \{s_1, \dots s_{|S|}\}$ be a set of transcripts of a given gene of interest. Let p be the position in the genome where the gene of interest begins. Each nucleotide n in each transcript of S is uniquely identified by its* genome position *$g(n) = p' - p$, where p' is the nucleotide position in the genome. Each transcript $s \in S$ may be completely described by the set P_s of its bases' genome positions. Let x be a genome position in P_s, the* successor[1] *of x in s is defined as:*

$$\text{successor}(x, s) = \min\left(y \in P_s \mid y > x\right).$$

We also define the set of neighbors

$$N_s = \{(u, \text{successor}(u, s)) \mid u \in P_s \cup \{-\infty\}\}.$$

The splicing graph *for a collection of transcripts S, $\mathcal{S}(S) = (V, E)$, is the directed graph with:*

$$V = \bigcup_{s \in S} P_s, E = \bigcup_{s \in S} N_s.$$

[1]For the definition of successor we consider $\min(\emptyset) = \infty$.

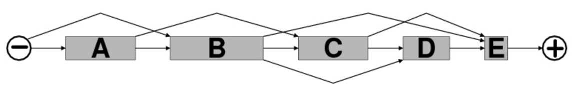

Figure 5.2: Splicing graph for the four alternative transcripts in Figure 5.1.

In words, $\mathcal{S}(S)$ is the graph where each position in S is a vertex, and two vertices are connected if they represent neighboring positions in at least one of the transcripts. Like de Bruijn subgraphs, splicing graphs usually have several induced paths, which can be collapsed into single nodes, resulting in a more compact representation. Figure 5.2 shows the compact representation for the transcripts in Figure 5.1.

An essential difference between de Bruijn subgraphs and splicing graphs is that, in opposite to de Bruijn subgraphs, splicing graphs are by definition acyclic: by ordering the vertices in increasing order of genome position we get a topological sorting of the vertices. The algorithm for constructing splicing graphs presented by Heber and colleagues actually builds a de Bruijn subgraph for the collection of transcripts. Although both graphs may be isomorphic in many cases, for genes which have exactly duplicated regions, the associated de Buijn subgraph is cyclic. Since these cases are rare, we use the term *splicing graph* as a synonym for the associated de Bruijn subgraph for a set of transcripts.

5.2 Clustering Transcripts

The set of all transcripts in a population of cells is called its *transcriptome*. Transcriptomes change over an organism's lifetime according to many different factors, like developmental state, or changes in the environment. Genomes, in contrast, usually remain intact during their whole lifetime. Until recently, transcriptomes were studied through *EST* sequencing projects. ESTs, or *expressed sequence tags*, are usually incomplete copies of transcripts, which are obtained by replicating the mRNA into a DNA form called *complementary DNA* (*cDNA*), binding the cDNA to small artificial bacterial chromosomes called *vectors*, inserting the vectors into bacteria, and finally amplifying and sequencing them. There are many problems in studying transcriptomes through ESTs. We name a few:

- The preparation of clone libraries for EST projects is a hard and time consuming work. The resulting libraries represent a limited number of mRNA molecules, and even in normalized libraries, highly abundant transcripts are over-represented, whereas rare transcripts may be missing.
- The sequencing reactions can only start in positions where the cDNA molecule was inserted into the vector. As a result, only the extremities of cDNA molecules can be sequenced, and the whole cDNA sequence is only discovered if the sequences of both extremities are so long that they overlap. Therefore, it is unlikely that the sequence of long transcripts will be entirely obtained without extra effort.
- Since the sequencing procedure is bacteria-dependent, transcripts that eventually cause bacteria death cannot be sampled.

With the advance of high throughput sequencing methods and the consequent abandon of bacterial amplification, there is hope for a better sampling of transcriptomes. A question that arises with these data is if the de Bruijn subgraph for transcripts may be used for more than displaying alternative splicing in a user friendly manner. Heber and colleagues (21) limit their studies to the construction of splicing graphs for a known set of splicing variants. They observed that the known variants are represented by paths in the splicing graph, and that the long induced paths in the graph correspond to exons. However, discovering new sets of splicing variants from sets of reads is a more interesting and useful task. In fact, Lacroix and colleagues (28), who developed a method for counting the abundance of transcripts based on splicing graphs, mentioned[2] that one of the problems they face is exactly the splicing graph construction without previous knowledge of the different transcript groups.

In Chapter 3 we showed that using a de Bruijn subgraph for separating repeat families is possible, but difficult even in simple genomes. In this section we investigate the possibility of using de Bruijn subgraphs for separating transcripts by gene. In a more formal formulation, we are interested in solving the following problem:

Definition 5.2. Transcript Clustering Problem. *Given a set of transcripts $S = \{s_1, \ldots, s_{|S|}\}$, find a function*

$$\varphi : S \mapsto \mathbb{N}$$

[2]Personal communication, F. Lacroix, 8th Workshop on Algorithms in Bioinformatics (WABI), Karlsruhe, Germany, 2008.

such that for every $s, t \in S$, $\varphi(s) = \varphi(t)$ if and only if s and t are transcripts of the same gene.

We devised an approach for solving the Transcript Clustering Problem which is similar to the one for finding repeat families presented in Chapter 3. For the Transcript Clustering Problem we can take advantage of several transcriptome data peculiarities:

- Unlike members of a repeat families, which are connected by non-repetitive sequences, transcripts are small individual molecules.
- If the same exon is found in many transcripts, all the variants have exactly the same copy of the exon (apart from sequencing errors).
- Exons are usually a few tens of bases long. This allows the use of underlying sequence graphs of higher dimension. As a result, clustering sequences by chance is unlikely.

Notice that combining or deleting nodes is not necessary. We simply build the sequence graph for the set of transcripts, identify the connected components, arbitrarily enumerate them, and define the function φ by mapping each sequence to the number given to the component that represents it. The procedure is summarized in Algorithm 7.

Algorithm 7 Transcript Clustering Problem

```
 1: function TranscriptClustering($S$)
 2:     Build the sequence graph $\mathcal{S}(S)$
 3:     Find the connected components in $\mathcal{S}(S)$
 4:     Arbitrarily enumerate the connected components in $\mathcal{S}(S)$
 5:     for all $s \in S$ do
 6:         Let $\kappa$ be the number of the component representing $s$ in $\mathcal{S}(S)$
 7:         Let $\varphi(s) = \kappa$
 8:     end for
 9:     return $\varphi$
10: end function
```

5.2.1 Proof of Concept

We applied Algorithm 7 to the set of human transcripts provided by the EBI's Alternative Splicing and Transcript Diversity database[3]. We used a 39-dimensional underlying sequence graph, which was formed by 21,190 connected components[4]. From these components, 9,820 contain only one transcript. Around 98.6% of the 11,370 components containing more than one transcript (11,215) are directed acyclic graphs, while less than 1.4% of them contain directed cycles.

An automatic analysis of these components is not possible due to the absence of a good annotation of the transcripts. However, some facts indicate that the resulting graph is a good approximation to the solution of the Transcript Clustering Problem for the given dataset:

Predominance of acyclic components. De Bruijn subgraphs associated to sets of splicing variants are not necessarily acyclic. On the other hand, in order to produce an associated graph with directed cycles in this case, the gene must have an exactly duplicated region of size at least 39, which is not common. Although the fact that less than 1% of the components contain directed cycles is not a proof that the graph yields the correct solution, the absence of cyclic components supports the hypothesis that the graph is a good approximation to the solution for the transcript clustering problem.

Clear separation between the input sequences and their reverse complements. The transcripts in the input file are all in the same orientation. Therefore, a component containing some input sequences in the given orientation and others in form of the reverse complement would indicate that sequences from different genes were clustered together. No such component was found in the graph.

Number of transcripts $\times$ number of components. In the solution for the Transcript Clustering Problem, the number of components must be exactly the number of genes in the genomic sequence, since every gene produced at least one transcript, and

[3]ASTD: `ftp://ftp.ebi.ac.uk/pub/databases/astd/current_release/human`, Release 1.1, build 5, file: 9606_transcripts.fa.gz, version from September 9, 2008.

[4]These numbers do not include components containing only reverse complements of sequences in the dataset. Although the graph was built with both transcript sequences and their reverse complements, all the graph components are composed either entirely by transcript sequences, or entirely by reverse complements

each splicing variant is correctly represented by the component of its corresponding coding sequence. In the sequence graph, the number of components, 21,190, is not far from the estimated number of genes in the human genome, around 27,000. The difference of nearly 6,000 components may be explained by an incomplete set of transcripts or by the presence of duplicated genes in the genome.

Identification of known splice variants by inspection. By manually comparing sequences of seven genes in the human chromosome number 1 to the nodes in the sequence graph, we could determine the content of seven of the components. The sequences in these components were used later for simulating transcriptomes, as described in Section 5.3.

5.3 Constructing Splicing Graphs from Short Reads

The results in Section 5.2.1 show that de Bruijn graphs are able to separate transcripts with common origin in different groups. However, the data used for constructing the graph in Section 5.2.1, namely a collection of transcripts of an organism, is not always available. And the method of choice for transcriptome sampling nowadays, namely EST sequencing, produces low quality samplings, as we discussed in the beginning of Section 5.2, and in most cases with incomplete gene sequences.

Of course one is only able to completely construct a graph with enough information about its vertices and edges. In the case of splicing graphs, the vertices are far easier to discover than the edges, since they represent the exons, which are long sequences, sometimes found in different transcripts with exactly the same sequence. Discovering the complete set of edges is more difficult, since some rare variants are poorly sampled, and only reads covering their splicing sites bring information about the edges.

New bacteria independent sequencing methods provide a more evenly distributed transcriptome sample. On the other hand, these methods produce reads which are much shorter than traditional Sanger reads. As a result, it is easier to obtain exon sequences than information about the exon order in different transcripts. This happens because many of the shorter reads are entirely contained in one exon, and do not provide any information about the exon neighborhood. The question we address in this section

Table 5.1: Genes with alternative transcripts used in the tests in Section 5.3. The column *Transc.* shows the number of transcripts.

Protein Description	**Gene Symbols**	**Transc.**
Alcohol Dehydrogenase	`AKR1A1, ALR, AKR1A1, ALDR1`	15
Aminopeptidase B	`RNPEP, APB, RNPEP`	13
Calcipressin-3	`DSCR1L2, DSCR1L2`	8
Cystein protease	`APG4C`	8
Dual specificity protein phosphatase 23	`DUSP23, VHZ, DUSP23, LDP3`	3
Potassium channel subfamily K	`KCNK1, TWIK1, HOHO1, KCNO1, KCNK1`	7
Serine/threonine-protein kinase PLK3	`PLK3, CNK, FNK, PRK, PLK3`	4

is to quantify the amount of reads necessary to (1) represent each gene by a unique component and (2) be able to identify the greatest number of alternative transcripts.

5.3.1 Datasets

In Section 5.2.1 we showed that a de Bruijn graph can separate large amounts of transcript sequences in components containing alternative transcripts of the same gene. We were able to identify seven of these components as known splicing variants. This section aims at investigating the amount of short reads needed for a trustful reconstruction of splicing graphs for a transcriptome. Notice, since we focus on coverage, that the size of the used datasets plays only a small role on the results. Therefore we opted for repeating the tests in many small artificial transcriptomes, instead of working with a smaller number of bigger datasets.

We used seven different sets of transcripts from the human genome obtained from the EBI's Alternative Splicing and Transcript Diversity database. The genes corresponding to the components in the sequence graph described in Section 5.2.1 were identified, and are presented in Table 5.1. We considered this set of 7 genes as a genome, and used the 58 different transcripts for the simulation of transcriptomes. To simulate the transcriptomes, we assigned to each transcript a multiplicity, corresponding to its abundance in the transcriptome. The abundances follow a Poisson distribution with mean 8. For each artificial transcriptome we created a set of reads with coverage 0.5, 2, 4, 8, and 16 times.

The coverage is always measured by means of the number of tuples of a given size in the transcriptome, counting with duplications. It means that, if the size of a collection of sequences S with tuple size d is given by

$$\text{size}(S, d) = \sum_{s \in S} \max\left(0, \text{length}(s) - d + 1\right),$$

a set of reads R covers the collection S with *coverage* at least c if

$$\text{size}(R, d) \geq c \cdot \text{size}(S, d).$$

Algorithm 8 Creation of Artificial Transcriptomes

```
 1: function SimulateReads($S, d, c, \lambda$)
 2:     $R \leftarrow \emptyset$
 3:     repeat
 4:         Generate $l \sim \text{Poisson}(\lambda)$
 5:         Uniformly choose $s \in S$
 6:         Uniformly choose $i \in \{1, \ldots, \text{length}(s) - d + 1\}$
 7:         $R \leftarrow R \cup \{s[i, \ldots, i + l]\}$
 8:     until $\text{size}(R, d) \geq c \cdot \text{size}(S, d)$
 9:     return $R$
10: end function
```

We created in total 50 different transcriptomes, and the corresponding sets of reads with the previously given coverage values. The reads are perfect copies of transcript sequences, and have Poisson distributed lengths with average 50. The tuple size used was 29. The creation of artificial read-sets is described by Algorithm 8. We built a sequence graph for each set of reads, and measured both the number of components for each group of transcripts, and the number of transcripts represented in the resulting sequence graph.

5.3.2 Results and Discussion

The graphs presented in Figures 5.3 and 5.4 summarize the results for the 50 different simulated transcriptomes. Figure 5.3 shows the average number of connected components representing variants of the same genes in the sequence graph. For small coverage values, the information obtained from short reads is not enough to cluster the transcript variants in single connected components. In these cases, alternative

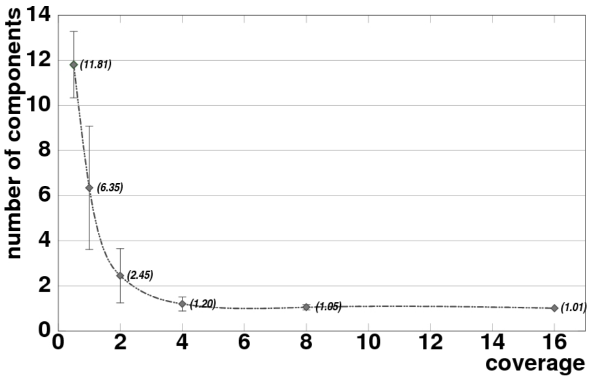

Figure 5.3: Average number of connected components per gene.

transcripts of some genes are sometimes represented in more than one component. On the other hand, the average number of components by gene decays fast as we increase the read set coverage. For coverage values greater than 4 times, most of the genes with alternative transcripts are clustered in exactly one component.

Having the gene represented by a single component is necessary, but not sufficient for finding all possible splicing variants in a transcriptome. For representing all variants, the graph must represent the exons in the sets of transcripts as vertices. Moreover, the read set must be big enough to cover all the splicing sites, otherwise the corresponding graph will miss edges, and some variants will not be discovered.

The data in Figure 5.3 shows that 4 times coverage is enough to get the majority of the exons represented by the sequence graphs. Notice that exon sequences can be shared by many different variants, which increase the chances of cover them. On the other hand, some exon neighborhoods may be found only in a few variants. As a result, even if the information in a read set is enough to find the splicing graph vertices, the edges found only in rare variants may be only observed in huge read sets. This is shown in Figure 5.4, where we see the percentage of sequences in the simulated transcriptome that are also represented by the sequence graph in different read sets. Although we may expect to observe more than 90% of the transcripts with coverage values as small as 6 times, we were never able to observe 100% of the transcripts even with 16 times coverage.

