Approximate Common Intervals Based Gene Cluster Models

Katharina Jahn

PhD thesis submitted to the
Faculty of Technology, Bielefeld University, Germany
for the degree of Dr. rer. nat.

Referees:

Prof. Dr. Jens Stoye, Bielefeld University
Prof. Dr. Sebastian Böcker, Friedrich Schiller University Jena

März 2010

Thesis defended on Monday, July 19th, 2010.

Committee:
Prof. Dr. Ellen Baake (chair),
Dr. Marília Dias Vieira Braga,
Prof. Dr. Jens Stoye (referee),
Prof. Dr. Sebastian Böcker (referee).

Gedruckt auf alterungsbeständigem Papier nach DIN-ISO 9706
Printed on non-aging paper according to DIN-ISO 9706

Zusammenfassung

Durch die zunehmende Verfügbarkeit von vollständig sequenzierten und assemblierten Genomen enstand in den letzten Jahren ein neues Teilgebiet der komparativen Genomik, in dem Genome verschiedener Spezies bezüglich der enthaltenen Gene und deren Anordnung auf den Chromosomen verglichen werden. Ziel solcher Studien ist es, die Funktionen von einzelnen Genen und Genverbünden aufzudecken und ein besseres Verständnis für die auf Genomebene wirkenden evolutionären Prozesse zu gewinnen. Im allgemeinen beobachtet man bei diesen Vergleichen eine graduelle Randomisierung der Genanordung und des Gengehalts, die um so stärker ausgeprägt ist, je geringer der Verwandtschaftsgrad der untersuchten Spezies ist. Einige lokale Bereiche bleiben jedoch auch über große evolutionäre Distanzen sehr gut konserviert. Dieses wiederholte gemeinsame Auftreten von bestimmten Genen ist ein zuverlässiger Indikator für deren funktionelle Interaktion. Die automatisierte Aufdeckung dieser Genverbünde, die häufig als Gencluster bezeichnet werden, erweist sich als anspruchsvolles Problem, da man häufig mit unvollständigen Konservierungsmustern konfrontiert ist, die durch lokale Genumordnungen, den Verlust einzelner Genvorkommen, Unterbrechungen durch unbeteiligte Gene oder fehlerhafte Homologie-Bestimmung entstanden sind. Diese Doktorarbeit beschäftigt sich mit der formalen Modellierung solcher approximativ konservierten Gencluster, ihrer effizienten Berechnung sowie mit sich daraus ergebenden Anwendungsgebieten in der komparativen Genomik.

Die in dieser Doktorarbeit entwickelten Gencluster-Modelle, die unter dem Begriff *approximate common intervals* zusammengefasst werden können, erweitern das etablierte Modell der *common intervals* dahingehend, dass nicht mehr nur lokale Genumordnungen innerhalb eines Clustervorkommens toleriert werden, sondern auch Insertionen und Deletionen von Genen. Diese Verallgemeinerung führt dazu, dass ein Gencluster nicht mehr durch eine eindeutige Menge von Genen beschrieben werden kann, die in allen Clustervorkommen vollständig und konsekutiv vorkommt. Stattdessen wird eine Konsensus-Menge definiert, die die unterschiedlichen Clustervorkommen bestmöglich repräsentiert. Hierfür wurden basierend auf der symmetrischen Mengendistanz zwei Konsenus-Modelle entworfen: Die *Median Gene Cluster* minimieren die Summe der paarweisen Distanzen zwischen der Konsensus-

Menge und dem Gengehalt der approximativen Vorkommen, während die *Center Gene Cluster* die maximale paarweise Distanz minimieren. Von beiden Modellen wurde zudem eine referenzbasierte Variante entwickelt, die statt der optimalen Konsensusmenge den bestmöglichen Repräsentanten auswählt, dessen Gene konsekutiv auf einem der untersuchten Genome vorkommen.

Das Verwenden einer Konsensusmenge bzw. eines Referenzvorkommens stellt einen entscheidenden Fortschritt gegenüber anderen Verallgemeinerungen der *common intervals* wie z.B. den *max-gap* oder *r-window* Genclustern dar. In diesen Modellen enthält ein Gencluster nur solche Gene, die in allen Cluster-Vorkommen gemeinsam auftreten, was beim Vergleich einer großen Anzahl von Genomen durch den Verlust einzelner Genvorkommen leicht dazu führen kann, dass nur ein Teil der kolokalisierten Gene in einem Gencluster zusammengefasst wird und dadurch scheinbare Lücken in den Cluster-Vorkommen auftreten. Es konnte an realen Datensätzen gezeigt werden, dass die oben genannten Effekte nicht auftreten, wenn *approximate common intervals* basierte Gencluster-Modelle verwendet werden, und dass dadurch komplexere Gencluster gefunden werden können.

Neben der Modellierung lag der Schwerpunkt dieser Doktorarbeit auf der Entwicklung von Algorithmen, die die effiziente Vorhersage von Genclustern unter den neuen Modellen ermöglichen. Zunächst wurde für den bekannten *Connecting Intervals* Algorithmus zur Berechnung von *common intervals* eine einfache Methode beschrieben, durch die der Speicherplatzbedarf von quadratischer auf lineare Abhängigkeit von der Eingabegröße reduziert werden kann. Darauf aufbauend wurden neue Algorithmen für die *approximate common intervals* basierten Gencluster-Modelle entwickelt. Für die referenzbasierten Modelle wurde ein effizienter Algorithmus gefunden, der quadratisch von der Eingabegröße und einer vom Benutzer vorgegebenen Distanz-Obergrenze abhängt. Für die *Median Gene Cluster* und die *Center Gene Cluster* wurde durch das Verwenden einer Filter-Technik und mehrerer algorithmischer Optimierungen der exponentielle Suchraum soweit eingeschränkt, dass ein großer Parameter-Raum in angemessener Zeit abgesucht werden kann, während die *worst-case* Laufzeit exponentiell mit der Anzahl verglichener Genome anwächst.

Weitere Punkte, die in dieser Doktorarbeit behandelt wurden, sind die statistische Signifikanz-Analyse von Genclustervorhersagen, die Anwendung der vorgestellten Methoden in der Gencluster-Vorhersage in prokaryotischen Genomen sowie die Entwicklung einer distanzbasierten Methode zur Phylogenie-Rekonstruktion.

Abstract

With the increasing availability of completely sequenced and assembled genomes, gene-order based comparisons of whole genomes have recently become an important field in comparative genomics. The aim of such studies is to reveal functional coupling between genes, and to gain a better understanding of large-scale evolutionary processes. In general, we observe in such studies a gradual randomization of gene order and gene content that is contrasted by a number of well-conserved segments that remain co-located across species. Such local aberrations from genome randomization are known to provide highly informative signals for functional analysis. However, incomplete conservation patterns caused by micro-rearrangements, gene losses, gene insertions, as well as errors in the homology assignment turn gene cluster detection into a computationally hard problem.