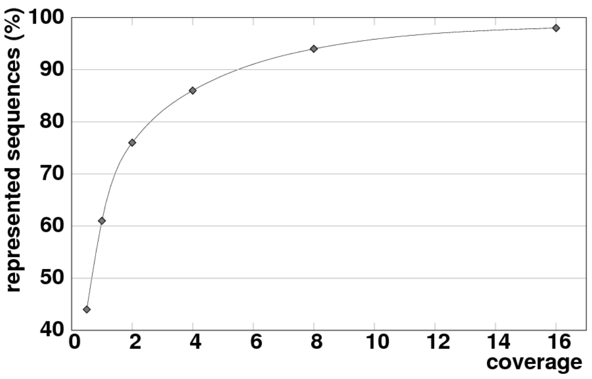

Figure 5.4: Percent of transcripts represented in the de Bruijn subgraphs for different coverage values of the transcriptome.

Part II

Fault Tolerant Interval Group Testing

Chapter 6

Group Testing

Group testing was born in the last years of the second world war to be used in a biomedical application. The exact place of birth was the Price Administration of the US Government, Research Division, Price Statistics Branch. The approach has both an official father, Robert Dorfman, who published the first report on the subject; and a putative father, David Rosenblatt, who worked in the same research group, and claims to have suggested the method's basic principle. A brief history of Group Testing, including briefs of Dorfman and Rosenblatt explaining their viewpoints about the origin of the approach, can be found in (17). By any means, what called the attention of the research group to the subject was the number of identical clinical tests performed in order to identify a few cases of syphilis in the US troops. The Price Administration Office observed that great amounts of money were invested in testing soldiers who were not infected, what was for sure good news to the majority of the soldiers, but a waste of public funds, since the aim of testing the troops was to find the infected individuals. The Research Division came with the idea of performing the analysis not in individual samples, but in mixtures of them.

The principle of group testing relies on the assumption that the number of infected individuals to be discovered, called from now on the *positives*, is much smaller than the total number of individuals. When this is true, the probability of picking an infected sample by chance is small. In fact, it is so small that the probability of having at least one positive in a reasonably large group of samples is also small. In this case, samples may be grouped, and tests can be performed not on individual samples, but on a mixture of them. This may sometimes reduce the resolution of the results: since we test groups instead of single samples, it may be the case that we are not able to distinguish infected (positive) from clean (negative) samples in groups whose mixture gives a positive result. On the other hand, each negative result for a mixture means that no sample used to produce the mixture is contaminated. In this case, we save many individual tests by testing the samples together.

The resolution of the tests' outcomes depends on the way the groups are designed. We can think about creating groups in such a way that a single round of tests always gives enough information to exactly identifying the positives. This strategy, where only a single round of tests is enough to pinpoint all positives, is called *non-adaptive* group testing. The alternative to non-adaptive group testing are approaches where tests are organized in batches and done in rounds called *stages*. In these cases, the tests to be performed in a stage depend on the output of the tests in the previous stage. This is called *adaptive* group testing.

Following the original motivation for group testing strategies, when optimizing group testing approaches, the main aim is the minimization of the number of tests needed for identifying the positives. Of course more important than knowing this number is to know how to group the samples in order to always identify the positives in the minimum number of tests. A set of rules for creating the groups in each stage is called an *algorithm*.

Although the main focus of group testing studies is the reduction of the amount of tests, in practice a series of other limitations appear. These limitations motivated the study of variants of the original problem of designing the groups. For instance, although the adaptive approach usually needs a smaller number of tests, the number of stages used is sometimes restricted by the time and cost of performing the tests in a stage. Since we are mostly speaking about physical or chemical tests, a new stage means postponing the final results in hours or even days. Moreover, tests sometimes cannot be automatically performed, and their parallelization is limited by the resources of the laboratory performing them. Therefore there is a preference for non-adaptive approaches (7), and even when adaptive group testing is used, the number of stages is seldom greater than two.

Another important practical obstacle for the approach is the amount of material in a sample. Particularly in applications for disease diagnoses, there is a maximum number of groups in which a sample may be contained simply because the samples are of finite size.

Finally, groups may not be arbitrarily big. There are cases where the group size is limited by physicochemical properties. In the original application for discovering patients infected with syphilis, in groups with more than 8 samples a single positive sample may be so diluted that the test output is not anymore accurate. This was the main reason why the group test approach for identifying syphilis infected soldiers was not used during the second war. However, the approach found recently its original purpose again, and is being successfully applied in tests for HIV identification. Actually, with respect to HIV tests, group testing found even more applications: Zenios and Wein (45) used the fact that the output of HIV tests is not binary, but continuous, and devised a method for estimating the virus prevalence, or the amount of contaminated people, in a population without the necessity of exactly identifying the positives. This does not only save tests, but also provides an important information while keeping the identity of infected people in secret.

6.1 Formal Definition for the Group Testing Problem

A typical group testing problem consists of three elements: a set $\mathcal{O}$ of exactly n known objects, which are the individuals to be tested; a set $P \subseteq \mathcal{O}$ of *positives*, which are the elements we want to identify; and a family $\mathcal{Q}$ of tests or *queries*, which are subsets of $\mathcal{O}$. Each set $Q \in \mathcal{Q}$ provides information about the P through an oracle that correctly answers the question

$$P \cap Q = \emptyset \; ?$$

An *algorithm* for finding the set of positives is a series of rules for grouping the objects in each stage, in one of the possible ways according to $\mathcal{Q}$. The efficiency of an algorithm is measured by the number of tests needed to identify the positives. Assuming the number of possible stages unlimited, we denote with $\mathcal{N}(n, p)$ the minimum number of tests necessary to identify p positives among n elements. The efficiency of group test algorithms is measured in terms of $\mathcal{N}(n, p)$.

A lower bound for the efficiency of group testing algorithms can be obtained with a simple information theoretical argument: if there are exactly p positives in our set of n samples, then a group testing algorithm needs to differentiate between the $\binom{n}{p}$ possible sets of positives. The outcome of each test is binary: each of them is either positive or negative. Therefore we can define, for each algorithm outcome, a binary vector where each position corresponds to a test in the algorithm. If the algorithm uses less than $\log \binom{n}{p} = \Omega(p \log \frac{n}{p})$ tests, the number of different outcomes is less than $2^{\log \binom{n}{p}} = \binom{n}{p}$, which is the number of possible sets of positives. As a result, there is at least one outcome for which the algorithm cannot correctly decide between two or more sets of positives. This lower bound of $\Omega(p \log \frac{n}{p})$ is attained by fully adaptive strategies.

A larger lower bound for non adaptive algorithms is obtained through the equivalency between superimposed codes and non-adaptive algorithms. Superimposed codes are sets of binary vectors with special properties, which are presented in Section 6.2.1. If the vectors in a superimposed code have length n, they can be interpreted as tests, where a sample is in a group if and only if the value of its corresponding position in the binary vector is equals to 1. From the correspondence between superimposed codes and group testing, we know that the minimum number of tests in a non-adaptive algorithm is equal to the minimum number of vectors in such a code, which is bound by $\Omega(\frac{p^2}{\log p} \log n)$. The best known non-adaptive strategy needs $O(p^2 \log n)$ tests.

If the number of stages is at most s, we denote the minimum number of tests necessary to identify the positives in the worst case with $\mathcal{N}(n, p, s)$. A variant of the classical group testing approach considers the case when the oracle is not perfect, but may erroneously answer at most e questions in the whole process. In this case, the minimum number of tests necessary to identify the positives is denoted by $\mathcal{N}(n, p, s, e)$.

6.2 Group Testing and Genome Research

The huge amount of data a genome researcher needs to face in his everyday tasks does not need to be mentioned anymore. Even simple tasks in a genome research project, like comparing two genomes, involve thousands of individuals tests. Therefore it is not surprising that group testing approaches are being used for improving the efficiency of tasks like DNA library screening, physical mapping, gene detection, assembly finishing, and many others (29). Group testing approaches in biology are non-adaptive in most cases. The groups are in most cases called *pools*, therefore the approach is commonly called *pooling design*.

6.2.1 Library Screening

Libraries are sets of clones, which are copied parts of a DNA molecule. Very small chemically modified DNA molecules called *probes* may bind to clones, helping biologists to identify them. Wu and colleagues (43) studied the problem of verifying if all clones in a library hybridize at least one probe in a given set. For this purpose, they were interested in pooling clones so that it is possible to identify up to p positives using the minimum number of tests in a non-adaptive approach. A positive in this case is a clone (or pool) which hybridizes with a probe. Probes are chemically labeled, so that biologists can easily test for their presence.

For designing the pools, Wu and colleagues suggest the use of a binary matrix where columns represent the clones, and rows represent pools. The position (i, j) in this matrix has the value 1 if the pool i contains the clone j. When a clone is positive, all pools containing a 1 in its corresponding column are also positive. We call this set of pools covering a clone its *roof.* The roof for a clone j is therefore defined as

$$\text{roof}(j) = \{i \mid M_{ij} = 1\}.$$

The roof of a set of clones is the union of all individual roofs.

In order to uniquely identify each possible outcome containing up to p positives, we need to design our pool in such a way that each possible combination of up to p positives has a unique roof. A matrix with such a property is called *$\overline{p}$-separable.* The minimum number of pools in a pool design is obtained when the $\overline{p}$-separable binary matrix representing the pool has n columns and the minimum possible number of lines.

The problem of identifying the positive clones given a set of positive pools is called *decoding*, and is NP-Complete (43). The following theorem by Wu *et al.* (43) gives a hint on how to design pools that can be easily decoded:

Theorem 6.1 (Theorem 2 in Wu et. al (43))**.** *For a $\overline{p}$-separable matrix, the union of negative pools always contains at least $n - p - k + 1$ clones if and only if no p-union contains a k-union.*

A p-union is the union of the roofs of exactly p columns. When $k = 1$, no column contains a p-union, the union of negative sets contains at least $n - p$ objects, and decoding can be solved in polynomial time $O(n)$. The matrix is called p-disjunct in this case.

6.2.2 Protein-Protein Interaction

Because many functions in organisms are controlled through complex networks of interactions between different kinds of proteins, an efficient way of identifying pairs of interacting proteins is of great interest to biologists. Li and colleagues (29) formulated a group testing approach using complete bipartite graphs for testing protein-protein interaction.

Let $K_{A,B}$ be a complete bipartite graph where each vertex represents a protein, and each edge an interaction. The proteins on partition A are called *baits*, and the ones on partition B are called *prays*. An edge is called positive if the two proteins it connects interact; otherwise it is called negative. Let $C \subseteq A$ and $D \subseteq B$. We say that the test (C, D) is positive if there is a positive edge (c, d), with $c \in C$, and $d \in D$.

Here, the same pooling matrix described in Section 6.2.1 is used. However the definition of disjunction is modified, so that it reflects the fact that edges are tested, instead of vertices. Let H be a bipartite graph, we say that a pool design matrix M is *$p(H)$-disjunct* if for any $p+1$ edges $e_0, e_1, \ldots, e_p$ of H, there exists a row indicating that a pool contains e_0, but not $e_1, \ldots, e_p$. In (29), two constructions that provide $p(H)$-disjunct matrices are presented.

6.2.3 Finding the Borders of Assembly Gaps

A genome can be seen as a set of *chromosomes*, which are very long strings on the small alphabet $\Sigma = \{A, C, G, T\}$. Due to technological limitations, biologists have only access to small substrings of the chromosomes, and need to put these small pieces together in order to obtain the whole genome. Usually, the amount of sequence data obtained in a genome sequencing project is not sufficient to uncover the whole genome sequence, which results in a set of large strings corresponding to the uncovered parts of the chromosomes. Therefore after a series of sequencing experiments, the chromosomes may be divided into two kinds of substrings: the parts uncovered by the sequencing data, called *contigs*, and the parts for which no sequence information is available, called *gaps*. By definition, contigs and gaps alternate in the chromosomes: contigs are flanked by at most two gaps, which are flanked by at most two contigs.

If a gap is small enough, the extremities of its flanking contigs can be used as starting point for a reaction called *polymerase chain reaction*, or simply PCR, which uses the original DNA molecule of the chromosomes to create many copies of the gap. These copies can be used to obtain the missing gap sequence, which is called *closing the gap*.

The problem is that, since we do not know the strings corresponding to the chromosomes, we cannot tell which pair of contigs flanks a gap. If we try to perform a PCR using the extremities of a wrong pair of contigs, we will never get a gap sequence. A partially known genome with c contigs has $2c$ extremities. Doing a PCR for each of the $\binom{2c}{2} - c$ possible pairs of contig extremities is infeasible. Fortunately when doing a PCR using many contig extremities, we get some result only if at least two of the extremities flank a gap. This variant where many extremities are used together in a single PCR is called *multiplex PCR*, and this is everything we need for a group testing approach.

Let G be a complete graph where the vertices are contig extremities. The edges corresponding to the pairs of gap flanking sequences form a hidden matching in G. Each unsuccessful PCR using many extremities reveals a set of vertices that contain no edges from the matching, whereas successful PCRs give hints about the location of the matching edges. By grouping the primers carefully it is possible to identify the correct pairs with less testing. Surprisingly in this case a not so careful choice is able to reach the best possible result. Alon and colleagues (3) proved that a completely randomized non-adaptive algorithm is able to identify the correct pairs of contig extremities using $O(n \log n)$ tests.

6.3 Testing with Inhibitors

Samples can be damaged or contaminated, and force the tests including them to always produce a negative result, even in the presence of positives. Samples that influence tests outcomes are called here *inhibitors*. In 1997, Farach and colleagues (18) introduced a variant of the group testing model where the search space contains three kinds of objects: positives, negatives, and inhibitors. They introduced a parameter r into the traditional model, which is an upper bound for the number of inhibitors in the search space, and devised a randomized algorithm able to identify the positives with $O((r+p)\log n)$ queries on average, assuming that $r+p \ll n$. One year later, de Bonis and Vaccaro (14) were able to devise an adaptive deterministic algorithm using $O((r^2+p)\log n)$ queries. This algorithm could be extended to a 3-stage algorithm that uses $O((r^2+p^2)\log n)$ tests.

In 2008, de Bonis (7) worked both with the model variant where p is the exact number of positives in $\mathcal{O}$, and the variant where this parameter is an upper bound for the amount of positives. She showed that any algorithm with any number of stages that finds up to p positives in the presence of at most r inhibitors is lower bounded by the length of a superimposed code of size n, which is $\Omega(\frac{r^2}{p\log r}\log n)$.

Chapter 7

Interval Group Testing

In the previous chapter we presented applications of traditional group testing to biological problems, with the typical set of objects and positives, and queries being any subset of the set of objects. In this chapter we present a variant of the group testing approach called *Interval Group Testing*, where the objects in the search space are linearly sorted, and only queries containing all elements inside an interval are allowed.

Interval Group Testing is the variant of the traditional group testing approach where the search space can be linearly sorted, and the only tests allowed are the tests containing all the objects in an interval. In this case, the search space, $\mathcal{O}$, can be represented by the set of the first n positive integers $[n] = \{1, 2, \ldots, n\}$, the set of positives P is an arbitrary subset of $\mathcal{O}$, and each query can be represented by an interval $[i, j]$. The queries are as usual binary tests asking "Is $P \cap \{i, i+1, \ldots, j\} \neq \emptyset$?", for some $1 \leq i \leq j \leq n$. The aim is to identify P with the minimum number of queries.

Group testing with interval tests also arises in a variety of domains, e.g., detecting holes in a gas pipe (17; 27), finding faulty links in an electrical or communication network, data gathering in sensor networks (23; 22; 24), just to mention a few. Our main motivation for the study of interval group testing comes from its application to the problem of determining exon-intron boundaries within a gene (32; 44). In a very simplified model, a gene is a collection of disjoint substrings within a long string representing the DNA molecule. These substrings, called *exons*, are separated by substrings called *introns*. The boundary point between an exon and an intron is called a *splice site*, because introns are spliced out between transcription and translation.

Determining the splice sites is an important task, e.g., when searching for mutations associated with a gene responsible for a disease.

In (44), a new experimental protocol is proposed that searches for the exons boundaries using group testing. This consists of selecting two positions in the cDNA, a copy of the original genomic DNA from which introns have been spliced out, and determining whether they are at the same distance as they were in the original genomic DNA string. If these distances do not coincide then at least one intron (and hence a splice site) must be present in the genomic DNA between the two selected positions. The advantages of splice site detection by distance measurements over sequence-based methods using, e.g., Hidden Markov Models are that this method works without expensive sequencing of genomic DNA and it gives the results directly from experiments, without relying on inference rules. The work (44) and the book (32) report about the experimental evaluation, on real data, of the algorithm ExonPCR that finds exon-intron boundaries within a gene. The authors of (44) only give a simple asymptotic analysis of their $\Theta(\log n)$-stage algorithm. In particular, they leave open the question whether there exist less obvious but more efficient query strategies for Interval Group Testing and, more importantly, algorithms able to cope with the technical limitation of the experiments, and particularly with errors. We remark that non-adaptive strategies are desirable in this context, in order to avoid long waiting periods necessary to prepare each experiment. However a totally non-adaptive algorithm (with $s = 1$) needs unreasonably many queries. Thus, the necessity arises to trade more stages for fewer queries, but without exceeding with stages. In (12) the first rigorous algorithmic study of the problem was presented, and for the case $s \leq 2$ a precise evaluation of $\mathcal{N}(n, p, s)$ was given. In (11) a sharper asymptotic estimation of $N(n, p, s)$ is found, which is optimal up to the constant of the main term in the case of large s. The necessity of dealing with errors in the tests had been already stated in the seminal papers (32; 44) and reaffirmed in the subsequent ones. However, to the best of our knowledge, ours are the first non-trivial results on this interesting variant of the problem.

7.1 Definitions and Notation

An instance of the problem is given by three non-negative integers n, p, s and a subset $P \subseteq O = \{1, 2, \ldots, n\}$, such that $|P| \leq p$. The set O is the search space and P is the set of *positive* objects that have to be identified. We assume that tests are arranged in *stages*: in each stage a certain number of tests is performed non-adaptively, while tests of a given stage can be determined depending on the outcomes of the tests in all previous stages.

For each value of the parameters n, p, s we want to determine the value of $\mathcal{N}(n, p, s)$, the worst-case number of tests that are necessary (and sufficient) to successfully identify all positives in a search space of cardinality n, under the hypothesis that the number of positives is at most p, and s-stage algorithms are used.

We use square brackets to denote intervals of integers in $[n]$. Then, for each $1 \leq i \leq j \leq n$, we denote by $[i, j]$ the set $\{i, i+1, \ldots, j\}$. Given an interval $\pi = [i, j]$, we denote its size by $|\pi|$, i.e., $|\pi| = j - i + 1$. By definition each query asks about the intersection of a given interval with the set of positive elements. Therefore, we identify a query with the interval it specifies. We say that a query $Q \equiv [i, j]$ covers an element $k \in [n]$ if and only if $k \in [i, j]$.

A query $Q \equiv [i, j]$ has two boundaries: the left, $(i - 1, i)$, and the right, $(j, j + 1)$. For the sake of definiteness, we assume that, for any $a \in [n]$, the query $[1, a]$ has left boundary $(0, 1)$, and the query $[a, n]$ has right boundary $(n, n + 1)$. A multiset of queries $\mathcal{Q}$ defines a set of boundaries $\mathcal{B}(\mathcal{Q}) = \{(i_1, i_1 + 1), (i_2, i_2 + 1), \ldots\}$, where $0 \leq i_k < i_{k+1} \leq n + 1$. Every interval $[i_k + 1, i_{k+1}]$ is called a *piece*. Because every query has two distinct boundaries, but two queries may share some boundaries, we have $|\mathcal{B}(\mathcal{Q})| \leq 2\,|\mathcal{Q}|$. A boundary B of a piece P is said to be *turned to* the piece if there is a query Q such that $P \subset Q$ and B is also a boundary of Q. A piece is called a *2-piece* if both its boundaries are turned to it. A piece that has only one of its boundaries turned to it is called a *1-piece*. If none of the boundaries is turned to the piece, it is called a *0-piece*. Figure 7.1 illustrate the definitions given so far.

Let π be a piece defined by the set $\mathcal{Q}$ of queries. We call *roof* of π, and denote it by $\mathcal{R}(\pi)$, the subset

$$\mathcal{R}(\pi) = \{Q \in \mathcal{Q} : \pi \subseteq Q\}\,.$$

The definition of roof can be extended to a set of pieces $\mathcal{P}$ as

$$\mathcal{R}(\mathcal{P}) = \bigcup_{\pi \in \mathcal{P}} \mathcal{R}(\pi).$$

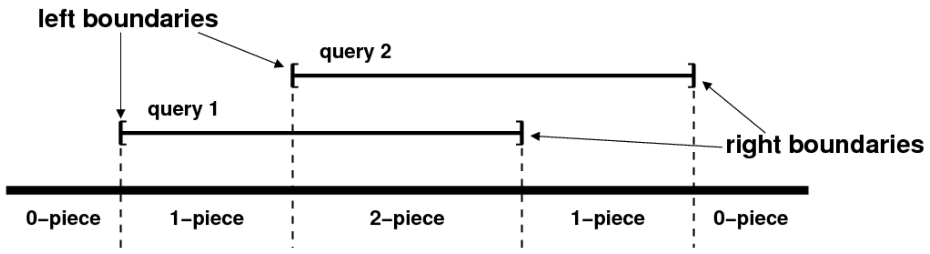

Figure 7.1: A set of two interval queries, which partition the set of objects into 5 pieces. The thicker line represents the set of objects.

Consider two pieces π_i and π_j. If $\mathcal{R}(\pi_i) \subset \mathcal{R}(\pi_j)$ then the piece π_i is called a *satellite* of piece π_j. There are 3 simple facts about satellite pieces:

- Every 1-piece is a satellite of some 2-piece.
- Every 0-piece contained in some query interval is a satellite of two 2-pieces, namely the next ones to both sides of b.
- If a 0-piece outside all query intervals exists, then it is a satellite of every 2-piece.

Moreover, if q is the number of queries in a stage and c_i is the number of i-pieces, we have we have $c_1 + 2c_2 \leq 2q$. Furthermore, $c_1 \leq q$ and $c_2 \geq 1$.

7.1.1 YES-sets

Let $\mathcal{Q}$ be an arbitrary set of interval queries. As soon as we start playing with possible answers to the queries in $\mathcal{Q}$, it becomes clear that not all combinations of positive and negative answers correspond to real situations. For instance, in Figure 7.2, answering Yes to all queries is not possible when looking for at most one positive: since the queries a, d, and e are disjoint, the positive would be at the same time in one of the pieces π_1 and π_2; and in one of the pieces π_4 and π_5; and in one of the pieces π_6 and π_7. Since a positive cannot be at the same time in three different intervals, this set of answers will never be observed when $p = 1$. The same happens for the case of at most 2 positives. If more than two positives are allowed, this pattern could be observed if the pieces π_2, π_4, and π_7 contain at least one positive. Another forbidden combination is the one where only the query b has a positive answer: since all pieces covered by

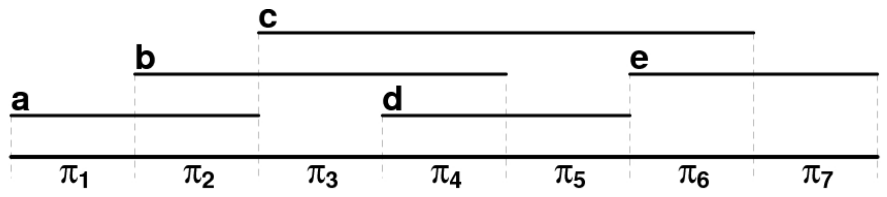

Figure 7.2: A query scheme with 5 queries dividing the search space in 7 pieces.

this query are also covered by the queries a, c, or d, a positive in the interval b would automatically force another positive answer.

A set $\mathcal{Y} \subseteq \mathcal{Q}$ of queries such that answering Yes to the queries in $\mathcal{Y}$ and No to all others corresponds to valid scenarios is called a YES-set for $\mathcal{Q}$. Formally speaking, a YES-set can be defined as follows:

Definition 7.1. *Let $\mathcal{Q}$ be a multiset of queries, and let $\mathcal{Y} \subseteq \mathcal{Q}$. If there is a set of pieces $\mathcal{P}$ such that $|\mathcal{P}| \leq p$ and $\bigcup_{\pi \in P} \mathcal{R}(\pi) = \mathcal{Y}$, then $\mathcal{Y}$ is a* YES-set *for $\mathcal{Q}$ in the case for p positives.*

A YES-set is called *specific* if the intersection of all its queries corresponds to a single piece, and the piece has at most one positive, otherwise it is called *unspecific.* More formally, a YES-set $\mathcal{Y} \subset \mathcal{Q}$ is specific if and only if there is a piece π of $\mathcal{Q}$, with $|\pi \cap P| \leq 1$, such that $\bigcap_{Q \in \mathcal{Y}} Q = \pi$.

7.1.2 A Simple Average Argument for 2-Stage Lower Bounds

Suppose we want to find at most p positives in a search space $\mathcal{O} = [n]$ in two stages. Let $\mathcal{Q}$ be the multiset of interval queries used in the first stage, and suppose that $\mathcal{Q}$ divides the search space in l pieces. There can be many different YES-sets for $\mathcal{Q}$, and positives are differently spread over the pieces in each YES-set. The analysis of the number of queries needed to uniquely identify the positives in a second stage for each YES-set gives us the opportunity to look for the YES-sets that force the greatest number of queries in total. These YES-sets correspond to the worst scenario faced when using the query scheme $\mathcal{Q}$, and the minimum (over all $\mathcal{Q}$) of the number of queries forced by such YES-sets corresponds exactly to $\mathcal{N}(n, p, 2)$.

Let $\mathcal{Y}$ be a YES-set for a query scheme. We call the *weight* of $\mathcal{Y}$ the number of queries needed to uncover the positives in the search space after observing $\mathcal{Y}$. It is difficult to characterize the heaviest YES-sets for a general query scheme. And actually we do not need to know how this YES-set looks like. All we need is the weight of this YES-set, or at least bounds for this number of queries. Consider a multiset of YES-sets, and let η be the average weight of the YES-sets in this multiset. Because η is smaller than or equal to the maximum weight in the multiset, it is also a lower bound for the weight of the heaviest YES-set in the multiset. Notice that the heavier the YES-sets in this multiset are, the closer the lower bound is to the real weight of the heaviest YES-set.

Of course the lower bound for a single query scheme is a quite poor information. Cicalese *et al.* proved bounds for the number of tests needed in many variants of the interval group testing approach (12; 11). In (12), they used an averaging argument to provide lower bounds for two-stage interval group testing strategies based on bounds for the non-adaptive case. Their bound for the 2-stage approach is based on averaging the weight of a multiset of YES-sets. And the beauty of this approach lies on the fact that no knowledge is needed about the weight of individual YES-sets. Cicalese *et al.* do not analyze YES-sets, but instead they concentrate on individual pieces, and the number of queries needed in each YES-set to look for positives inside the pieces in the second stage.

Suppose that the number of queries needed to reveal the positives inside a piece π_j when observing the YES-set $\mathcal{Y}_i$ is given by $w(\mathcal{Y}_i, \pi_j)\,|\pi_j|$, where $w(\mathcal{Y}_i, \pi_j)$, the piece *weight*, is a function of the YES-set and the piece, and $|\pi_j|$ is the size of the piece. If the total number of pieces in the query scheme is l, we may express the weight of the YES-set $\mathcal{Y}_i$ as:

$$w(\mathcal{Y}_i) = \sum_{j=1}^{l} w(\mathcal{Y}_i, \pi_j)\,|\pi_j|\,.$$

This means that the average weight in a multiset of m YES-sets is given by:

$$\begin{aligned}\eta &= \frac{\sum_{i=1}^{m}\sum_{j=1}^{l} w(\mathcal{Y}_i, \pi_j)\,|\pi_j|}{m} \\ &= \frac{\sum_{j=1}^{l}\left(|\pi_j|\sum_{i=1}^{m} w(\mathcal{Y}_i, \pi_j)\right)}{m}\end{aligned}$$

Suppose now that we know neither the piece sizes nor their individual weights, but we know that the sum of weights for every piece in all YES-sets is never smaller than a

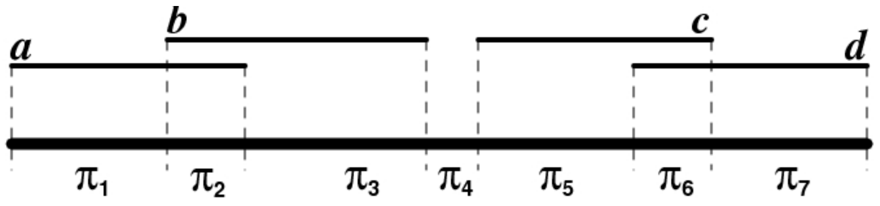

Figure 7.3: Pattern of four queries analysed in Section 7.1.2.

number r, that is:

$$\sum_{i=1}^{m} w(\mathcal{Y}_i, \pi_j) \geq r.$$

Moreover, we know that the total number of YES-sets is not larger than a number $k \geq m$. We have:

$$\begin{aligned}
\eta &= \frac{\sum_{j=1}^{l} \left(|\pi_j| \sum_{i=1}^{m} w(\mathcal{Y}_i, \pi_j)\right)}{m} \\
&\geq \frac{\sum_{j=1}^{l} \left(|\pi_j|\, r\right)}{m} \\
&\geq \sum_{j=1}^{l} |\pi_j| \frac{r}{k} \\
&= n\frac{r}{k}
\end{aligned}$$

This reasoning proves the following lemma, which was used in (12; 11) for proving lower bounds in 2-stage strategies, and can be used without modifications for finding lower bounds in fault tolerant interval group testing, as shown in the next chapter.

Lemma 7.2. *Consider a multiset of k (not necessarily distinct!)* YES*-sets, and for each $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, \ell$, let $w(\mathcal{Y}_i, \pi_j)$ be the weight of the jth piece in the* YES *vector associated to the ith* YES*-sets. If there exists an $r > 0$ such that for all $j = 1, 2, \ldots, \ell$, it holds that $\sum_{i=1}^{k} w(\mathcal{Y}_i, \pi_j) \geq r$, then an adversary can force at least $\frac{r}{k}n$ queries in the second stage.*

Using the Average Argument

We exemplify the use of the average argument by giving a lower bound for the number of queries used by an algorithm able to find up to 2 positives using the queries shown in Figure 7.3 in the first stage. We analyze 2 YES-sets: $\mathcal{Y}_1 = \{a, b\}$ and $\mathcal{Y}_2 = \{c, d\}$. In traditional interval group testing, the number of queries necessary to identify p positives in a search space of size n is given by $\mathcal{N}(n, 1, 1) = \lceil \frac{1}{2}n \rceil$, if $p = 1$, or $\mathcal{N}(n, p, 1) = n$, if $p \geq 2$ (12).