This thesis is about the formal modeling of approximately conserved gene clusters, their efficient computation and applications in comparative genomics. The gene cluster models developed in this work can be summarized under the term *approximate common intervals*. They extend the established *common intervals* model to deal not only with micro-rearrangements within cluster occurrences but also with insertions and deletions of genes. Due to this generalization, a gene cluster is no longer determined by a specific set of genes which occurs everywhere as a complete and consecutive block. Instead a consensus set is defined that best represents a set of similar cluster occurrences. For that purpose, we defined two consensus models based on the symmetric set distance: *Median Gene Clusters* choose the consensus gene set to minimize the overall distance to the approximate occurrences, while *Center Gene Clusters* minimize the largest pairwise distance between the consensus set and any of the approximate occurrences. For both models, we developed also a reference-based version that picks the representative directly from the set of similar cluster occurrences. Both approaches, the optimized consensus set and the reference occurrences improve substantially over earlier generalizations of the common intervals model, like *max-gap* or *r-window* gene clusters. In the latter models, only those genes belong to a cluster that occur in each approximate occurrence. When comparing a large number of genomes, the loss of individual genes in different cluster occurrences has the effect that only a part of the co-localized genes is combined

into a gene cluster and that artificial gaps are introduced into cluster occurrences. Our experimental results show that these effects do not occur in approximate common intervals based gene cluster models and that more complex conservation patterns can be detected.

Besides the formal modeling of gene clusters, we focused in this thesis on the development of efficient algorithms for the prediction of gene clusters under the novel models. We introduced a simple extension of the *Connecting Intervals* algorithm for the computation of common intervals to reduce its space complexity from quadratic to linear dependency on the input size. Based on these results, we developed new algorithms for approximate common intervals computation. For the reference-based version, we devise an efficient polynomial-time algorithm that depends quadratically on the input size and a user-defined distance threshold. For median and center gene clusters we use filter techniques and algorithmic optimizations that restrict the exponential search space sufficiently to search a large parameter range in reasonable time, albeit the asymptotic runtime of our approach grows exponentially with the number of genomes compared.

Further topics of this thesis are the analysis of the statistical significance of predicted gene clusters, the application of the developed approaches to gene cluster prediction in prokaryotic genomes and the development of a distance-based approach to phylogeny reconstruction.

Acknowledgments

I am very grateful to the many people who supported and encouraged me during the past years. Without them the development of this thesis would not have been such an invaluable and successful experience.

First of all, I would like to thank my advisor, Jens Stoye, who gave me the opportunity to do my PhD at the Genome Informatics group at Bielefeld University. Over the past years, he was always ready to give valuable input and new insights and allowed me enough freedom to pursue my own ideas. Also, I owe thanks to Sebastian Böcker for our long-term collaboration and for reviewing this thesis. Additionally, I would like to thank Ellen Baake and Marília Braga for being part of my dissertation committee. Special thanks go to my former bachelor student Leon Kuchenbecker who still continues to work with me on the gene cluster project, and who did a great job in wrapping a graphical user interface around the C code of my gene cluster algorithms.

Furthermore, my thanks go to all the former and present members of the Genome Informatics group at Bielefeld University for the interesting discussions, both on- and off-topic, the great atmosphere and the fun. Many of them became good friends over the past years: Roland Wittler, Inke Herms, Marília Braga, Nondas Fritzilas, Martin Milanič, Yasmín Ríos-Solís , José (Zé) Augusto Amgarten Quitzau, Peter Husemann, Wolfgang Gerlach. Some of them deserve extra credit for proof-reading parts of this thesis: Roland, Inke, Marília, and Peter. I am also very grateful to our secretary Heike Samuel for handling all the administrative stuff.

With gratitude, I would like to acknowledge the financial and academic support of the International NRW Graduate School in Bioinformatics and Genome Research at Bielefeld University.

I am indebted to my parents and my brother Bernd for always supporting and encouraging me. Finally, I should not forget to mention my friends who always reminded me that there is a life beyond algorithms and gene clusters.

München, June 2011

Katharina Jahn

Contents

Chapter 1

Introduction

In the past century, a series of breakthrough discoveries provided unprecedented insights into the basic mechanisms of life. Important roles played the understanding how the inheritable characteristics of a living organism are encoded in its genome in form of genes, the disclosure of the physical genome representation in DNA molecules, and the discovery of the mechanisms by which the encoded genetic information is used in the living organism.

Current genome research adds further insights into these mechanisms as it reveals on a large scale the function of individual genes, gene products and non-coding regions, elucidates their interaction and compares them across genomes. Such sequence based analyses are possible since the 1970s when the first techniques for the sequencing of short DNA segments were developed. Since then, advancements in the technology made sequencing ever faster and cheaper and increased drastically the amount of available sequence data, in the meantime not only of DNA fragments but of whole genomes. Still, a few years ago only a small number of model organisms were completely sequenced, but nowadays the genomes of a broad range of species are available and their number grows rapidly as illustrated in Figure 1.1. According to the GOLD database the sequencing of 1100 genomes is finished at present, while more than 4500 further sequencing projects are currently under way [54].

With this abundance of data a new line of genome research opened up, which is referred to as comparative genomics. This young discipline aims at the analysis and comparison of whole genomes based on changes in their large-scale structure to gain insights into the functionality of genomes as a whole and the evolutionary processes that act on them. It has long been known that genomes evolve not only by point mutations of the nucleotide sequence but also by means of large-scale modifications of the genome organization. These changes are caused by rearrangement operations like inversions and transpositions of chromosome segments, chromosome fusion and fission, as well as duplication processes that can affect the whole genome,

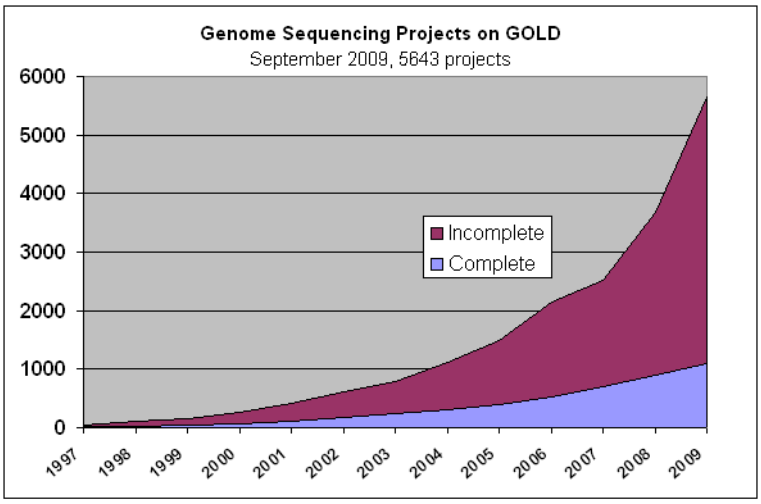


Figure 1.1: The number of completed and ongoing sequencing projects in the GOLD database. Figure taken from `http://genomesonline.org/gold_statistics.htm`

chromosomes, chromosome segments as well as single genes. These duplications are typically followed by gene losses and sometimes the development of new functionalities in the duplicated genes, as selective constraints usually act only on the maintenance of a single gene copy.

The analysis of genome rearrangement was pioneered by Dobzhansky and Sturtevant [22] in the first half of the 20th century, long before the first genome was sequenced or even the structure of DNA was discovered. In modern times, genome rearrangement studies were introduced by David Sankoff [77] and Nadeau and Taylor [57] with the objective to infer evolutionary distances between species based on the amount of rearrangement that took place between their genomes. This approach is based on the implicit assumption that after speciation events the genomes of the new species are reshuffled over time in a random process. It is then assumed that the most parsimonious sequence of rearrangement operations that transforms two genomes into each other is a measure for their divergence time. Genome rearrangement studies are a highly active field of research. For a recent overview of the field, we refer to the surveys of Li et al. [52], or Bourque and Zhang [13], or the recent book by Fertin et al. [13].