We start analyzing $\mathcal{Y}_1$. Notice that when this YES-set is observed we can have three situations: either both positives are in the piece π_2, or the two positives are separated in two of the pieces π_1, π_2, and π_3, or one positive is in piece π_2 and the other is in the uncovered piece π_4. Since each piece that may contain a positive has to be analyzed in the second stage with a strategy able to find the positives in any possible case, the piece π_2 has to be analyzed with a strategy able to find up to two positives, the pieces π_1, π_3, and π_4 with a strategy able to find up to one positive, and all other pieces may be ignored in the second stage. This means that this YES-set forces at least $\frac{1}{2}|\pi_1|$ queries in piece π_1, $|\pi_2|$ queries in piece π_2, $\frac{1}{2}|\pi_3|$ queries in piece π_3, and $\frac{1}{2}|\pi_4|$ queries in piece π_4 in the second stage. Moreover, the weight given from $\mathcal{Y}_1$ to each piece is given by:

$$\begin{aligned}
w(\mathcal{Y}_1, \pi_1) &= \frac{1}{2} \\
w(\mathcal{Y}_1, \pi_2) &= 1 \\
w(\mathcal{Y}_1, \pi_3) &= \frac{1}{2} \\
w(\mathcal{Y}_1, \pi_4) &= \frac{1}{2} \\
w(\mathcal{Y}_1, \pi_5) &= 0 \\
w(\mathcal{Y}_1, \pi_6) &= 0 \\
w(\mathcal{Y}_1, \pi_7) &= 0
\end{aligned}$$

Notice that a similar analysis gives also the weights:

$$\begin{aligned}
w(\mathcal{Y}_2, \pi_1) &= 0 \\
w(\mathcal{Y}_2, \pi_2) &= 0 \\
w(\mathcal{Y}_2, \pi_3) &= 0 \\
w(\mathcal{Y}_2, \pi_4) &= \frac{1}{2} \\
w(\mathcal{Y}_2, \pi_5) &= \frac{1}{2} \\
w(\mathcal{Y}_2, \pi_6) &= 1 \\
w(\mathcal{Y}_2, \pi_7) &= \frac{1}{2}
\end{aligned}$$

As a result, for each piece π_i, it holds that $w(\mathcal{Y}_1, \pi_i) + w(\mathcal{Y}_2, \pi_i) \geq \frac{1}{2}$. Using the notation of Lemma 7.2, we have $k = 2$ and $r = 0.5$ therefore there is at least one YES-set that can force at least $\frac{r}{k}n = \frac{n}{4}$ queries in the second stage, given a lower bound of $4 + \frac{n}{4}$ queries for algorithms using this pattern of queries.

In this example we analyzed only one set of queries. In the next chapter we will use similar arguments applied to general query multisets.

Chapter 8

Fault-Tolerant Interval Group Testing

In Chapter 7, we presented the interval group testing problem, and some properties of each search space and queries. In this chapter we introduce a new parameter in the problem, which is the maximum number e of errors that one may find in the tests. An error is a wrong answer to a single test. In this chapter we provide bounds for the number $\mathcal{N}(n, p, s, e)$, where n is the search space size, p is the maximum number of positives, s is the maximum number of stages we have to identify them, and e is the maximum number of errors. We focus on the case when the maximum number of stages is 2, which is the most common case in practical uses of this approach. For the case when non-adaptive search is used we are able to precisely defining this number. For the 2 stage case we provide tight bounds in the case where only one error is allowed.

For the worst case analysis, we substitute the oracle by an *adversary*. The main difference between the oracle and the adversary is that, while the oracle answers the questions according to the *static* set of positives, the adversary can postpone the definition of the positive set as long as the answers in different stages are consistent[1]. For lower bounds, we look at the problem from the adversary viewpoint and analyze the maximum number of queries he can force in every case. For upper bounds we see the problem from the opposite viewpoint, the *questioner's* perspective, and design algorithms to reduce the maximum number of queries needed in the worst case. The best adversary is always able to force the questioner to the worst case, therefore he is able to force at least $\mathcal{N}(n, p, s, e)$ queries, and the best questioner is always able to

[1]With *consistent* here we mean that there is always at least one set of positives for which all the answers given so far are valid

reveal the positives using the minimum number of queries, therefore he is always able to solve the problem using at most $\mathcal{N}(n, p, s, e)$ queries. When the best questioner meets the best adversary, we observe exactly $\mathcal{N}(n, p, s, e)$ queries.

8.1 Non-Adaptive Fault-Tolerant Interval Group Testing

In adaptive group testing, there is a basic difference between the last stage and all the others stages: while the first stages may focus on reducing the search space, the last stage must point out which are the positives. In other words: the last stage is non-adaptive per se. As a result, studying non-adaptive group testing is a prerequisite to the study of any adaptive strategy. Therefore the results in this section will be the basis for the analysis of the more practical two batch case.

The following two theorems completely characterize 1-stage e-fault tolerant interval group testing.

Theorem 8.1. *For all $n \geq 1$ and $e \geq 0$, it holds that $\mathcal{N}(n, 1, 1, e) = \left\lceil \frac{(2e+1)(n+1)}{2} \right\rceil$.*

Proof. The lower bound directly follows from the following claim.

Claim. Every strategy that correctly identifies the (only) positive or reports $P = \emptyset$, uses a set of questions such that there are at least $2e+1$ questions' boundaries $(i, i+1)$ for each $i = 0, 1, \ldots, n$.

By contradiction, let us consider a strategy such that for some $i \in [n]$ there are $b \leq 2e$ questions with a boundary $(i, i+1)$. Let $\mathcal{Q}$ be the set of such questions and $\mathcal{Q}_1$ the set of all questions in $\mathcal{Q}$ which contain i. Assume, without loss of generality, that $|\mathcal{Q}_1| \geq |\mathcal{Q} \setminus \mathcal{Q}_1|$.

Let the adversary answer

- NO to all the questions having empty intersection with $\{i, i+1\}$,
- YES to all questions including both i and $i+1$,
- YES to exactly $\lceil |\frac{\mathcal{Q}}{2}| \rceil$ questions in $\mathcal{Q}_1$ and NO to the remaining ones in $\mathcal{Q}_1$,

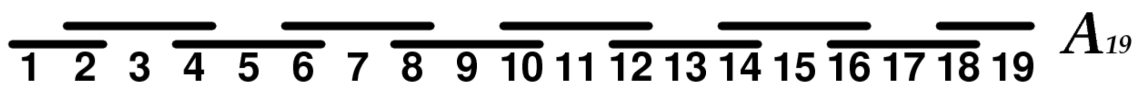

Figure 8.1: Pattern found in an optimal algorithm for a search space with an **odd** number of elements. The numbers represent the search space, whereas the horizontal bars indicate the interval queries.

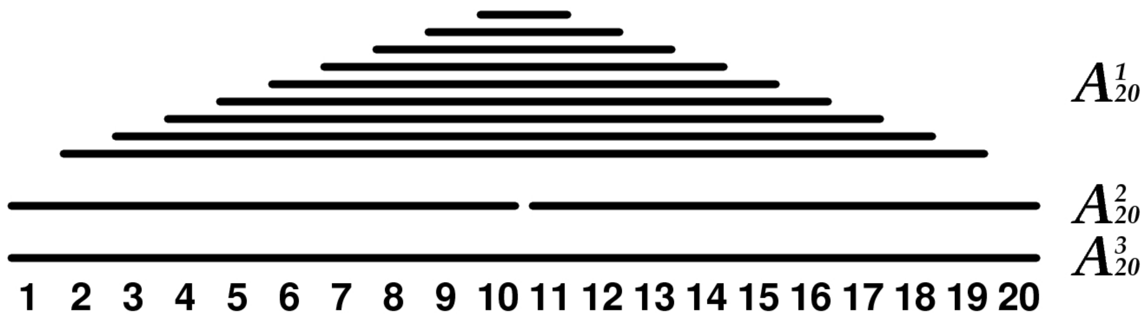

Figure 8.2: Patterns found in an optimal algorithm for a search space with an **even** number of elements. The numbers represent the search space, whereas the horizontal bars indicate the interval queries.

- YES to all the questions in $\mathcal{Q} \setminus \mathcal{Q}_1$.

A moment reflection shows that, due to the possibility of having up to e erroneous answers, the above set of answers is consistent with both cases when $P = \{i\}$ and $P = \{i+1\}$[2]. Hence, the given strategy cannot correctly discriminate among the above possibilities. The claim is proved.

Therefore, any strategy that is able to correctly identify P must use in total at least $(2e+1)(n+1)$ boundaries. Then, the desired result follows by observing that each question can cover at most 2 boundaries.

We now turn to the upper bound. Direct inspection shows that for $n \le 3$ there exists an easy strategy with the desired number of questions.

For each $k \ge 2$, let $\mathcal{A}_{2k+1} = \{[1,2],[2,4],[4,6],\ldots,[2k-2,2k],[2k,2k+1]\}$ and $\mathcal{A}^1_{2k} = \{[2,2k-1],[3,2k-2],\ldots,[k,k+1]\}$, $\mathcal{A}^2_{2k} = \{[1,k],[k+1,2k]\}$, and $\mathcal{A}^3_{2k} = \{[1,k]\}$.

Then, for $n \ge 4$, the following strategy attains the desired bound.

[2]In particular, for the cases, $i = 0$ (respectively $i = n$) the ambiguity is whether P contains no elements or the element is 1 (resp. n).

If n is odd, the strategy consists of asking $2e+1$ times the questions in $\mathcal{A}_n$. Figure 8.1 shows the groups of questions found in this algorithm. These amount to $(2e+1)\lceil(n+1)/2\rceil = \lceil(2e+1)(n+1)/2\rceil$ questions which clearly cover $2e+1$ times each boundary $(i, i+1)$, for each $i = 0, 1, \ldots, n$.

If n is even, let $k = n/2$. Now, the strategy consists of asking $2e + 1$ times the questions in $\mathcal{A}_n^1$, plus $e + 1$ times the questions in $\mathcal{A}_n^2$, plus e times the questions in $\mathcal{A}_n^3$. Figure 8.2 shows the groups of questions found in this algorithm. In total, in this case, the strategy uses $(2e + 1)(k - 1) + 2(e + 1) + e = (2e + 1)k + e + 1 = \lceil(2e+1)(2k+1)/2\rceil = \lceil(2e+1)(n+1)/2\rceil$, as desired. □

For the case of more positives we have the following generalization.

Theorem 8.2. *For all integers $n \geq 1, p \geq 2, e \geq 0$, it holds that*

$$\mathcal{N}(n, p, 1, e) = (2e+1)n.$$

Proof. The upper bound is trivially obtained by a strategy made of $(2e+1)$ copies of the singleton questions $\{1\}, \{2\}, \ldots, \{n\}$.

The lower bound is obtained proceeding in a way analogous to the argument used in the previous theorem. Here, we argue that every strategy that correctly identifies P must ask, for each $i = 1, 2, \ldots, n-1$, at least $2e + 1$ questions with boundary $(i, i+1)$ and including i, and at least $2e + 1$ questions with boundary $(i, i+1)$ and including $i+1$. Moreover, it must ask at least $2e+1$ questions with boundary $(0, 1)$ and $2e+1$ questions with boundary $(n, n+1)$. For otherwise, assume that there exists $i \in \{1, 2, \ldots, n-1\}$, such that one of the above $4e+2$ boundaries $(i, i+1)$ is missing.

Proceeding as in the proof of the previous theorem, it is possible to define an answering strategy for the adversary that balances the answers on the two sides of the boundary so that only with the information provided by the answers and given the possible number of lies, it is not possible to discriminate between the case $P = \{i\}$ and the case $P = \{i, i+1\}$, or between the case $P = \{i+1\}$ and the case $P = \{i, i+1\}$.

Alternatively, if some of the above boundaries $(0, 1)$ (resp. $(n, n+1)$) are missing, the adversary can answer in such a way that it is not possible to discriminate between the case $P = \emptyset$ and $P = \{1\}$ (resp. $P = \{n\}$). □

8.2 Two Stage Fault-Tolerant Interval Group Testing

We proved in the previous section that the number of queries necessary and sufficient to detect more than one positive in a set of n elements in the presence of at most e lies is $(2e+1)n$. The reader may be pondering at this point about the utility of group testing, since the optimal strategy implies exhaustively testing each single element in order to be sure it is a positive. In fact, this result shows that non-adaptive interval group testing does not bring any advantage. The study of non-adaptive group testing is anyway important because, as explained above, the last stage in an adaptive strategy is always a non-adaptive strategy.

8.2.1 Possible Scenarios

When errors are allowed, YES-sets may represent different scenarios. As an example, consider the YES-set in Figure 8.3. In an error-free case, the only piece containing the positive is clearly π_4. If one error is allowed, then either the answer to b or the answer to c can be wrong[3]. If the answer to b is wrong, the answer to d is for sure correct, since at most one answer can be wrong; the only positive query is c; and the positive must be in π_5. Analogously, if the answer to c is wrong, the positive is in π_3. Because the hypothesis that no error occurred cannot be discarded, the pieces π_3, π_4, and π_5 must be further investigated. Fortunately, since the positive can be found in the pieces π_3 and π_5 only in case an error occurred, the questioner may use the less expensive error-free strategies to investigate these pieces. At the end, if the YES-set in Figure 8.3 is observed in the case when at most one error is allowed, according to the bounds from the last section, the questioner will need at least

$$\frac{3}{2}\,|\pi_4| + \frac{1}{2}(|\pi_3| + |\pi_5|)$$

queries. Remembering the averaging argument presented in Section 7.1.2, in this YES-set the pieces π_3 and π_5 have weight $\frac{1}{2}$, the piece π_4 has weight $\frac{3}{2}$, and all the other pieces have weight 0.

Of course the scenario gets even more complicated and the pieces get even more weight when more errors are allowed. When two errors are allowed, some new scenarios are

[3]Notice that these are the only answers that may be wrong in case of at most 1 positive.

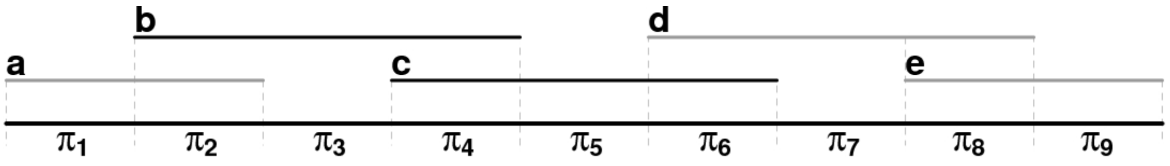

Figure 8.3: A multiset of queries with 2 positive answers (to the queries b and c) and three negative answers (to the queries a, d, and e). The stronger horizontal line represents the search space, while the thinner lines represent the interval queries. Queries in black have positive answers, whereas the ones in grey have negative answer. The projection of the borders in the search space are indicated by dashed lines and the consequent pieces are labeled $\pi_1, \ldots, \pi_9$.

possible, and other pieces enter the game. For instance, now both the answer for b and the answer for d can be wrong, which allows the positive to be in piece π_6. The same happens with queries a and c, and the piece π_2. The hypothesis that only one error occurred cannot be discarded, so that pieces π_3 and π_5 must be searched with at least a 1-error tolerant algorithm. Also the hypothesis that no error occurred cannot be ignored, and piece π_4 must be analyzed with an even heavier 2-error tolerant algorithm. Using Theorem 8.1, we get a total of

$$\frac{5}{2}\,|\pi_4| + \frac{3}{2}(|\pi_3| + |\pi_5|) + \frac{1}{2}(|\pi_2| + |\pi_6|)$$

queries in the second stage.

When errors are allowed, the number of Yes-sets increases, and the simple definition given in Definition 7.1 does not cover all valid sets of answers to a multiset of queries. As a result, the definition of Yes-sets becomes slightly more complex, as well as their analysis.

Definition 8.3. *Let $\mathcal{Q}$ be a multiset of queries, and let $\mathcal{Y} \subseteq \mathcal{Q}$. If there is a set of pieces $\mathcal{P}$ such that $|\mathcal{P}| \leq p$ and $|(\mathcal{R}(\mathcal{P}) \cup \mathcal{Y}) \setminus (\mathcal{R}(\mathcal{P}) \cap \mathcal{Y})| \leq e$, then $\mathcal{Y}$ is a* Yes-set *for $\mathcal{Q}$ in the case for p positives and e lies.*

In other words, for error-tolerant interval group testing algorithms, a Yes-set is the set of positively answered questions in a valid scenario, which may differ from the roof of a set of pieces by at most e answers.

Definition 8.3 suggests a simple way to construct Yes-sets for instances (n, p, s, e): we choose a set of pieces $\mathcal{P}$, with $|\mathcal{P}| \leq p$, and let $\mathcal{Y} = \mathcal{R}(\mathcal{P})$. Sets built in this way are called *consistent.* We may also use the allowed lies when creating YES-sets by adding (or removing) up to e elements to (from) $\mathcal{Y}$. Sets created on this way are called *inconsistent.* Notice that the consistency is not a property of the set, but of the way the set was constructed. If two consistent YES-sets $\mathcal{A}$ and $\mathcal{B}$ differ in at most e elements, then $\mathcal{B}$ can be constructed by consistently creating $\mathcal{A}$, and further transforming $\mathcal{A}$ into $\mathcal{B}$.

8.2.2 Bounds for Two-Stage Algorithms with One Positive

The aim of this section is to prove asymptotically tight upper and lower bounds on the query number of 2-stage interval group testing algorithms when up to one of the answers is a lie. We shall first analyze the case when P contains *at most one positive.*

We start with some notations and facts which will be used for the proof of the lower bound.

Let $\mathcal{Q}$ be a set of interval questions. For any piece π, cut by $\mathcal{Q}$, we denote by $\mathcal{R}(\pi)$, the *roof* of π, the set of query intervals in $\mathcal{Q}$ containing π.

Let $\pi_1, \ldots, \pi_\ell$ be the pieces determined by the intervals of $\mathcal{Q}$. Given the YES-set $\mathcal{Y}$, we define the *weight* it assigns to the piece π_i according to the following scheme:

- A piece gets weight 1/2 if it can contain a positive and there will not be a lie in the next stage.
- A piece gets weight 3/2 if it can contain a positive and there might be still a lie in the next stage.

Here, "can" means that this possibility is consistent with the YES-set.

We denote with $w_{\mathcal{Q}}(\mathcal{Y})$ the weighted sum of the lengths of the pieces cut by $\mathcal{Q}$ weighted according to the weights associated to $\mathcal{Y}$. In formulas, if $w_{\mathcal{Q}}(\mathcal{Y}, \pi_j)$ is the weight given to the piece π_j, we have

$$w_{\mathcal{Q}}(\mathcal{Y}) = \sum_{j=1}^{l} w_{\mathcal{Q}}(\mathcal{Y}, \pi_j)\, |\pi_j| \,.$$

Assume now that $\mathcal{Q}$ is the set of interval questions asked in the first stage of a two stage group testing algorithm which finds more than one positive. Using Theorems 8.1

and 8.2 it follows that if $\mathcal{Y}$ is the set of intervals in $\mathcal{Q}$ that answer YES, the number of queries to be asked in the second stage in order to find all the positives is *at least* $w_{\mathcal{Q}}(\mathcal{Y})$. Since each piece π_j that may have a positive must be solved as an independent interval group testing problem with universe of size $|\pi_j|$ at the second stage, and w_j associates the correct lower bound factor given by Theorems 8.1 and 8.2 in the case of one error.

In order to prove the promised bound we will show that for each possible set of interval questions $\mathcal{A}_1$ there exists a YES-set $\mathcal{Y}$ such that $w_{\mathcal{A}_1}(\mathcal{Y}) \geq n/|\mathcal{A}_1|$.

The following proposition allows us to limit the analysis for the lower bound to a subset of all possible first stages.

Proposition 8.4. *(12) Let $\mathcal{Q}$ be a set of interval questions producing a partition of the search space in which there are pieces a and b such that $\mathcal{R}_{\mathcal{Q}}(a) = \mathcal{R}_{\mathcal{Q}}(b)$. Then, there exists a set of interval question $\mathcal{Q}'$ of the same cardinality of $\mathcal{Q}$ such that the following two conditions hold: (i) for each two pieces a' and b' in the partition produced by $\mathcal{Q}'$ it holds $\mathcal{R}_{\mathcal{Q}'}(a') \neq \mathcal{R}_{\mathcal{Q}'}(b')$; (ii) for each* YES*-set $\mathcal{Y}'$ for $\mathcal{Q}'$ there exists a* YES*-set $\mathcal{Y}$ for $\mathcal{Q}$ such that $w_{\mathcal{Q}'}(\mathcal{Y}') = w_{\mathcal{Q}}(\mathcal{Y})$.*

After these preliminaries we can start the proof of the lower bound. Let $\mathcal{Q}$ be the set of questions asked in the first stage by a two stage interval group testing algorithm. Let $q = |\mathcal{Q}|$. By virtue of Proposition 8.4 we can assume that for each two pieces π_1 and π_2 determined by $\mathcal{Q}$ it holds that $\mathcal{R}(\pi_1) \neq \mathcal{R}(\pi_2)$. We also have that the total number ℓ of pieces is at most $2q$, since the number of pieces covered by query intervals is at most $2q-1$ (by induction) and by Proposition 8.4, at most one piece π_o is outside all query intervals ($\mathcal{R}(\pi_o) = \emptyset$).

We adapt the bounds for the 2-stage strategy for 2 positives, given by Cicalese et al. (12), to the case where a 2-stage strategy for at most one positive may contain at most one error.

Lemma 8.5.

$$\mathcal{N}(n, 1, 2, 1) \geq \sqrt{5n} - O(1).$$

Proof. We use the notation of Lemma 7.2 and show that we may achieve $r \geq 2.5$ using at most $2t_1 + 2$ YES-*sets*, where t_1 is the number of queries in the first stage. Then, by Lemma 7.2, the number of queries we need is at least $\min\left(t_1 + \frac{5n}{4t_1+4}\right) = \sqrt{5n} - O(1)$.

To achieve $r \geq 2.5$, we create a *specific* YES-set for each piece defined by the t_1 queries in the first stage. Recall that there are at most $2t_1$ distinct (according to question containment) pieces. This already guarantees $r \geq 1.5$. Moreover, each pair of adjacent pieces fall in one of the following cases, depending on how many queries separate them:

Case 1. Consider the case where two pieces are separated by the boundary of exactly one query. Let $(i, i{+}1)$ be such boundary. The YES-set created for the piece containing i assigns weight $1/2$ to the piece containing $i+1$, since there is the chance that exactly the query having the boundary $(i, i+1)$ was an error.

Therefore, by symmetry, each piece in a pair of neighbors separated by a single boundary automatically gets an extra weight $\frac{1}{2}$.

Case 2. When the pieces are separated by the boundary $(i, i+1)$ of exactly two queries, the YES-set created for one of them indicates precisely that piece as the one containing a positive. In these cases, we do not get the extra weight of $\frac{1}{2}$ for the neighbor. However, we can use the fact that, since there is no piece between these two boundaries, the number of pieces is at most $2t_1 - 1$, and so is the number of YES-sets used so far. Therefore we may create an *unspecific* YES-set involving both pieces. This is a YES-set that answers yes to all queries including both pieces and answers the two questions with boundaries $(i, i+1)$ inconsistently, i.e., one indicating the piece containing i and one indicating the piece containing $i+1$. This gives us the desired extra weight $\frac{1}{2}$ to each piece.

Case 3. Using the same argument as in the previous case, if a pair of pieces is separated by more than 2 boundaries, the number of pieces is at most $2t_1 - 2$, and we may use two new specific YES-sets, one for each piece in the pair. At the end, each of the pieces gets an extra weight of $\frac{3}{2}$ without exceeding the $2t_1 - 1$ YES-sets.

Therefore, we are able to extend the previously suggested multiset of YES-sets in such a way that each piece gets extra weight $\frac{1}{2}$ from each of its neighbors. As a result, all the pieces, but the ones on the extremities, surely have sum of weights at least 2.5. For pieces on the extremities, creating two extra consistent YES sets, one for each, gives desired total weight. At the end, we have a multiset with the desired properties. □

Lemma 8.6.

$$\mathcal{N}(n, 1, 2, 1) \leq \sqrt{5.5n}.$$

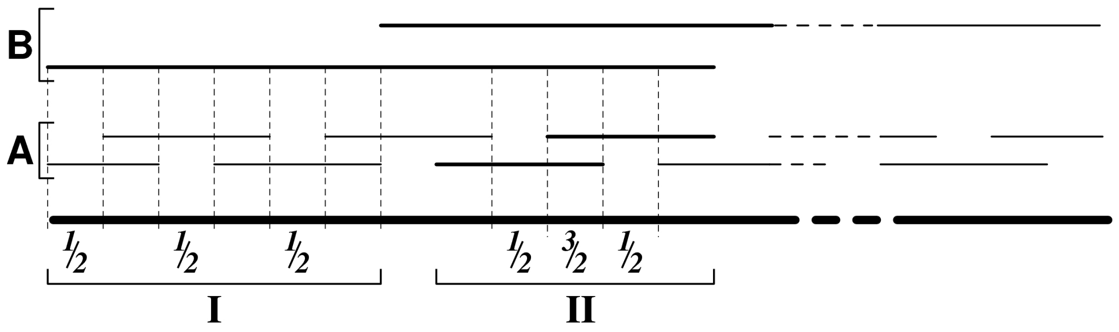

Figure 8.4: Query multiset used in the first stage of a 2-stage strategy. The thicker line represents the set of objects, whereas all the others represent the a- and b-queries. In the figure we may see the two patterns that compete for the worst case. Darker lines indicate queries that answer YES.

Proof. We show a 2-stage algorithm which is able to find a positive in a set of n elements using at most $\sqrt{5.5n}$. The first stage consist in queries divided in two groups, as shown in Fig. 8.4:

Group A: Consists of t_A overlapping queries that divide the set of objects in $2t_A$ pieces of the same size. Queries in this group are called a-queries.

Group B: Consists of rt_A overlapping queries, for $0 < r < 1$. Queries in this group are called b-queries.

An inspection of the possible YES-sets gives two situations as candidates for the worst case:

I. When a single b-query answers YES correctly, one of the a-queries surely lies. In any case, since at most one error is allowed, the positive must be in one of the single-covered pieces. As a consequence, all such pieces covered by the non overlapping part of the b-query need to be checked in the second stage. Since the non-overlapping part of the b-query has size $\frac{n}{2rt_A}$, half of this piece is covered by single a-queries, and an error-free strategy may be used in the second stage, the number of queries needed in the following stage is $\frac{n}{8rt_A}$.

II. When two overlapping a-queries answer YES, together with the corresponding b-queries, we must look for a positive in the piece corresponding to the overlapping part as if there was no error. We also need to consider the hypothesis that one of the a-queries gave the wrong answer. Therefore the two pieces corresponding

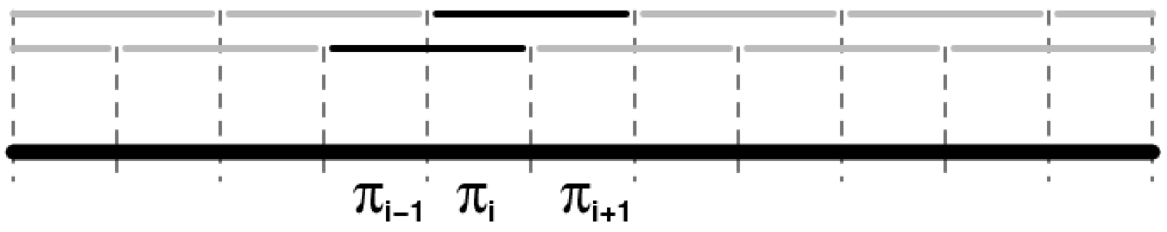

Figure 8.5: Queries in the first stage of the algorithm used in the proof of Theorem 8.7.

to the non-overlapping parts must also be investigated. In the last case, we may take advantage of the fact that they are only possible in the presence of one error, and use the error-free strategy on the two pieces of size $\frac{n}{2t_A}$. This gives us a total of $\frac{5n}{4t_A}$ questions in the second stage.

The total number of queries used by this strategy is given by

$$\min\left(t_A(1+r)+\max\left(\frac{5n}{4\,t_A},\frac{n}{8\,rt_A}\right)\right).$$

By choosing $r = 0.1$, we equalize both worst case candidates and get

$$\min_{t_A}\left(1.1\,t_A+\frac{5n}{4\,t_A}\right)=\sqrt{5.5n}.$$

□

8.2.3 Bounds for Two-Stage Algorithms and p Positives

The following lemmas summarize our finding on two stage interval group testing with at most one error in the tests.

Lemma 8.7.

$$\mathcal{N}(n,2,2,1)\leq 4\sqrt{n}+1.$$

Proof. Consider an algorithm where the first stage consists of q queries dividing the search space in $q-1$ pieces of the same size $\frac{n}{q-1}$. The multiset of queries in the first stage is:

$$\mathcal{Q}=\{\pi_1,\pi_1\cup\pi_2,\pi_2\cup\pi_3,\ldots,\pi_{q-2}\cup\pi_{q-1},\pi_{q-1}\}.$$

Notice that each piece in this stage is covered by two queries (Figure 8.5).

The worst YES-sets for this algorithm are the ones containing only two overlapping queries. Consider the YES-sets $\mathcal{Y}_i = \{\pi_{i-1} \cup \pi_i, \pi_i \cup \pi_{i+1}\}$, for $1 < i < q-1$. If no error occurred, both positives must be in π_i; by Theorem 8.2, $\frac{3n}{q-1}$ queries are needed to find the positives in the second stage. Since there is the possibility that the other query covering the piece π_{i-1} got an incorrect answer, this piece must also be investigated in the second stage. Notice that this piece can contain at most one positive, otherwise there would be two errors. By Theorem 8.1, $\frac{n}{2(q-1)}$ queries are needed to find the positive in the second stage. Because the piece π_{i+1} fall in the same case, the minimum number of queries needed by this algorithm is given by

$$\min_q \left(q + \frac{3n}{q-1} \right) = 4\sqrt{n} + 1.$$

□

Before giving a lower bound for the number of queries needed to successfully find at most 2 positives while tolerating at most one error, we define three types of YES-sets:

Type 0: Choose p pieces $\pi_1, \dots, \pi_p$ and define the YES-set as $\cup_{i=1}^{p} \mathcal{R}(\pi_i)$. In this case, because each piece contains one positive, each piece gets weight $\frac{3}{2}$.

Type 1: Choose $p-1$ pieces $\pi_1, \dots, \pi_{p-1}$ and define the YES-set as $\cup_{i=1}^{p-1} \mathcal{R}(\pi_i)$. Notice that in this case, since we have $p-1$ base pieces, each can contain two positives, and get therefore weight 3. Moreover, for each 2-piece each corresponding satellite piece gets also a weight of $\frac{3}{2}$.

Type 2: Choose $p-2$ pieces $\pi_1, \dots, \pi_{p-2}$ and define the YES-set as $\cup_{i=1}^{p-2} \mathcal{R}(\pi_i)$. Notice that in this case both the case pieces and the satellites get a weight of 3.

Lemma 8.8.

$$\mathcal{N}(n, 2, 2, 1) \geq 2\sqrt{3n}.$$

Proof. To prove the lower bound, we show that it is always possible to get weight at least 3 in each piece using at most q YES-sets. We start with a simple case where there is only one single 2-piece. In this case we build two YES-sets of type 1, both having the single 2-piece as base piece. Notice that this already gives weight 3 to this 2-piece. Since all other pieces are satellites of this single 2-piece, all of them also gets automatically a weight of 3, and we are done.

When there is more than one 2-piece, we create for each 2-piece a YES-set of type 1. As each 0-piece is a satellite of at least two pieces, 2-pieces also get a weight of at least 3. The 1-pieces get at least weight $\frac{3}{2}$, since each of them is a satellite of a 2-piece. The extra $\frac{3}{2}$ weight to each 1-piece is reached through the creation of extra type 0 YES-sets: we organize the 1-pieces in pairs, and create one YES-set of type 0 for each pair, using the 1-pieces as base pieces.

Remembering that $c_1 + 2c_2 = 2q$, where c_i is the number of i pieces, we have that $\frac{c_1}{2} + c_2 \leq q$ if c_1 is even, and $\frac{(c_1-1)}{2} + c_2 \leq q - 1$ if c_1 is odd. In any case, the number of YES-sets is not greater than q. After Theorem 7.2, we need at least $\frac{3n}{q}$ queries in the second stage, which gives a lower bound of

$$\min_q \left(q + \frac{3n}{q} \right) = 2\sqrt{3n}.$$

□

For cases when $p \geq 3$, the best bounds found so far, to the best of our knowledge, are given by the following theorem.