The second major line in comparative genomics is the study of gene clusters. Especially in the comparison of prokaryotic genomes, typically a pattern of very low overall gene order conservation is observed that is contrasted by a number of small segments with highly conserved gene order or at least gene content. An example of such a setting is visualized in the dot plot given in

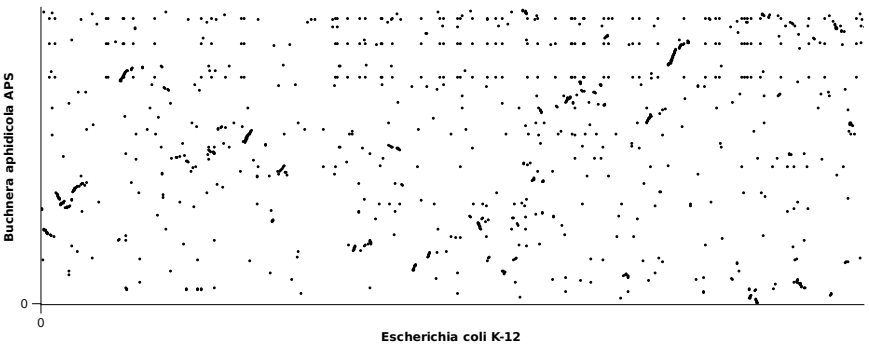


Figure 1.2: Visualization of gene order conservation between the two γ-proteobacteria Escherichia coli (4183 genes) and Buchnera aphidicola (564 genes). Dotted diagonals indicate local conserved gene content and order. (Dotted horizontal lines indicate large gene families in Escherichia coli)

Figure 1.2. These conserved segments correspond usually to functionally related genes, like operons, for which joint transcription is favorable. This conservation pattern shows that whole genome evolution of prokaryotes is driven by two adverse processes: on the one hand, a random drift of overall gene order and gene content, and on the other hand, strong selective constraints on the local organization of certain genome segments. It is therefore believed that such local aberrations from the genome randomization provide highly informative signals for functional analysis. The most popular objectives are operon prediction [17, 24, 68, 74, 86, 87], the general analysis of genome organization in the presence of functional constraints [38, 39, 44, 58, 82, 81] and the disclosure of horizontal gene transfer [35, 50].

For this type of studies, gene clusters are predicted automatically based on a (more or less) formal notion what kind of conservation patterns are interesting and which are not. The outcome of such approaches, i.e. the conserved segments, are referred to as gene clusters in the literature. This term is somewhat misleading, as there is *per se* no guarantee that these conserved segments are indeed gene clusters in the biological sense. A specific conservation pattern may as well occur by chance or due to lack of divergence time of the compared species. However, as it is common practice, we stick to this imprecise terminology in this thesis keeping in mind that a "gene cluster" is in fact a gene cluster prediction. To verify that it is also a gene cluster in the biological sense, experimental evaluation is necessary which is a time and cost intensive endeavor. However, a pre-evaluation of gene cluster predictions can be employed to identify by statistical means those candidates for which other explanations than selective constraints are the least likely.

Gene cluster prediction focused originally on prokaryotic genomes, as they were previously known to contain operons. However, in recent years,

it was discovered that also in eukaryotic genomes conserved structures similar to operons exist which are believed to build similar functional units. Therefore, gene cluster detection is nowadays also applied to this genome type [66].

Based on the model of genome evolution by means of rearrangement processes, the conservation of gene clusters is also used as a measure for the phylogenetic relationship between species. Such analyses are based on the assumption that the amount of genome rearrangement that took place between species is reflected in the number and size of segments that are still conserved between the genomes such that it can be used as a measure for their evolutionary distance. This approach works independently of a specific model for genome rearrangement and thereby avoids the problems that are still connected to such a modeling due to the fact that the underlying biological processes are not yet completely understood.

We give an overview of different gene cluster models later on in Section 1.2. As these models require the formalization of a genome model that represents genomes at a suitable resolution for gene cluster detection, we address this topic afore.

1.1 Genome Model

To compare genomes on a more abstract level than nucleotide sequence, the first thing to do is to decide on the type of genomic marker that should be used to model the studied genomes. Most typically these markers are simply genes, but other levels of granularity can be found in the literature like syntenic regions [65] or protein domains [62]. In the following, we concentrate on genes as markers reducing chromosomes to linear (or circular) arrangements of genes. However, it should be mentioned that for the algorithmic approaches introduced in this thesis, the specific type of the compared sequences is of no importance.

The next step towards a gene order based model is the identification of the genes on the genomes under consideration. A variety of approaches for gene detection have been developed in the past years. We give a short survey on this topic in Section 1.1.1. Once the genes and their order on the genomes are determined, we need to establish gene homologies to define which genes we consider to be conserved among genomes. Also for this step one can choose between a large number of approaches. We review the most popular ones in Section 1.1.2.

The next step towards a gene order based genome representation is the decision between the sequence and permutation based genome model. While the sequence model allows for multiple occurrences of genes and differences in the gene content of the compared genomes, the permutation model requires a one-to-one mapping between the genes of the different genomes, i.e.

every gene occurs exactly once in each of the studied genomes. This restriction originates from genome rearrangement theory where it was introduced for complexity reasons. However, from the biological side, this reduction is questionable as multiple occurrences of genes and differences in the genome content across species are not extraordinary. For the permutation model, such genomes would have to be artificially transformed into one-to-one mappings by removing genes. Although most known gene cluster models were originally defined on permutations, they were later on extended to the sequence model, as duplicate genes and differences in the gene content are of minor importance for the computational complexity of gene cluster detection. In the following, we focus exclusively on the sequence model as we believe that this is a more realistic genome representation.

Finally, a decision has to be made whether gene orientation should be represented in the genome model. DNA molecules are double strands. They consist of two anti-parallel chains of complementary nucleotides, often referred to as the plus and minus strand. Both strands encode genes that form templates for transcription, but they are read in opposite directions. This means that adjacent genes located on the same strand can be transcribed together while genes on opposite strands can not. Joint transcription of genes is often observed in operons and may therefore seem to be a reasonable constraint in prokaryotic gene cluster prediction. However, it has been observed that functional units can be distributed over both strands while being located in close proximity [88, 53]. Based on this observation it is safer to allow for changes in the gene orientation within a gene cluster and most gene cluster models follow this idea. Conserved gene orientation can still be established in a post-processing step, if it turns out to be relevant for further analysis.

1.1.1 Gene finding

Besides costly gene detection in lab experiments, there are different computational approaches to determine the set of genes present on a chromosome as well as their linear arrangement. First, it is often possible to infer gene occurrences based on sequence similarity by mapping the nucleotide sequences of known genes to newly sequenced genomes. Alternatively, one can infer from the presence of gene products in cells, mRNAs or proteins, that the corresponding gene has to occur in the genome and search for its location using sequence alignment techniques. Finally, there are probabilistic approaches that predict genes in an *ab initio* style from a nucleotide sequence based on signatures that are typically associated with coding regions, like promoter sequences in prokaryotes or CpG islands in eukaryotes. Popular tools for this purpose are GLIMMER [76, 19] and GeneMark [9] for prokaryotic genomes, and GENSCAN [14], geneid [61] and SNAP [46] for eukaryotic genomes. Naturally, there is no guarantee that computational approaches detect all genes

present in a genome. Also some of their predictions may be false positives. Both types of errors affect the quality of the genome representation.

1.1.2 Partitioning genes into homology families

Once the genes and their arrangement on the studied genomes are determined, we need to establish homologies between the genes within and across species to enable genome comparison.