Theorem 8.9. *For $p \geq 3$,*

$$\mathcal{N}(n, p, 2, 1) \leq 2\sqrt{6(p-2)n} + 1.$$

Proof. For proving this upper bound we apply the same algorithm used in the proof of Theorem 8.7. Let q be the number of queries used in the first stage and consider the same worst case YES-sets analyzed in the previous theorem: let $\mathcal{Y}_i = \{\pi_{i-1} \cup \pi_i, \pi_i \cup \pi_{i+1}\}$, for $1 < i < q - 1$. Since we may have more than 2 positives, the patterns observed in the proof of the previous theorem become heavier: now the two not entirely covered pieces π_{i-1} and π_{i+1} may contain 2 positives as well, and require, by Theorem 8.2, at least $\frac{3n}{2(q-1)}$ queries in the second stage. Each pair of positive answers $\mathcal{Y}_i$ requires a total of $\frac{6n}{q-1}$ queries in the second stage.

A smart adversary notices that such a YES-set only needs 3 positives to be built, leaving $p-3$ positives to be used to create an even worse situation for the questioner. With each extra positive it is possible to create an extra pair of positive answers, which looks exactly like $\mathcal{Y}_i$. The adversary can create the YES-set $\mathcal{Y}$ by combining $p-2$ of such YES-sets that do not overlap: for each two YES-set $\mathcal{Y}_j$ and $\mathcal{Y}_k$, $j<k$, it holds that $j<k-1$.

The problem for the questioner when facing a YES-set like $\mathcal{Y}$ is that, although it is clear that not all pairs of positive answers can contain 3 positives, and that most of them must have exactly one, he cannot differentiate them, and need to use the more powerful, and expensive, algorithms in each of them. This forces the use of $\frac{6n(p-2)}{q-1}$ queries in the second stage. Optimizing with respect to q, the total number of queries in the two stages is

$$\min_q \left(q+\frac{6n(p-2)}{q-1}\right)=2\sqrt{6n(p-2)}+1.$$

□

For the lower bound we need the following lemma proved by Cicalese *et. al* (12).

Lemma 8.10 (Lemma 4 in (12)). *Let x,y be positive integers with $x\geq 2y$ and x even. In x cells arranged in a cycle, we can place pebbles from x sets, each with y pebbles, so that every cell gets y pebbles, and every pair of neighbored cells get pebbles from $2y$ distinct sets. If $x>2y$ is odd, at most $x+1$ sets are needed to achieve the same.*

Theorem 8.11. *For $p\geq 3$,*

$$\mathcal{N}(n,p,2,1)\geq 2\sqrt{3n(p-1)}+O(p),$$

provided that the number of queries in the first stage is at least $2\frac{(p-1)^2}{p-2}$.

Proof. Let q be the number of queries asked in the first stage. We show that it is possible to achieve weight at least $3(p-1)$ with no more than $q+\frac{p}{2}+1$ YES-sets. For doing that, we analyze three different cases:

Case 1. $c_2 \leq p-2$**.** We create q YES-sets of type 2, with all the c_2 2-pieces as base pieces. Since all 1-pieces are satellites of some 2-piece, the YES-sets give a weight at least $3q$ to all the pieces. Using $q = \omega(p)$, we have reached the desired weight.

Case 2. $c_2 \geq p-1$ **and** $q(p-2) \geq c_2(p-1)$**.** We create q YES-sets of type 2, using $(p-2)$ 2-pieces as base pieces for them. This is done in such a way that every 2-piece appears at least $p-1$ times. Due to the assumptions for this case, the previous condition is always fulfilled. As a result, the desired weight is reached.

Case 3. $c_2 \geq p-1$ **and** $q(p-2) < c_2(p-1)$**.** We create c_2 type 1 YES-sets with $p-1$ 2-pieces as base pieces in such a way that each 2-piece appears exactly $p-1$ times. This guarantees a weight $p-1$ to every 2-piece.

Consider now the 0-pieces in such an algorithm. Because $c_2 \geq p-1$ and $q(p-2) < c_2(p-1)$, we have $c_2 > 2(p-1)$. Consider now a cyclic ordering of the 2-pieces, which consists of their natural ordering closed to a cycle. Remember also that each 0-piece inside an interval is between two 2-pieces and is a satellite of them. We assume that the only 0-piece outside all query intervals, if existing, is between the first and the last 2-piece in the cyclic order, and is a satellite of them. By Lemma 8.10, if c_2 is even, there is a way to create the YES-sets so that each of the neighboring 2-pieces of each of the 0-pieces appear in $2(p-1)$ *distinct* YES-sets. As a result, they get a weight of $\frac{3}{2}$ from each of them, assuring the weight of at least $3(p-1)$. If c_2 is odd, c_2+1 YES-sets of type 1 are needed.

With the at most c_2+1 YES-sets used so far, we achieve a weight of at least $3(p-1)$ in every 0-piece and 2-piece. Moreover, the 1-pieces already have a weight at least $\frac{3}{2}(p-1)$, since they are all satellite of 2-pieces. The question now is how to guarantee the weight at least $3(p-1)$ in the 1-pieces too.

We consider now the case when $c_1 \geq p-1$. In this case we create $\lceil \frac{c_1}{2} \rceil$ YES-sets of type 1, each of them with $p-1$ 1-pieces as base pieces. Notice that this can be done in such a way that every piece appears at least $\frac{p-1}{2}$ times, giving to each 1-piece the desired lower bounded weight with $\frac{c_1}{2} + c_2 \leq q$ YES-sets.

We are left with the case when $c_1 < p-1$. In this case we build $\lceil \frac{p-1}{2} \rceil$ YES-sets of type 2, each with all 1-pieces as base pieces. Because $\frac{c_1}{2} + c_2 \leq q$, and in special that $c_2 \leq q$, we can say that the number of YES-sets is upper bounded by $q + \frac{p}{2}$.

Since in each case we reached the lower bound of $3(p-1)$ for the weight of every piece, and in the worst case we needed no more than $q+\frac{p}{2}+1$ Yes-sets, the minimum number of queries needed in the worst case is

$$\min_q \left(q + \frac{3(p-1)}{q+\frac{p}{2}+1} n \right) = 2\sqrt{3n(p-1)} + O(p).$$

□

8.2.4 More Errors

The bounds presented so far only deal with the case where at most one error may be expected. This may be useful for experiments where the probability of an error occurrence is very small. But in such experiments it may also be the case that the probability of error is so small that fault-tolerance is not really necessary. Here we generalize some previously given bounds to the case where the number of errors is bounded by a parameter e.

Theorem 8.12.

$$\mathcal{N}(n,2,2,e) \geq 2\sqrt{(2e+1)n}.$$

Proof. Observe that the analysis of Yes-sets made in Theorem 8.8 can be repeated here, only taking care of using the right weight for each piece: here pieces containing a single positive get weight at least $\frac{2e+1}{2}$ in the corresponding Yes-set, while pieces containing more than one positive get weight at least $2e+1$. As a result, we get weight at least $2e+1$ in each piece with at most q Yes-sets, giving a lower bound of

$$\min_q \left(q + \frac{(2e+1)}{q} n \right) = 2\sqrt{(2e+1)n}.$$

□

Theorem 8.13. *For $p \geq 3$,*

$$\mathcal{N}(n,p,2,e) \geq 2\sqrt{(2e+1)(p-1)n} + O(p),$$

provided that the number of queries in the first stage is at least $2\frac{(p-1)^2}{p-2}$.

Proof. Also here we use the same reasoning as in the corresponding 1-error version (Theorem 8.9). Let q be the number of queries in the first stage of the algorithm. The new piece weights allow us to get at least weight $(2e+1)(p-1)$ for the weight of each piece using no more than $q + \frac{p}{2} + 1$ YES-sets. Therefore the minimum number of queries needed in the worst case is

$$\min_q \left(q + \frac{(2e+1)(p-1)}{q + \frac{p}{2} + 1} n \right) = 2\sqrt{(2e+1)(p-1)n} + O(p).$$

□

Theorem 8.14. *For $p \geq 2$,*

$$\mathcal{N}(n, p, 2, e) \leq 2\sqrt{(2e+1)(e+1)(p-1)n}.$$

Proof. We divide the search space in q pieces and create for each of them $(e+1)$ queries covering only the corresponding piece. This gives $q(e+1)$ queries in the first stage. Notice that in this case every error can be easily identified in the first stage and would allow to the use of less queries in the second stage. Therefore the worst case occurs when all queries covering $(p-1)$ pieces answer Yes. In this case, all the positive pieces have to be analyzed in the second stage with an e-error tolerant strategy. Since each piece has size $\frac{n}{q}$, the total of queries is given by:

$$\min_q \left(q(e+1) + (p-1)(2e+1)\frac{n}{q} \right) = 2\sqrt{(e+1)(2e+1)(p-1)n}.$$

□

8.3 Conclusion

In this chapter we presented bounds for the number of queries needed for solving different instances of the Interval Group Testing problem. The non-adaptive case is fully characterized and optimal algorithms for all studied instances are given in Section 8.1. For 2-stage algorithms tolerating one error we give good approximations to the number of queries needed. The ratio between the upper and the lower bounds

Table 8.1: Upper and lower bounds for instances of the Interval Group Testing problem. The instances are presented on the leftmost column, while the ratio upper/lower bound for each instance is presented in the rightmost columns. On the leftmost column, parameters in bold highlight the differences between the instance shown in the corresponding row and the instance in the instance directly above.

Parameters	Lower Bound	Upper Bound	$\frac{\text{Upper Bound}}{\text{Lower Bound}}$
$\boldsymbol{p=1}$, $\boldsymbol{s=1}$, $\boldsymbol{e\geq 1}$	$\left\lceil\frac{(2e+1)(n+1)}{2}\right\rceil$	$\left\lceil\frac{(2e+1)(n+1)}{2}\right\rceil$	1
$\boldsymbol{p\geq 2}$, $s=1$, $e\geq 1$	$(2e+1)n$	$(2e+1)n$	1
$\boldsymbol{p=1}$, $\boldsymbol{s=2}$, $\boldsymbol{e=1}$	$\sqrt{5n}$	$\sqrt{5.5n}$	$\sqrt{1.1}$
$\boldsymbol{p=2}$, $s=2$, $e=1$	$2\sqrt{3n}$	$4\sqrt{n}$	$\sqrt{\frac{4}{3}}$
$\boldsymbol{p\geq 3}$, $s=2$, $e=1$	$2\sqrt{3(p-1)n}$	$2\sqrt{6(p-2)n}+1$	$\sqrt{2\frac{p-2}{p-1}}$
$\boldsymbol{p=2}$, $s=2$, $\boldsymbol{e\geq 1}$	$2\sqrt{(2e+1)n}$	$2\sqrt{(e+1)(2e+1)n}$	$\sqrt{e+1}$
$\boldsymbol{p\geq 3}$, $s=2$, $e\geq 1$	$2\sqrt{(2e+1)(p-1)n}$	$2\sqrt{(e+1)(2e+1)(p-1)n}$	$\sqrt{e+1}$

in these cases are $\sqrt{1.1}\approx 1.005$ for up to one positive, $\sqrt{\frac{4}{3}}\approx 1.155$ for up to two positives, and $\sqrt{2\frac{p-2}{p-1}}<\sqrt{2}\approx 1.414$ for up to $p\geq 3$ positives.

The bounds for one error could be generalized for any upper bound on the number of errors. However, the ratio between upper and lower bound in these cases is $\sqrt{e+1}$. The lower and upper bounds are summarized in the Table 8.1 and the ratio between them is shown in the rightmost column of the same table.

Chapter 9

Conclusions and Outlook

In the first part of this thesis, we presented and analyzed a method for constructing a compacted form of de Bruijn subgraphs that we called *sequence graph*. Although the contraction of induced paths into single nodes is a common practice in de Bruijn graph based applications, the direct construction of de Bruijn subgraphs in their contracted form does not seem to be explored or at least documented. In the three last chapters of the first part we discussed two applications for sequence graphs: repeat family identification in partially sequenced genomes and splicing graph construction.

Our first experiments with the identification of repeat families, presented in Chapter 3, showed that the repeat family identification is possible, but a simple approach based on elimination of long nodes may split parts of the repeat sequences into separated groups. Therefore in Chapter 4 we proposed a method for combining similar vertices inside repetitive regions and applied this variant of the repeat family identification method in artificially created reads from several bacterial genomes, as well as to short reads of a plant genome. Despite some problems with low genome coverage and families with few members, the method was able to identify repeat families in bacterial genomes.

Like many other de Bruijn subgraph based applications, the method for identification of repeat families does not scale well from prokaryotic to eukaryotic genomes. The higher complexity, variation inside repeat families, and repeat content in eukaryotic genomes complicate the isolation of repeat families by simply deleting vertices without accidentally spliting families in different groups. The precise separation of repeat families in eukaryotic genomes relies on a better understanding of the distinct repetitive elements and their patterns in de Bruijn subgraphs. On the other hand, the identification of repeat families by visual inspection is often a simple task. While it is not clear whether a de Bruijn subgraph based approach will ever be able to identify the majority of the repeat families in a genome or not, we believe that these subgraphs could be used as base for data visualizations in softwares for the support of manual repeat analysis.

A good surprise was the results of the use of de Bruijn subgraphs in the construction of splicing graphs. Although splicing almost only occurs in the complicated eukaryotic genomes, the fact that mature RNA molecules are not embedded in large chromosomes and that common subsequences of splicing variants are copies of the same locus in the genome allow the use of graphs of higher dimension and a better separation of mRNAs in components corresponding to sets of splicing variants of the same gene. Of course this approach only deal with the construction of splicing graphs. Problems related to the identification (and quantification) of the variants were not studied here and are a natural and important continuation to this work.

The second part of this thesis was dedicated to the study of interval group testing strategies in the presence of errors. We were able to fully characterize the non-adaptive case, where all questions are asked at once. For 2-stage strategies we gave good approximation algorithms for the case where at most one error is allowed and the number of positives is at most 2. For other upper bounds on the number of positives we provided a 1.414 approximation algorithm. Apart from the case of one positive, the 2-stage algorithms can be adapted to the case where the number of errors is upper bounded by a parameter e. Unfortunately the corresponding approximation rate is $\sqrt{e+1}$. These results leave place for improvements both in cases where more errors are allowed and where the number of stages is bigger than 2. Since strategies with few stages are usually preferred, improvements in the direction of allowing more errors should have a higher priority.

Appendix A

Complete Set of Tables for Section 4.1

The following tables present the complete data used in the sequence graph analysis in Section 4.1.

The Tables A.1, A.2, A.3, and A.4 present the percent of known families that could be identified in the final graph for genome coverage values 25%, 50%, 75%, and 100% respectively. The Tables A.5, A.6, A.7, and A.8 present the average number of distinct families in the same component for for genome coverage values 25%, 50%, 75%, and 100% respectively. Finally, the Tables A.9, A.10, A.11, and A.12 present the average number of components in which the same repeat family is found for genome coverage values 25%, 50%, 75%, and 100% respectively.

Table A.1: Average percentage of known families which were discovered at 25% coverage.

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	51	49	72	71	81	81	82
Burkholderia xenovorans	50	50	84	84	88	88	90
Colwellia psychrerythraea	20	20	33	33	33	33	33
Desulfitobacterium hafniense	87	80	95	95	100	100	100
Desulfovibrio desulfuricans	50	43	93	93	97	97	100
Escherichia coli	45	42	75	73	85	85	92
Geobacter uraniumreducens	46	46	72	72	72	72	80
Gloeobacter violaceus	63	63	85	85	88	88	98
Granulibacter bethesdensis	10	7	30	27	53	50	70
Haloarcula marismortui	14	13	47	44	76	70	90
Halobacterium sp-plasmid pNRC100	45	37	54	47	51	51	59
Legionella pneumophila	35	32	30	33	40	40	30
Legionella pneumophila-Philadelphia 1	43	38	70	63	85	80	87
Methanosarcina acetivorans	96	95	99	99	100	100	100
Methylococcus capsulatus	44	44	82	83	96	96	98
Nitrosospira multiformis	87	70	97	93	100	100	100
Photobacterium profundum	100	100	100	100	100	100	100
Pseudomonas syringae	97	97	100	100	100	100	100
Pyrococcus furiosus	57	51	72	69	80	80	94
Ralstonia solanacearum	63	63	88	88	95	95	98
Rhodopirellula baltica	89	89	100	100	100	100	100
Roseobacter denitrificans	70	50	90	87	100	100	100
Salinibacter ruber	97	97	100	100	100	100	100
Shewanella oneidensis	27	27	29	29	41	34	33
Sulfolobus solfataricus	99	99	99	99	100	100	100

Table A.2: Average percentage of known families which were discovered at 50% coverage.

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	51	48	72	71	81	83	82
Burkholderia xenovorans	50	50	84	84	88	90	90
Colwellia psychrerythraea	20	20	33	33	33	33	33
Desulfitobacterium hafniense	87	80	95	97	100	100	100
Desulfovibrio desulfuricans	50	43	93	93	97	100	100
Escherichia coli	44	42	74	73	85	92	92
Geobacter uraniumreducens	46	45	72	72	72	80	80
Gloeobacter violaceus	63	63	85	85	88	98	98
Granulibacter bethesdensis	10	7	30	27	50	70	73
Haloarcula marismortui	14	14	47	44	72	90	91
Halobacterium sp-plasmid pNRC100	41	37	48	47	51	65	59
Legionella pneumophila	37	32	30	33	40	27	30
Legionella pneumophila-Philadelphia 1	43	38	67	62	83	95	87
Methanosarcina acetivorans	95	95	99	99	100	100	100
Methylococcus capsulatus	44	44	83	83	96	98	98
Nitrosospira multiformis	87	70	97	93	100	100	100
Photobacterium profundum	100	100	100	100	100	100	100
Pseudomonas syringae	97	97	100	100	100	100	100
Pyrococcus furiosus	57	51	71	69	80	96	94
Ralstonia solanacearum	63	63	88	88	95	98	98
Rhodopirellula baltica	89	89	100	100	100	100	100
Roseobacter denitrificans	67	50	90	87	100	100	100
Salinibacter ruber	97	97	100	100	100	100	100
Shewanella oneidensis	27	27	29	31	41	33	33
Sulfolobus solfataricus	99	99	99	99	100	100	100

Table A.3: Average percentage of known families which were discovered at 75% coverage.

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	51	48	71	81	81	83	82
Burkholderia xenovorans	50	50	84	88	88	90	90
Colwellia psychrerythraea	20	20	33	33	33	33	33
Desulfitobacterium hafniense	80	80	95	100	100	100	100
Desulfovibrio desulfuricans	50	43	93	97	97	100	100
Escherichia coli	44	42	74	85	85	92	92
Geobacter uraniumreducens	46	45	72	72	72	80	80
Gloeobacter violaceus	63	63	85	88	88	98	98
Granulibacter bethesdensis	10	7	23	53	50	70	73
Haloarcula marismortui	14	14	44	75	69	90	91
Halobacterium sp-plasmid pNRC100	41	37	47	59	51	65	59
Legionella pneumophila	33	32	30	33	40	27	30
Legionella pneumophila-Philadelphia 1	43	37	62	87	83	95	87
Methanosarcina acetivorans	95	95	99	100	100	100	100
Methylococcus capsulatus	44	44	83	96	96	98	98
Nitrosospira multiformis	87	70	97	100	100	100	100
Photobacterium profundum	100	100	100	100	100	100	100
Pseudomonas syringae	97	97	100	100	100	100	100
Pyrococcus furiosus	57	51	69	84	80	96	94
Ralstonia solanacearum	63	63	88	95	95	98	98
Rhodopirellula baltica	89	89	100	100	100	100	100
Roseobacter denitrificans	60	50	87	100	100	100	100
Salinibacter ruber	100	97	100	100	100	100	100
Shewanella oneidensis	27	26	29	38	41	33	33
Sulfolobus solfataricus	99	99	99	100	100	100	100

Table A.4: Average percentage of known families which were discovered at 100% coverage.

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	49	72	71	81	81	83	82
Burkholderia xenovorans	50	84	84	88	88	90	90
Colwellia psychrerythraea	20	33	33	33	33	33	33
Desulfitobacterium hafniense	80	95	95	100	100	100	100
Desulfovibrio desulfuricans	50	93	93	97	97	100	100
Escherichia coli	43	75	73	85	85	92	92
Geobacter uraniumreducens	46	72	72	72	72	80	80
Gloeobacter violaceus	63	85	85	88	88	98	98
Granulibacter bethesdensis	7	30	27	53	50	70	73
Haloarcula marismortui	13	47	44	75	69	90	91
Halobacterium sp-plasmid pNRC100	39	57	47	56	51	61	59
Legionella pneumophila	32	30	33	33	40	30	30
Legionella pneumophila-Philadelphia 1	38	70	63	87	83	93	83
Methanosarcina acetivorans	95	99	99	100	100	100	100
Methylococcus capsulatus	44	82	83	96	96	98	98
Nitrosospira multiformis	70	97	93	100	100	100	100
Photobacterium profundum	100	100	100	100	100	100	100
Pseudomonas syringae	97	100	100	100	100	100	100
Pyrococcus furiosus	53	72	69	84	80	94	94
Ralstonia solanacearum	63	88	88	95	95	98	98
Rhodopirellula baltica	89	100	100	100	100	100	100
Roseobacter denitrificans	50	90	87	100	100	100	100
Salinibacter ruber	97	100	100	100	100	100	100
Shewanella oneidensis	27	29	29	38	41	33	35
Sulfolobus solfataricus	99	99	99	100	100	100	100

Table A.5: Average number of families in each component in a sequence graph (25% coverage).

Combine Factor (τ)	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**
Bifidobacterium longum	1.2	1.2	1.2	1.2	1.2	1.2	1.3
Burkholderia xenovorans	1.0	1.0	1.5	1.5	2.1	2.1	2.4
Colwellia psychrerythraea	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfitobacterium hafniense	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfovibrio desulfuricans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Escherichia coli	1.0	1.0	1.1	1.0	1.1	1.1	1.1
Geobacter uraniumreducens	1.4	1.4	1.4	1.4	1.4	1.4	1.5
Gloeobacter violaceus	1.0	1.0	1.1	1.1	1.4	1.4	1.7
Granulibacter bethesdensis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Haloarcula marismortui	1.0	1.0	1.0	1.1	1.5	1.6	2.0
Halobacterium sp-plasmid pNRC100	1.1	1.1	1.2	1.2	1.4	1.5	1.5
Legionella pneumophila	1.0	1.0	1.2	1.2	1.3	1.3	1.2
Legionella pneumophila-Philadelphia 1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Methanosarcina acetivorans	1.6	1.6	1.5	1.5	2.2	2.2	3.6
Methylococcus capsulatus	1.0	1.0	1.0	1.0	1.1	1.1	1.5
Nitrosospira multiformis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Photobacterium profundum	1.4	1.5	1.5	1.5	1.5	1.6	1.5
Pseudomonas syringae	1.5	1.5	2.0	2.0	2.2	2.2	3.6
Pyrococcus furiosus	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Ralstonia solanacearum	1.4	1.4	3.4	3.5	4.4	4.4	6.9
Rhodopirellula baltica	1.3	1.3	1.3	1.4	1.4	1.4	1.7
Roseobacter denitrificans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Salinibacter ruber	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Shewanella oneidensis	1.5	1.5	2.2	2.2	2.3	2.5	2.7
Sulfolobus solfataricus	2.0	2.1	3.1	3.6	4.6	4.7	5.4

Table A.6: Average number of families in each component in a sequence graph (50% coverage).

Combine Factor (τ)	**1.0**	**1.5**	**2.0**	**2.5**	**3.0**	**3.5**	**4.0**
Bifidobacterium longum	1.2	1.2	1.2	1.2	1.2	1.3	1.3
Burkholderia xenovorans	1.0	1.0	1.5	1.5	2.1	2.4	2.3
Colwellia psychrerythraea	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfitobacterium hafniense	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfovibrio desulfuricans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Escherichia coli	1.0	1.0	1.1	1.0	1.1	1.1	1.1
Geobacter uraniumreducens	1.4	1.4	1.4	1.4	1.4	1.5	1.5
Gloeobacter violaceus	1.0	1.0	1.1	1.1	1.4	1.7	1.7
Granulibacter bethesdensis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Haloarcula marismortui	1.0	1.0	1.0	1.1	1.6	2.0	2.0
Halobacterium sp-plasmid pNRC100	1.1	1.1	1.2	1.2	1.5	1.3	1.5
Legionella pneumophila	1.0	1.0	1.2	1.2	1.3	1.2	1.2
Legionella pneumophila-Philadelphia 1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Methanosarcina acetivorans	1.6	1.6	1.5	1.6	2.2	3.3	3.7
Methylococcus capsulatus	1.0	1.0	1.0	1.0	1.1	1.4	1.5
Nitrosospira multiformis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Photobacterium profundum	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Pseudomonas syringae	1.5	1.5	2.0	2.0	2.2	3.6	3.6
Pyrococcus furiosus	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Ralstonia solanacearum	1.4	1.4	3.5	3.5	4.4	6.9	6.9
Rhodopirellula baltica	1.3	1.3	1.4	1.4	1.4	1.7	1.7
Roseobacter denitrificans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Salinibacter ruber	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Shewanella oneidensis	1.5	1.5	2.2	2.2	2.3	2.7	2.7
Sulfolobus solfataricus	2.0	2.2	3.2	3.6	4.6	5.2	5.4

Table A.7: Average number of families in each component in a sequence graph (75% coverage).

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	1.2	1.2	1.2	1.2	1.2	1.3	1.3
Burkholderia xenovorans	1.0	1.0	1.5	2.1	2.1	2.4	2.3
Colwellia psychrerythraea	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfitobacterium hafniense	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfovibrio desulfuricans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Escherichia coli	1.0	1.0	1.1	1.1	1.1	1.1	1.1
Geobacter uraniumreducens	1.4	1.4	1.4	1.4	1.4	1.5	1.5
Gloeobacter violaceus	1.0	1.0	1.1	1.4	1.4	1.7	1.7
Granulibacter bethesdensis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Haloarcula marismortui	1.0	1.0	1.0	1.5	1.6	2.0	2.0
Halobacterium sp-plasmid pNRC100	1.1	1.1	1.2	1.3	1.5	1.3	1.5
Legionella pneumophila	1.0	1.0	1.2	1.4	1.3	1.2	1.2
Legionella pneumophila-Philadelphia 1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Methanosarcina acetivorans	1.6	1.6	1.5	2.1	2.2	3.5	3.7
Methylococcus capsulatus	1.0	1.0	1.0	1.1	1.1	1.4	1.5
Nitrosospira multiformis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Photobacterium profundum	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Pseudomonas syringae	1.4	1.5	2.0	2.2	2.2	3.6	3.6
Pyrococcus furiosus	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Ralstonia solanacearum	1.4	1.4	3.5	4.4	4.5	6.9	6.9
Rhodopirellula baltica	1.3	1.3	1.4	1.4	1.4	1.7	1.7
Roseobacter denitrificans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Salinibacter ruber	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Shewanella oneidensis	1.5	1.4	2.2	2.3	2.3	2.7	2.7
Sulfolobus solfataricus	2.0	2.3	3.2	4.5	4.7	5.2	5.4

Table A.8: Average number of families in each component in a sequence graph (100% coverage).

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	1.2	1.2	1.2	1.2	1.2	1.3	1.3
Burkholderia xenovorans	1.0	1.5	1.5	2.1	2.1	2.4	2.4
Colwellia psychrerythraea	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfitobacterium hafniense	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Desulfovibrio desulfuricans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Escherichia coli	1.0	1.1	1.0	1.1	1.1	1.1	1.1
Geobacter uraniumreducens	1.4	1.4	1.4	1.4	1.4	1.5	1.6
Gloeobacter violaceus	1.0	1.1	1.1	1.4	1.4	1.7	1.7
Granulibacter bethesdensis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Haloarcula marismortui	1.0	1.0	1.1	1.5	1.6	2.0	1.9
Halobacterium sp-plasmid pNRC100	1.1	1.2	1.2	1.3	1.5	1.4	1.5
Legionella pneumophila	1.0	1.2	1.2	1.4	1.3	1.2	1.2
Legionella pneumophila-Philadelphia 1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Methanosarcina acetivorans	1.6	1.5	1.5	2.1	2.2	3.6	3.6
Methylococcus capsulatus	1.0	1.0	1.0	1.1	1.1	1.5	1.5
Nitrosospira multiformis	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Photobacterium profundum	1.5	1.5	1.5	1.5	1.5	1.5	1.5
Pseudomonas syringae	1.4	2.0	2.0	2.2	2.2	3.6	3.6
Pyrococcus furiosus	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Ralstonia solanacearum	1.4	3.2	3.5	4.4	4.5	6.9	6.9
Rhodopirellula baltica	1.3	1.3	1.4	1.4	1.4	1.7	1.7
Roseobacter denitrificans	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Salinibacter ruber	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Shewanella oneidensis	1.5	2.2	2.2	2.3	2.3	2.7	3.0
Sulfolobus solfataricus	2.1	2.9	3.2	4.5	4.7	5.4	5.4

Table A.9: Average number of components per repeat family in a sequence graph (25% coverage).

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	2.7	2.5	3.3	3.1	3.7	3.5	3.8
Burkholderia xenovorans	2.6	2.6	2.0	2.0	1.5	1.5	1.3
Colwellia psychrerythraea	3.6	2.9	3.1	3.1	3.1	2.9	3.1
Desulfitobacterium hafniense	3.7	3.3	3.8	3.6	3.7	3.7	3.3
Desulfovibrio desulfuricans	2.3	2.2	2.8	2.5	3.6	3.3	3.3
Escherichia coli	2.3	2.2	3.0	3.0	3.0	3.1	3.1
Geobacter uraniumreducens	1.8	1.8	2.2	2.2	2.5	2.5	2.2
Gloeobacter violaceus	3.5	3.4	3.6	3.4	2.0	2.0	1.3
Granulibacter bethesdensis	2.0	2.0	2.0	2.0	2.1	2.1	2.1
Haloarcula marismortui	2.2	2.1	2.0	2.0	1.7	1.6	1.6
Halobacterium sp-plasmid pNRC100	2.6	2.4	2.6	2.5	1.6	1.5	1.5
Legionella pneumophila	2.6	2.5	2.6	2.9	2.0	2.2	1.6
Legionella pneumophila-Philadelphia 1	2.7	2.1	3.7	3.2	4.1	3.9	4.2
Methanosarcina acetivorans	3.1	2.9	2.9	2.8	1.9	1.9	1.5
Methylococcus capsulatus	2.2	2.2	3.1	2.9	3.5	3.5	2.7
Nitrosospira multiformis	3.4	3.1	4.5	4.2	5.4	5.1	3.4
Photobacterium profundum	1.9	1.7	1.8	1.8	1.8	1.7	1.9
Pseudomonas syringae	2.2	2.1	1.9	1.9	1.6	1.6	1.2
Pyrococcus furiosus	1.8	1.7	1.7	1.6	1.6	1.7	1.8
Ralstonia solanacearum	2.4	2.4	1.4	1.4	1.2	1.2	1.1
Rhodopirellula baltica	2.3	2.2	2.6	2.5	2.4	2.4	2.0
Roseobacter denitrificans	2.5	2.3	3.3	3.0	3.8	3.8	3.5
Salinibacter ruber	3.1	2.8	2.1	1.9	1.0	1.0	1.0
Shewanella oneidensis	2.0	2.0	1.3	1.3	1.4	1.3	1.3
Sulfolobus solfataricus	2.2	2.1	1.5	1.5	1.3	1.3	1.2

Table A.10: Average number of components per repeat family in a sequence graph (50% coverage).