Theoretically, the objective for homology assignment is clear: two genes are homologs if they originate from a common ancestral gene. In the literature this concept is further subdivided to distinguish genes that originated from a single gene in a speciation event (orthologs) from genes that originated from a gene duplication in a common ancestor (paralogs) [26]. For the sequence model, this distinction is of minor importance. So we do not elaborate further on this topic. In practice, homology assignment alone turns out to be challenging enough. In absence of other sources of knowledge, it is usually based solely on sequence similarity which has proven to be a reliable indicator for gene homology.

The process begins with an all-against-all comparison of the genes present in the studied genomes. Pairwise similarity scores are obtained by local sequence alignment computation using for example the BLAST [1] software either directly on the nucleotide sequences or on the derived amino acid sequences.

The resulting similarity scores are the basis of a clustering process to establish gene families. For this step, one can choose among various approaches. Typically these approaches first build families based on sequence similarity [18, 79, 64] and then edit the clusters by merging or splitting families, or removing/adding single genes. Other approaches start the clustering from reciprocal best hits that are then extended to larger families. The well-known COG database [83] and the eggNOG database [42] rely on triangles of reciprocal best hits as starting points for the clustering process. Alternative approaches employ hierarchical clustering methods [47, 67, 89]. Many other approaches can be found in the literature that are based for example on Markov clustering [16, 3], random walks [59] or tree reconciliation [72]

This abundance of approaches illustrates the importance of homology detection but shows also that this problem is not ultimately solved. As with gene prediction, one should be aware that any homology assignment may contain errors which affect the quality of the genome representation.

1.2 Gene cluster models

The next step in gene cluster detection is the decision on a specific gene cluster model for the subsequent genome comparison. At present, there is no universally accepted model that covers all relevant gene cluster features.

This is largely due to the fact that it is not finally concluded what exactly these features are. Moreover, already for those features regarded as essential, it turns out difficult to combine them into a single gene cluster model [34].

In the literature, two major lines of gene cluster detection can be distinguished. The first comprises rather pragmatic approaches that favor fast prediction results over the use of formal cluster definitions and rigorous objective functions. The number of these approaches grew rapidly in the past decade and we refrain from giving an exhaustive survey. An overview of the most popular approaches can be found in [75].

The second line of gene cluster detection is characterized by a clear separation between the model definition and the actual gene cluster computation. Such approaches have the advantage that the properties of the predicted gene clusters are formally defined. This simplifies the comparison between different approaches, the statistical analysis of the predictions and to some extent also the assessment of the chosen model against the biological reality.

In the following, we give an overview of formal gene cluster definitions. Similar surveys are given in [6, 34] All these models have in common that gene clusters consist of two ingredients: a set of genes that constitute the cluster and its occurrences on the studied genomes, i.e. the genome segments that contain all or some of the cluster elements.

1.2.1 Co-linear gene clusters

The most rigorous gene cluster definition requires gene clusters to be perfectly conserved in respect to both gene content and gene order. This means that contiguous regions need to occur in the studied genomes that consist of the same genes in the same or reversed order. While co-linear gene clusters are very easy to detect — they correspond to conserved substrings —, many interesting gene clusters will be overlooked due to micro-rearrangements or small changes in the gene content by gene losses or insertions.

1.2.2 Common intervals

A slightly more relaxed gene cluster definition are common intervals. These require still a perfectly conserved gene content but allow for changes in the gene order. Different variants of common intervals are described in the literature that differ in the degree of gene order fixation. In the general common intervals model any order is allowed, while framed common intervals [8] and nested common intervals [11] restrict the inner order of cluster occurrences to some extent.

Computing common intervals is not a difficult task. There exist a number of efficient approaches for both genome representations, permutations and strings [85, 32, 31, 78, 21, 20]. However, this gene cluster model is still too strict for practical purposes. Gene losses and gene insertions occur fre-

quently even in well-conserved gene clusters. These small deviations in the gene content prevent such clusters from being detected under the common intervals model. In practice, post-processing techniques can be used to join closely located small common intervals to larger approximately conserved gene clusters [79].

1.2.3 r-window gene clusters

The most simple approach to a formal model of approximate gene clusters are r-windows [23]. The algorithmic idea behind this model is to scan the given genomes for windows of a fixed size r that share at least $k \leq r$ genes. For $k = r$ this gene cluster model is equivalent to common intervals. In a dot plot representation, an r-window gene cluster in two genomes corresponds to a square with side length r that contains at least k dots.

The crucial point in this approach is the choice of the parameters. If the ratio $\frac{k}{r}$ is too big, some interesting gene clusters will be missed, and if the ratio is chosen too small, gene clusters will be predicted that are in fact noise. For multiple genome comparison, there is the problem that the set of genes that constitutes a gene cluster becomes smaller and smaller. Defined as a minimal consensus, i.e. the set of genes that occur in the respective window of each genome, the value of k needs to be decreased to a disproportionately low value. The computational complexity of this approach increases exponentially with the number of compared genomes.

1.2.4 Max-gap clusters

An alternative model for approximate gene clusters are max-gap clusters [7, 30], also refered to as *gene teams* or *homology teams*. This model specifies no fixed length of gene cluster occurrences, but uses a parameter g that limits the maximum size of a gap that may occur between two successive occurrences of genes from a gene cluster. For $g = 0$ this model is equivalent to common intervals. A point of criticism against this approach is the fact that the gap size in cluster occurrences is constrained only individually for each gap, while there is no prevention against sparse gene clusters that have a gap of maximum size between each successive pair of gene occurrences from the cluster. There is no biological evidence that numerous small gaps are more likely to occur in real gene clusters than a few larger ones. Like with r-window gene clusters, the gene set that constitutes a max-gap gene cluster is a minimum consensus which can lead to undesired effects in multiple genome comparison. While the computation of max-gap clusters is possible in polynomial time for pairwise genome comparison, the computational complexity increases exponentially with the number of compared genomes, as was the case for r-window gene clusters. Despite these problems, max-gap clusters are currently the most-widely used model for approximate gene clus-

ters. Recently, a graph-based variant of max-gap clusters was proposed [90] that uses a parameter to constrain the amount of reshuffling that is allowed within a gene cluster.

1.2.5 Approximate common intervals based models

The most recent models for approximate gene clusters can be summarized under the term *approximate common intervals*, as they are based on an extension of the concept of common intervals to allow for a certain amount of differences in the gene content of similar intervals. The basic idea is to define a maximum distance between a consensus gene set and its approximate occurrences that can be freely distributed over gene losses and insertions located anywhere in the approximate cluster occurrences. In contrast to previous approaches, the consensus is not a minimal consensus but a set of genes that is either close to the gene content of its approximate occurrences [15] or optimized in the sense that the total distance to its approximate occurrences is minimized [71, 12].

The search space of these approaches increases exponentially either with the number of allowed deviations in the gene content or the number of compared genomes. However, using efficient filter techniques, we were able to show that such approaches are feasible for a broad range of parameter settings [12]. In this thesis, we see that for a restricted type of consensus set, approximate common intervals based gene cluster computation is even possible in polynomial time.

1.3 Thesis overview

In this thesis, we study novel approaches to approximate gene cluster computation. We extend the concept of common intervals to approximate common intervals by allowing for incomplete conservation of character content, define different gene cluster models based on approximate common intervals and develop efficient computational approaches that allow for the detection of approximate gene clusters in multiple genomes. We evaluate the presented approaches experimentally and design a statistical framework to estimate the significance of gene cluster predictions under approximate common intervals based gene cluster models.

After a general introduction of the basic notation and concepts, we introduce in Chapter 2 the approximate common intervals based gene cluster models studied in this thesis. The following four chapters are dedicated to the algorithms for gene cluster detection that were developed in the scope of this PhD project: In Chapter 3, we revisit the problem of common interval computation and show that the space complexity of the well-known Connecting Intervals Algorithm can be reduced from quadratic to linear dependence on the input size. Then, in Chapters 4 and 5, we show that a restricted

type of approximate gene clusters that are based on reference occurrences can be computed efficiently using an extension of the Connecting Intervals Algorithm. In Chapters 6 and 7, we propose two algorithms for the general approximate gene cluster detection problem that apply filter techniques to narrow down the exponential search space.

In Chapter 8, we discuss the significance of gene clusters detected under the approximate common intervals model and propose a test statistic to assess the probability that detected gene clusters occur by chance.

In Chapter 9, we apply the presented algorithms to different genomic datasets. First, we assess the general applicability of the presented approaches to multiple genomes and compare the different algorithms among each other and to related approaches. Then, we analyze the predicted gene clusters from a biological point of view. In Chapter 10, we explore a different field of application for approximate gene cluster detection, namely whole genome based phylogeny reconstruction. We propose a simple distance measure based on approximate gene cluster conservation and compare our approach to other whole genome based phylogeny reconstruction methods. We conclude the thesis in Chapter 11 with some final considerations about approximate gene cluster detection and an outlook on future directions of the field.

Parts of this thesis have been published in advance. The results on Reference Gene Clusters (Chapter 5) and the algorithms of Chapter 4 appeared in [40]. The computation of Median Gene Clusters (Chapter 6) and the basic idea of Center Gene Clusters (Chapter 7) were published in [12], while the algorithm for center computation is part of [36]. Moreover, the results on phylogeny reconstruction appeared in [41]. Finally, the presented algorithms are integrated into a software package that is publicly available at http://bibiserv.techfak.uni-bielefeld.de/gecko/.

Chapter 2

Approximate common intervals based models

In the previous chapter, we became already informally acquainted to gene cluster models based on common intervals. Before we can give an expedient description of our approaches to gene cluster detection under these models, we need to formalize the concepts used in Section 1.2.5 and need to define the underlying algorithmic problems. We address these prerequisites in this chapter: After a general introduction of definitions and notations, we formalize the problems of detecting perfectly conserved and different types of approximately conserved gene clusters.

2.1 Basic notations and definitions

For the most part of this thesis, we assume that the genomes studied are linear and uni-chromosomal. This restriction is primarily introduced to simplify notation. We will see later on that the extension of the different approaches to cover circular and multi-chromosomal genomes is straight-forward. For the time being, we use the terms "genome" and "chromosome" interchangeably to refer to a uni-chromosomal linear genome.

We model a chromosome as a *string*, i.e. a finite, ordered sequence of characters, over a finite alphabet Σ of gene family ids. Each index position in such a string stands for the respective gene locus on the underlying chromosome, while a character occurring at a position constitutes the affiliation of the corresponding gene to a certain homology family as assigned in the partitioning process described in Section 1.1.2. An example string over the alphabet $\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$ is shown in Figure 2.1.

Given such a string S, we denote its *length*, the number of characters in the string, by $|S|$. A special case is the *empty string* denoted as ε that contains no elements, i.e. $|\varepsilon| = 0$. We use the notation $S[i]$, $1 \leq i \leq |S|$, to refer to the ith character of S. Accordingly, we say that character $c \in \Sigma$

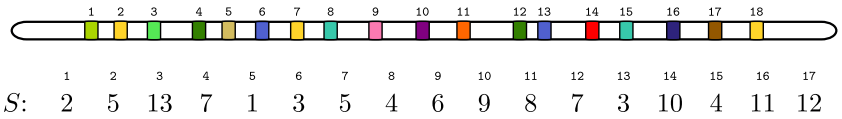


Figure 2.1: String representation of a linear chromosome.

occurs at position i in S if and only if $S[i] = c$. To capture the character content of S regardless of sequential arrangement and multiple character occurrences, we define the *character set* of S as

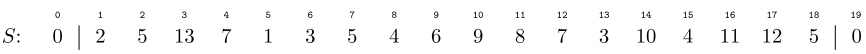
$$\mathcal{CS}(S) \;=\; \{S[i] \mid 1 \leq i \leq |S|\}.$$

This describes the complete set of gene family ids occurring on the respective chromosome, informally referred to as *gene content*. The character set of the empty string ε is the empty set, $\mathcal{CS}(\varepsilon) = \emptyset$.

Dealing with strings of gene family ids, it is reasonable to assume $|\Sigma| \in \Theta(|S|)$, as the size of these families within a genome is typically in $O(1)$. Additionally, we assume without loss of generality that $\Sigma = \{1, \ldots, |\Sigma|\}$. This is feasible because such an encoding can always be constructed in an $O(|S| \log(|S|))$ preprocessing step which is subsumed by the complexities of all algorithms presented in this thesis.

Among other things, this standardization of Σ allows for a convenient representation of all (character) sets $C \subseteq \Sigma$ based on *bit strings*. These are special strings defined over a two character alphabet $\{0, 1\}$. We call a bit string of length $|\Sigma|$ *bit representation* of a $C \subseteq \Sigma$, if the bit at each position i, $1 \leq i \leq |\Sigma|$, is 1 if character i occurs in C and 0 otherwise.

To simplify notation in the following definitions and the algorithms described in the next chapters, we introduce a terminal character $0 \notin \Sigma$ that virtually extends a string S on both ends, i.e. $S[0] = S[n+1] = 0$, $n = |S|$. For our example string S, this looks as follows:



However, we do not treat these positions as proper elements of S, in the sense that they do not count for the string length and intervals of S can not cover either of them. Their purpose is to catch overruns of the string boundaries.

In the following, we need to refer to the gene content of segments of a chromosome. For that purpose, we define $S[i, j]$ to be the *substring* of S that starts with its ith and ends with its jth character, $1 \leq i \leq j \leq |S|$. The corresponding index interval $[i, j]$ is called a *location* of a character set $C \subseteq \Sigma$ if and only if $C = \mathcal{CS}(S[i, j])$. In case, it is not clear from the context which string an index interval refers to, we extend $[i, j]$ to $[i, j]_S$ to indicate

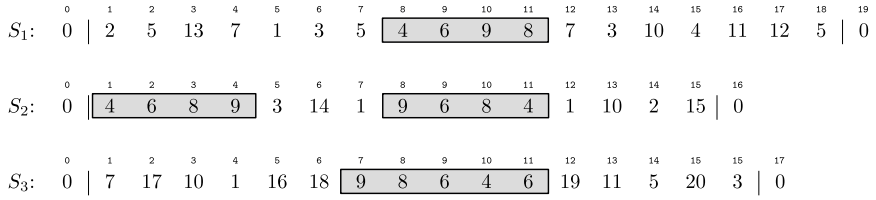


Figure 2.2: Common intervals in three strings.

that we refer to the respective interval in S. Throughout this thesis, the terms "substring $S[i,j]$" and "interval $[i,j]$" (respectively "interval $[i,j]_S$") will be used interchangeably. In particular, when writing about "characters" or the "character set" of an interval, we refer to the respective substring. Both terms describe the set of gene family ids occurring in the chromosome segment defined by the interval. We say an interval $[i,j]$ is *empty* if $j < i$. Every empty interval is a location of the empty character set $C = \emptyset$.