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	2.7	2.4	3.3	3.1	3.6	3.8	3.8
Burkholderia xenovorans	2.6	2.6	2.0	2.0	1.5	1.3	1.3
Colwellia psychrerythraea	3.6	2.9	3.1	3.1	3.0	3.1	3.1
Desulfitobacterium hafniense	3.6	3.3	3.7	3.5	3.7	3.5	3.3
Desulfovibrio desulfuricans	2.3	2.2	2.8	2.5	3.4	3.3	3.1
Escherichia coli	2.3	2.2	3.0	3.0	3.0	3.1	3.1
Geobacter uraniumreducens	1.8	1.8	2.2	2.2	2.5	2.2	2.1
Gloeobacter violaceus	3.5	3.4	3.6	3.4	2.0	1.4	1.3
Granulibacter bethesdensis	2.0	2.0	2.0	2.0	2.1	2.1	2.1
Haloarcula marismortui	2.2	2.0	2.0	2.1	1.6	1.6	1.6
Halobacterium sp-plasmid pNRC100	2.6	2.3	2.6	2.4	1.5	2.0	1.5
Legionella pneumophila	2.6	2.5	2.8	2.9	2.1	1.6	1.6
Legionella pneumophila-Philadelphia 1	2.6	2.1	3.5	3.2	4.0	4.1	4.1
Methanosarcina acetivorans	3.0	2.9	2.8	2.8	1.9	1.5	1.4
Methylococcus capsulatus	2.2	2.2	3.1	3.0	3.5	2.9	2.7
Nitrosospira multiformis	3.2	3.1	4.4	4.1	5.3	3.5	3.4
Photobacterium profundum	1.7	1.7	1.8	1.8	1.8	1.8	1.9
Pseudomonas syringae	2.1	2.1	1.9	1.9	1.6	1.2	1.2
Pyrococcus furiosus	1.8	1.7	1.7	1.6	1.6	1.8	1.8
Ralstonia solanacearum	2.4	2.4	1.4	1.4	1.2	1.1	1.1
Rhodopirellula baltica	2.3	2.2	2.5	2.5	2.4	2.0	2.0
Roseobacter denitrificans	2.5	2.3	3.3	3.0	3.8	3.5	3.5
Salinibacter ruber	3.1	2.7	2.1	1.9	1.0	1.0	1.0
Shewanella oneidensis	2.0	2.0	1.3	1.3	1.4	1.3	1.3
Sulfolobus solfataricus	2.2	2.0	1.5	1.4	1.3	1.2	1.2

Table A.11: Average number of components per repeat family in a sequence graph (75% coverage).

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	2.6	2.4	3.2	3.7	3.6	3.8	3.7
Burkholderia xenovorans	2.6	2.6	2.0	1.5	1.5	1.3	1.3
Colwellia psychrerythraea	3.2	2.9	3.1	3.1	2.9	3.1	3.1
Desulfitobacterium hafniense	3.3	3.3	3.7	4.1	3.7	3.5	3.3
Desulfovibrio desulfuricans	2.3	2.2	2.9	3.6	3.3	3.3	3.1
Escherichia coli	2.2	2.2	3.0	3.1	3.1	3.1	3.1
Geobacter uraniumreducens	1.8	1.8	2.2	2.5	2.5	2.2	2.1
Gloeobacter violaceus	3.4	3.4	3.5	2.0	2.0	1.4	1.3
Granulibacter bethesdensis	2.0	2.0	2.0	2.1	2.1	2.1	2.1
Haloarcula marismortui	2.2	2.0	2.0	1.6	1.6	1.6	1.6
Halobacterium sp-plasmid pNRC100	2.5	2.2	2.5	2.0	1.5	1.8	1.5
Legionella pneumophila	2.4	2.5	2.8	2.3	2.2	1.6	1.6
Legionella pneumophila-Philadelphia 1	2.6	2.2	3.6	4.1	3.8	4.1	4.1
Methanosarcina acetivorans	3.0	2.9	2.8	1.9	1.9	1.5	1.4
Methylococcus capsulatus	2.2	2.2	3.0	3.6	3.4	2.9	2.7
Nitrosospira multiformis	3.2	3.0	4.1	5.5	5.1	3.5	3.4
Photobacterium profundum	1.7	1.7	1.8	1.9	1.8	1.8	1.9
Pseudomonas syringae	2.2	2.1	1.9	1.6	1.6	1.2	1.2
Pyrococcus furiosus	1.8	1.6	1.6	1.8	1.6	1.8	1.8
Ralstonia solanacearum	2.4	2.4	1.4	1.2	1.2	1.1	1.1
Rhodopirellula baltica	2.2	2.2	2.5	2.4	2.4	2.0	2.0
Roseobacter denitrificans	2.3	2.3	3.1	3.8	3.8	3.5	3.5
Salinibacter ruber	3.0	2.7	2.1	1.1	1.0	1.0	1.0
Shewanella oneidensis	2.0	2.0	1.3	1.4	1.4	1.3	1.3
Sulfolobus solfataricus	2.1	2.0	1.5	1.3	1.3	1.2	1.2

Table A.12: Average number of components per repeat family in a sequence graph (100% coverage).

Combine Factor (τ)	1.0	1.5	2.0	2.5	3.0	3.5	4.0
Bifidobacterium longum	2.5	3.4	3.1	3.7	3.6	3.8	3.7
Burkholderia xenovorans	2.6	2.0	2.0	1.5	1.5	1.3	1.3
Colwellia psychrerythraea	3.1	3.1	3.1	3.1	2.9	3.1	3.1
Desulfitobacterium hafniense	3.3	4.1	3.7	3.9	3.7	3.3	3.3
Desulfovibrio desulfuricans	2.3	2.8	2.5	3.6	3.3	3.3	3.1
Escherichia coli	2.2	3.0	3.0	3.1	3.1	3.1	3.1
Geobacter uraniumreducens	1.8	2.2	2.2	2.5	2.5	2.2	2.1
Gloeobacter violaceus	3.4	3.6	3.5	2.0	2.0	1.3	1.3
Granulibacter bethesdensis	2.0	2.0	2.0	2.1	2.1	2.1	2.2
Haloarcula marismortui	2.1	2.0	2.0	1.6	1.6	1.6	1.6
Halobacterium sp-plasmid pNRC100	2.4	2.8	2.5	1.7	1.5	1.6	1.5
Legionella pneumophila	2.4	2.6	2.9	2.3	2.2	1.6	1.6
Legionella pneumophila-Philadelphia 1	2.5	3.8	3.2	4.1	3.8	4.1	4.1
Methanosarcina acetivorans	2.9	2.9	2.8	1.9	1.9	1.5	1.4
Methylococcus capsulatus	2.2	3.1	2.9	3.6	3.4	2.8	2.7
Nitrosospira multiformis	3.1	4.4	4.2	5.4	5.1	3.5	3.5
Photobacterium profundum	1.7	1.9	1.8	1.8	1.8	1.9	1.8
Pseudomonas syringae	2.1	1.9	1.9	1.6	1.6	1.2	1.2
Pyrococcus furiosus	1.7	1.7	1.6	1.8	1.7	1.8	1.8
Ralstonia solanacearum	2.4	1.4	1.4	1.2	1.2	1.1	1.1
Rhodopirellula baltica	2.2	2.6	2.5	2.4	2.4	2.0	2.0
Roseobacter denitrificans	2.3	3.3	3.1	3.8	3.8	3.6	3.5
Salinibacter ruber	3.0	2.1	2.0	1.1	1.0	1.0	1.0
Shewanella oneidensis	2.0	1.3	1.3	1.4	1.4	1.3	1.3
Sulfolobus solfataricus	2.1	1.6	1.5	1.3	1.3	1.2	1.2

Bibliography

[1] Carolyn Abraham. My dad has Einstein's brain. The Guardian, April 8, 2004.

[2] Martin Aigner. *A Course in Enumeration*, volume 238 of *Graduate Texts in Mathematics*. Springer Verlag, 2007.

[3] Noga Alon, Richard Beigel, Simon Kasif, Stefen Rudich, and Benny Sudakov. Learning a hidden matching. *SIAM J. Comput.*, 33(2):487–501, 2004.

[4] Zhirong Bao and Sean R. Eddy. Automated de novo identification of repeat sequence families in sequenced genomes. *Genome Res.*, 12:1269–1276, 2002.

[5] Susan M. Berget, Claire Moore, and Phillip A. Sharp. Spliced segments at the 5' terminus of adenovirus 2 late mRNA. *Proc. Natl. Acad. Sci. USA*, 74(8):3171–3175, August 1977.

[6] Shahid H. Bokhari and Jon R. Sauer. A parallel graph decomposition algorithm for DNA sequencing with nanopores. *Bioinformatics*, 21(7):889–896, 2005.

[7] Annalisa De Bonis. New combinatorial structures with applications to efficient group testing with inhibitors. *J. Comp. Optim.*, 15:77–94, 2008.

[8] 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.*, 18:810–820, March 2008.

[9] Mark Chaisson, Pavel Pevzner, and Haixu Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[10] Mark J. Chaisson and Pavel A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Res.*, 18:324–330, March 2008.

[11] Ferdinando Cicalese, Peter Damaschke, Libertad Tansini, and Sören Werth. Overlaps help: Improved bounds for group testing with interval queries. *Discrete Applied Mathematics*, 155(3):288–299, February 2007.

[12] Ferdinando Cicalese, Peter Damaschke, and Ugo Vaccaro. Optimal group testing algorithms with interval queries and their application to splice site detection. *Int. J. Bioinformatics Research and Applications*, 1(4):363–388, 2005.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[14] Annalisa de Bonis and Ugo Vaccaro. Improved algorithms for group testing with inhibitors. *Information Processing Letters*, 66:57–64, 1998.

[15] Marian C. Diamond, Arnold S. Scheibel, Greer M. Murphy, and Thomas Harvey. On the brain of a scientis: Albert einstein. *Experimental Neurology*, 88:198–204, 1985.

[16] Reinhard Diestel. *Graph Theory*. Number 173 in Graduate Texts in Mathematics. Springer-Verlag, third edition, 2005.

[17] Ding-Zhu Du and Frank K. Hwang. *Combinatorial Group Testing and its Applications*, volume 12 of *Series on Applied Mathematics*. World Scientific, 2nd edition, 2000.

[18] M. Farach, S. Kannan, E. Knill, and S. Muthukrishnan. Group testing problems with sequences in experimental molecular biology. In B. Carpentieri, A. De Santis, U. Vaccaro, and J. Storer, editors, *Proceedings of the Compression and Complexity of Sequences 1997*, pages 357–367, 1997.

[19] Anne E. Hall, Aretha Fiebig, and Daphne Preuss. Beyond the arabidopsis genome: Opportunities for comparative genomics. *Plant Physiology*, 129:1439–1447, August 2002.

[20] Lenwood S. Heath and Amrita Pati. Genomic signatures in de Bruijn chains. In Raffaele Giancarlo and Sridhar Hannenhalli, editors, *WABI*, volume 4645 of *Lecture Notes in Computer Science*, pages 216–227. Springer, 2007.

[21] Steffen Heber, Max Alekseyev, Sing-Hoi, Haixu Tang, and Pavel A. Pevzner. Splicing graphs and EST assembly problem. *Bioinformatics*, 18 Suppl. 1:S181–S188, 2002.

[22] Yao-Win Hong and A. Scaglione. Group testing for sensor networks: the value of asking the right questions. In *Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers (COSSAC 2004)*, volume 2, pages 1297–1301, 2004.

[23] Yao-Win Hong and A. Scaglione. On multiple access for distributed dependent sources: a content-based group testing approach. In *Proceedings of the IEEE Information Theory Workshop, 2004 (ITW 2004)*, pages 298–303, 2004.

[24] Yao-Win Hong and A. Scaglione. Generalized group testing for retrieving distributed information. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2005)*, volume 3, pages 681–684, 2005.

[25] Ramana M. Idury and Michael S. Waterman. A new algorithm for DNA sequence assembly. *J. Comput. Biol.*, 2(2):291–306, 1995.

[26] Warren R. Jelinek, Thomas P. Toomey, Leslie Leinwald, Craig H. Duncan, Paul A. Biro, Prabhakara V. Choudary, Sherman M. Weissman, Carol M. Rubin, Catherine M. Houck, Prescott L. Deininger, and Carl W. Schmid. Ubiquitous, interspersed repeated sequences in mammalian genomes. *Proc. Natl. Acad. Sci. USA*, 77(3):1398–1402, March 1980.

[27] Z. A. Cox Jr., Xiaorong Sun, and Yuping Qiu. Optimal and heuristic search for a hidden object in one dimension. In *IEEE International Conference on Systems, Man, and Cybernetics. 'Humans, Information and Technology'*, volume 2, pages 1252–1256, October 1994.

[28] Vincent Lacroix, Michael Sammeth, Roderic Guigo, and Anne Bergeron. Exact transcriptome reconstruction from short sequence reads. In *Workshop on Algorithms in Bioinformatics (WABI 2008)*, pages 50–63, 2008.

[29] Yingshu Li, My T. Thai, Zhen Liu, and Weili Wu. Protein-protein interaction and group testing in bipartite graphs. *Intl. J. Bioinf. Res. Appl.*, 1(6):414–419, 2005.

[30] Jacques Mahillon and Michael Chandler. Insertion sequences. *Microbiology and Molecular Biology Reviews*, 62(3):725–774, September 1998.

[31] Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *Workshop on Algorithms in Bioinformatics (WABI 2007)*, pages 289–301, 2007.

[32] Pavel A. Pevzner. *Computational Molecular Biology: an algorithmic approach.* The MIT Press, 2000.

[33] Pavel A. Pevzner and Haixu Tang. Fragment assembly with double-barreled data. *Bioinfomatics*, 17:225–233, 2001.

[34] Pavel A. Pevzner, Haixu Tang, and Glenn Tesler. *De novo* repeat classification and fragment assembly. In *Annual International Conference in Computational Molecular Biology (RECOMB 2004)*, pages 213–222, March 2004.

[35] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA*, 98(17):9748–9753, August 2001.

[36] Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. A new approach to fragment assembly in dna sequencing. In *Annual International Conference in Computational Molecular Biology (RECOMB 2001)*, pages 256–267. ACM Press, 2001.

[37] Eline T. Luning Prak and Haig H. Kazazian Jr. Mobile elements and the human genome. *Nature Reviews*, 1:134–144, November 2000.

[38] Benjamin Raphael, Degui Zhi, Haixu Tang, and Pavel Pevzner. A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Res.*, 14:2336–2346, 2004.

[39] Walter L. Ruzzo and Martin Tompa. A linear time algorithm for finding all maximal scoring subsequences. In *Seventh International Conference on Intelligent Systems for Molecular Biology (ISMB 1999)*, pages 234–241, 1999.

[40] João Carlos Setubal and João Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing, 1997.

[41] Sherman K. Stein. *Mathematics: The Man-Made Universe.* Dover Publications, Inc., 1999.

[42] Nicholas Wade. Human genome appears more complicated. The New York Times, August 24, 2001.

[43] Weili Wu, Yingshu Li, Chih hao Huang, and Ding-Zhu Du. *Data Mining in Biomedicine*, volume 7 of *Springer Optimization and Its Applications*, chapter Molecular Biology and Pooling Design, pages 133–139. Springer, 2008.

[44] Guorong Xu, Sing-Hoi Sze, Cheng-Pin Liu, Pavel A. Pevzner, and Norman Arnheim. Gene hunting without sequencing genomic clones: Finding exon boundaries in cDNAs. *Genomics*, 47(2):171–179, January 1998.

[45] Stefanos A. Zenios and George M. Lawrence A. Weinstock. Pooled testing for HIV prevalence estimation: Exploiting the dilution effect. *Statistics in Medicine*, 17:1446–1467, 1998.

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

[47] Yu Zhang and Michael S. Waterman. An Eulerian path approach to local multiple alignment for DNA sequences. *Proc. Natl. Acad. Sci. USA*, 102(5):1285–1290, February 2005.