We distinguish different types of intervals in a string S. We call an interval $[i,j]$ with $j \geq i$ *left-maximal* if $S[i-1] \notin \mathcal{CS}(S[i,j])$, *right-maximal* if $S[j+1] \notin \mathcal{CS}(S[i,j])$ and *maximal* if it is both left- and right-maximal.

Example 1 *Interval* $[4,12]$ *is left- but not right-maximal in string* S. *A right- but not left-maximal interval is* $[5,13]$. *An example of a maximal interval is* $[4,13]$, *while interval* $[5,12]$ *is neither left- nor right-maximal.*

Using a comparative approach for gene cluster detection, we typically deal with several genomes at a time. In the following, we assume that these are given as a set of $k \geq 2$ strings denoted as $\mathcal{S} = \{S_1, \ldots, S_k\}$. The alphabet of $\mathcal{S}$ is the union of the alphabets of the individual strings. Therefore, its size is in $O(k \cdot |S_{max}|)$, where S_{max} is the longest of the k strings. We now have all prerequisites to define the concept of common intervals in multiple strings:

Definition 1 *Given a set of strings* $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, *we call a* k*-tuple of maximal intervals* $([i_1,j_1]_{S_1}, \ldots, [i_k,j_k]_{S_k})$ ***common intervals*** *of* $\mathcal{S}$ *if and only if there is a* $C \subseteq \Sigma$ *with:*

$$C = \mathcal{CS}(S_1[i_1,j_1]) = \ldots = \mathcal{CS}(S_k[i_k,j_k]).$$

We refer to such a C as a *character set of common intervals*. In terms of gene cluster detection, a character set of the form of C constitutes a *perfect gene cluster* and the associated k-tuple constitutes a combination of perfect occurrences on the input genomes.

Example 2 *The set of strings* $\mathcal{S} = \{S_1, S_2, S_3\}$ *of Figure 2.2 contains two* k*-tuples of common intervals with character set* $C = \{4,6,8,9\}$*:*
$([8,11]_{S_1}, [1,4]_{S_2}, [7,11]_{S_3})$ *and* $([8,11]_{S_1}, [8,11]_{S_2}, [7,11]_{S_3})$.

We want to represent in the same style also gene clusters that are only approximately conserved. As a first step to this end, we introduce a set distance measure to quantify differences between a gene cluster and the gene content of one of its approximate occurrences. In this thesis, we choose the *symmetric set distance* for this purpose, which defines the distance between two sets C and C' as the cardinality of their *symmetric difference*:

$$D(C, C') \;=\; |C \setminus C'| + |C' \setminus C|.$$

We adopt this measure for two reasons: First, it constitutes a metric and therefore meets all intuitive notions of a distance measure such as validity of the triangle inequality. Second, in context of gene clusters, it corresponds to a plain summing over the genes deleted plus the genes inserted in a cluster occurrence. (Due to the representation of gene clusters as sets — not multi-sets — of characters, multiple insertions, respectively deletions, of genes of the same type count as single modifications.) In absence of a general model of gene cluster evolution, choosing such a basic distance measure appears to be a reasonable choice. However, we discuss alternative set distances in Chapter 11.

Using a set distance measure, such as the symmetric set distance D, we can extend the concept of character set locations towards approximate conservation: Given an integer $\delta \geq 0$, we say an interval $[i, j]$ in a string S is a *δ-location* of a character set C if and only if $C \cap \mathcal{CS}(S[i, j]) \neq \emptyset$ and $D(C, \mathcal{CS}(S[i, j])) \leq \delta$. For general $\delta > 0$, we call $[i, j]$ an *approximate location* of C. For $\delta = 0$, we use the terms location, *0-location* and *perfect location* synonymously.

The next step to model approximate gene clusters is to constrain the maximal distance between a gene cluster and the gene contents of its approximate occurrences that we allow to still speak of a conserved gene cluster. There are two general ways to limit the distance between a character set C and a set of character sets $\mathcal{C} = \{C_1, \ldots, C_k\}$. We can either introduce a distance threshold δ_{sum} to constrain the sum of the distances between C and the C_ℓ, $1 \leq \ell \leq k$,

$$\sum_{\ell=1}^{k} D(C, C_\ell) \leq \delta_{sum},$$

or a distance threshold δ_{pw} to constrain the pairwise distances between C and each C_ℓ, $1 \leq \ell \leq k$, i.e.

$$D(C, C_\ell) \leq \delta_{pw} \text{ for all } 1 \leq \ell \leq k.$$

In the following, we refer to the former as *sum distance constraint* and to the latter as *pairwise distance constraint*.

With these prerequisites it is now straight-forward to introduce the notion of approximate conservation into the concept of common intervals. Depending on the way distances are constrained, we end up with two different

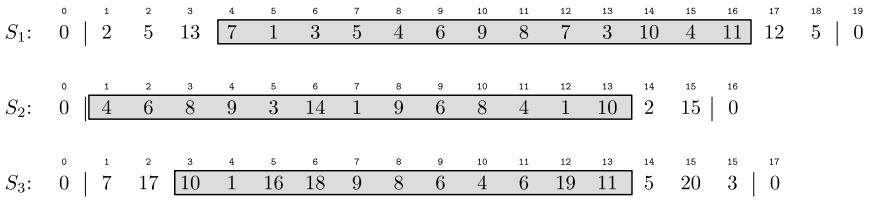


Figure 2.3: Approximate common intervals for sum distance threshold $\delta_{sum} \geq 8$, respectively pairwise distance threshold $\delta_{pw} \geq 3$.

definitions of *approximate common intervals*. Using the sum distance constraint, we get:

Definition 2 *Given a set of $k \geq 2$ strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ and a distance threshold δ_{sum}, we call a k-tuple of maximal intervals $([i_1, j_1]_{S_1}, \ldots, [i_k, j_k]_{S_k})$* ***sum distance constrained approximate common intervals*** *in $\mathcal{S}$ if and only if there is a $C \subseteq \Sigma$ with*

$$\sum_{\ell=1}^{k} D\big(C, \mathcal{CS}(S_\ell[i_\ell, j_\ell])\big) \leq \delta_{sum}.$$

The analogous definition for the pairwise distance constraint reads as follows:

Definition 3 *Given a set of $k \geq 2$ strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ and a distance threshold δ_{pw}, we call a k-tuple of maximal intervals $([i_1, j_1]_{S_1}, \ldots, [i_k, j_k]_{S_k})$* ***pairwise distance constrained approximate common intervals*** *in $\mathcal{S}$ if and only if there is a $C \subseteq \Sigma$ with*

$$D\big(C, \mathcal{CS}(S_\ell[i_\ell, j_\ell])\big) \leq \delta_{pw} \text{ for all } 1 \leq \ell \leq k.$$

Sets of the form of C in the last two definitions are called *close* to the respective k-tuple.

Example 3 *The interval combination $\mathcal{I} = ([4, 16]_{S_1}, [1, 13]_{S_2}, [3, 13]_{S_3})$ in Figure 2.3 forms a k-tuple of approximate common intervals in $\mathcal{S}$ for any sum distance threshold $\delta_{sum} \geq 8$. A close set is $C = \{1, 3, 4, 6, 8, 9, 10, 11\}$. For $\delta_{pw} \geq 3$, $\mathcal{I}$ is also a k-tuple of pairwise distance constrained approximate common intervals. But as $D(C, \mathcal{CS}(S_3[3, 13])) = 4$ holds, C is only a close set for $\delta_{pw} \geq 4$ under pairwise distance constraint.*

We should note at this point that these basic definitions of approximate common intervals are rather permissive: Choosing the distance threshold high enough, any combination of maximal intervals is possible no matter if they have any characters in common with each other or with their close set(s). Even empty close sets are possible. To avoid the latter two effects, we

Σ:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\mathcal{CS}(S_1[4,16])$:	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
$\mathcal{CS}(S_2[1,13])$:	1	0	1	1	0	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0
$\mathcal{CS}(S_3[3,13])$:	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0	1	0	1	1	0
Median:	1	0	1	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Center 1:	1	0	0	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0
Center 2:	1	0	1	1	0	1	0	1	1	1	1	0	0	0	0	1	0	0	0	0
Center 3:	1	0	1	1	0	1	0	1	1	1	1	0	0	0	0	0	0	1	0	0
Center 4:	1	0	1	1	0	1	0	1	1	1	1	0	0	0	0	0	0	0	1	0

Figure 2.4: Median and centers of the interval combination of Example 3.

can simply require that there is a close set C for which $\mathcal{CS}(S_\ell[i_\ell, j_\ell]) \cap C \neq \emptyset$ holds for all $1 \leq \ell \leq k$. We call such C *intersecting close set*. To prevent the formation of approximate common intervals with disjoint character sets, we can require that the involved intervals have pairwise intersecting character sets, i.e. $\mathcal{CS}(S_\ell[i_\ell, j_\ell]) \cap \mathcal{CS}(S_{\ell'}[i_{\ell'}, j_{\ell'}]) \neq \emptyset$ holds for all $1 \leq \ell, \ell' \leq k$. We call such approximate common intervals *pairwise set intersecting*.

Remember our goal to model approximate gene clusters in a common intervals style. Approximate common intervals clearly correspond to a grouping of approximate occurrences of a gene cluster. However, what is still missing in this model is the notion of a consensus gene cluster in the concept of approximate common intervals. Close sets are a first step in this direction, but what we actually need is a true optimality criterion, i.e. the definition of a "closest" set that minimizes its distance to the elements of a k-tuple of approximate common intervals. Technically, this boils down to defining a consensus set of a set of sets. We can define such a set C for any combination of sets $\mathcal{C} = \{C_1, \ldots, C_k\}$, $k \geq 2$, over an alphabet Σ either by minimizing the overall distance between C and the $C_1, \ldots, C_k$, i.e.

$$\sum_{\ell=1}^{k} D(C, C_\ell) \leq \sum_{\ell=1}^{k} D(C', C_\ell) \text{ for all } C' \subseteq \Sigma,$$

or by minimizing the maximum pairwise distance:

$$\max_{1 \leq \ell \leq k} \{D(C, C_\ell)\} \leq \max_{1 \leq \ell \leq k} \{D(C', C_\ell)\} \text{ for all } C' \subseteq \Sigma.$$

In the former case, we name such a C *median representative* of $\mathcal{C}$ or *median* for short. In the latter case, we name C *center representative* of $\mathcal{C}$ or *center* for short. Note that a median of sets is not necessarily unique. This is due to the fact that for even k a character occurring in the median can occur in exactly half of the k sets. When removing this character from the median, the total distance to the sets stays unchanged and the remaining characters form an alternative median. The center, as well, can be multiply defined, even for odd k. This is illustrated in the following example:

Example 4 *The median in Figure 2.4 is unique and has a total distance of* 8 *to the three character sets. There are four different centers which have each a pairwise distance of* 3 *to all three character sets.*

In context of approximate common intervals, both types of consensus sets are special close sets, namely the "closest" for the respective distance constraining mode. However, they are not necessarily intersecting close sets. If they are, we call them *intersecting median*, respectively *intersecting center*.

Before we begin with the definition of gene cluster models, we introduce some further terminology to refer to approximate occurrences of gene clusters. We distinguish additional subtypes of maximal intervals: Given a character set C, we say a maximal interval $[i, j]$ in S with $\mathcal{CS}(S[i, j]) \cap C \neq \emptyset$ is *closed* with respect to C, or C-closed for short, if and only if $S[i-1] \notin C$ and $S[j+1] \notin C$. This means that extending $[i, j]$ in either direction, a new character is added, that is neither an element of C nor does it occur already in $\mathcal{CS}(S[i, j])$. For the next subtype of maximal intervals, we define the *left-most essential position* i^* of $[i, j]$ with respect to C as the smallest index i', $i \leq i' \leq j$, such that $S[i'] \in C$. Analogously, we define the *right-most essential position* j^* of $[i, j]$ with respect to C as the largest index j', $i \leq j' \leq j$, such that $S[j'] \in C$. Simply put, these are the outermost positions in $[i, j]$, that contain an occurrence of C if such exist. Interval $[i^*, j^*]$ is called the *C-essential subinterval* of $[i, j]$. The characters at positions i^* and j^* are called *left-most essential character*, respectively *right-most essential character*, with respect to C. These concepts are used to define the second subtype of maximal intervals which comprises all intervals $[i, j]$ with $\mathcal{CS}(S[i, j]) \cap C \neq \emptyset$ for which $\mathcal{CS}(S[i, j]) = \mathcal{CS}(S[i^*, j^*])$ holds. Intervals of the form of $[i, j]$ are called *compact with respect to* C, or *C-compact* for short. The idea behind this definition is that such $[i, j]$ are minimal in the sense that there is no proper subinterval of $[i, j]$ that is maximal and comprises its C-essential subinterval. Or, stated in simpler terms, a compact interval covers only "necessary" positions, i.e. occurrences of elements of C, the characters between such occurrences and for the sake of interval maximality at its boundaries additional occurrences of these in-between characters. If $[i, j]$ is both closed and compact with respect to C, we call it *optimal* with respect to C, or *C-optimal* for short.

Example 5 *Interval* $[2, 13]$ *in* S_1 *in Figure 2.3 is maximal but neither closed nor compact with respect to* $C = \{1, 3, 4, 6, 8, 9, 10, 11\}$. *It is not closed because a neighboring character* $S_1[14] = 10$ *occurs in* C. *The* C*-essential subinterval of* $[2, 13]$ *is* $[5, 13]$. *As both intervals differ in their character set,* $[2, 13]$ *can not be compact. Interval* $[4, 16]$ *is an example of a* C*-optimal interval in* S_1.

The idea behind these definitions is to find in a genomic area that contains genes from a cluster the optimal placement of approximate cluster occur-

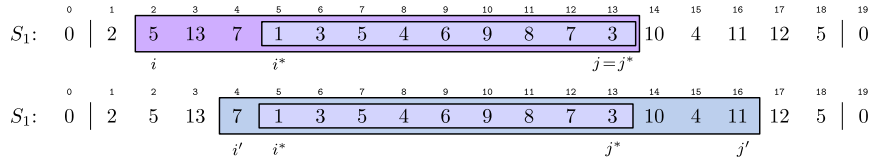


Figure 2.5: Construction of a corresponding C-optimal δ-location via the C-essential subinterval for $C = \{1, 3, 4, 6, 8, 9, 10, 11\}$.

rences. In other words, we want to introduce an optimality criterion for approximate locations of character sets. Unlike with perfect locations, interval maximality is not sufficient for this purpose, as approximate locations can be highly overlapping:

Example 6 *The list of maximal* 3*-locations of* $C = \{1, 3, 4, 6, 8, 9, 10, 11\}$ *in* S_1 *reads as follows:* $[\mathbf{5}, \mathbf{11}]$, $[8, 15]$, $[4, 15]$, $[10, 16]$, $[\mathbf{8}, \mathbf{16}]$, $[8, 17]$, $[6, 16]$, $[\mathbf{4}, \mathbf{16}]$, $[4, 18]$, $[2, 16]$. *The intervals printed in bold are* C*-optimal.*

However, we observe that only a small fraction of approximate locations is optimal with respect to C, so that the redundancy problem can be reduced by focusing on C-optimal intervals. In doing so, no approximate location is completely lost, as we can assign each approximate location its corresponding C-optimal approximate location that covers its C-essential subinterval and has the same or possibly a lower distance to C:

Observation 1 *Consider an interval* $[i, j]$ *in a string* S *and a character set* C. *If* $[i, j]$ *is a* δ*-location of* C *for a* $\delta \geq 0$*, then there is also a* δ*-location that is* C*-optimal and subsumes the* C*-essential subinterval of* $[i, j]$.

Proof: We prove this by giving a construction algorithm: First, we identify the C-essential subinterval $[i^*, j^*]$ which exists because $\mathcal{CS}(S[i, j]) \cap C \neq \emptyset$ in each δ-location $[i, j]$ of C. Then, we extend $[i^*, j^*]$ on both ends to $[i', j']$ until $S[i' - 1] \notin C \cup \mathcal{CS}(S[i^*, j^*])$ and $S[j' + 1] \notin C \cup \mathcal{CS}(S[i^*, j^*])$. This happens at the latest when the extended string boundaries are reached, i.e. $i' = 1$ respectively $j' = |S|$. By construction, such a $[i', j']$ is C-optimal and encloses $[i^*, j^*]$. It is also a δ-location of C: By definition, $D\big(\mathcal{CS}(S[i^*, j^*]), C\big) \leq D\big(\mathcal{CS}(S[i, j]), C\big)$, and in the following interval extension only two types of characters are added: those already contained in $\mathcal{CS}(S[i^*, j^*])$ which are neutral with respect to the distance to C, and elements from $C \setminus \mathcal{CS}(S[i^*, j^*])$ which reduce the distance. Therefore, also the distance constraint is met. □

Example 7 *Figure 2.5 visualizes the construction of the* C*-optimal interval* $[4, 16]_{S_1}$ *from the interval* $[2, 13]_{S_1}$ *via its* C*-essential subinterval* $[5, 13]_{S_1}$. *The distance between* C *and* $\mathcal{CS}(S_1[2, 13])$ *equals* 5 *and decreases to* 2 *when* $[2, 13]_{S_1}$ *is replaced by* $[4, 16]_{S_1}$.

At this point, we have introduced all foundations necessary to define the gene cluster discovery problems studied in this thesis.

We conclude this section with a final remark on the terminology used in this thesis. For the sake of readability, we use some concepts defined only for character sets also for intervals and refer by that implicitly to the corresponding character set. Based on the unique relation between an interval and its character set, we do not introduce any ambiguities in doing so. For example, we refer to all kinds of (approximate) locations of an interval, meaning the respective (approximate) locations of the character set of the interval. Also, we refer by "the distance to an interval" never to the physical distance on the string but to the set distance to the respective character set. We inherit this abbreviating notation also to combinations of intervals. In particular the terms "median" and "center" are used for approximate common intervals.

2.2 Perfectly conserved gene clusters

We begin our problem definitions with the most rigorous model, the perfectly conserved gene clusters. This concept requires the character set defining a gene cluster to have a perfect location in each of the studied genomes. Using the concept of common intervals, this can be formalized as follows:

Problem 1 *Given a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of $k \geq 2$ strings, find all $C \subseteq \Sigma$ that are character sets of common intervals in $\mathcal{S}$.*

We call such character sets *perfectly conserved gene clusters* in $\mathcal{S}$, or *perfect gene clusters* for short. For further analyses, not only the character set of a gene cluster but also its occurrences, namely the maximal locations in the studied genomes, will be of interest. Hence, we augment our problem statement in the following way:

Problem 2 *Given a character set $C \subseteq \Sigma$ that is a perfect gene cluster in $\mathcal{S} = \{S_1, \ldots, S_k\}$, list all maximal locations of C in $\mathcal{S}$.*

As every maximal location of a perfect gene cluster C can be combined with others to a k-tuple of common intervals, we can solve both Problems by computing all common intervals in $\mathcal{S}$. How this computation can be done efficiently and how the perfect gene clusters and their perfect occurrences can be extracted therefrom is the topic of Chapter 3.

2.3 Reference based approximate gene clusters

Our first approach to model approximately conserved gene clusters is based on the concept of reference intervals. This means that we do not consider all subsets of the alphabet of gene family ids for being approximate gene clusters,

but only those having a perfect occurrence in at least one input genome and approximate occurrences in the others. We formalize the respective gene cluster discovery problem using the concepts of approximate common intervals and close sets. Depending on the distance constraining mode, two different problem sets can be defined.

2.3.1 Sum distance constrained reference gene clusters

Under the sum distance constraint, the approximate gene cluster discovery problem based on reference occurrences reads as follows:

Problem 3 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s and a distance threshold δ_{sum}, find all $C \subseteq \Sigma$ with $|C| \geq s$ that have a perfect location in $\mathcal{S}$ and are intersecting close sets of sum distance constrained approximate common intervals in $\mathcal{S}$.*

We refer to character sets of the form of C as *reference-based approximate gene clusters*, or as *reference gene clusters* for short. If the distance constraining mode is not clear from the context, we add the term *sum distance constrained*.

As with perfect gene clusters, the (approximate) occurrences of a reference gene cluster C should be reported for further evaluation:

Problem 4 *Given a set $C \subseteq \Sigma$ that is a sum distance constrained reference gene cluster in $\mathcal{S} = \{S_1, \ldots, S_k\}$ for distance threshold δ_{sum}, list all C-optimal intervals in $\mathcal{S}$ that occur in sum distance constrained approximate common intervals with intersecting close set C.*

The restriction to C-optimal intervals reduces output redundancy. With C being an intersecting close set, it follows from Observation 1 that the skipped intervals are represented by corresponding C-optimal intervals.

2.3.2 Pairwise distance constrained reference gene clusters

The reference gene cluster discovery problem can be formulated analogously using the pairwise distance constraint:

Problem 5 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s and a distance threshold δ_{pw}, find all $C \subseteq \Sigma$ with $|C| \geq s$ that have a perfect location in $\mathcal{S}$ and are intersecting close sets of pairwise distance constrained approximate common intervals in $\mathcal{S}$.*

We call reference gene clusters of the form of C *pairwise distance constrained*. If the distance constraining mode is clear from the context, or if we refer to reference gene clusters in general we omit this specification.

Here, we need to define again the set of approximate locations that should be reported alongside the approximately conserved gene cluster. Based on

Observation 1, we are once more only interested in C-optimal approximate locations:

Problem 6 *Given a set $C \subseteq \Sigma$ that is a pairwise distance constrained reference gene cluster in $\mathcal{S} = \{S_1, \ldots, S_k\}$ for distance threshold δ_{pw}, list all C-optimal intervals in $\mathcal{S}$ that occur in pairwise distance constrained approximate common intervals with intersecting close set C.*

By definition, we can solve the four problems stated in this section by computing a restricted type of approximate common intervals that have an intersecting close set with a perfect location in the studied genomes. How this computation can be done efficiently and how the reference gene clusters and their approximate occurrences can be extracted will be discussed in Chapter 5.

2.4 Median based approximate gene clusters

For the general approximate gene cluster discovery problem, we drop the constraint requiring a consensus gene cluster to have a reference occurrence in the studied genomes. In doing so, we gain the opportunity to employ the optimality criterion introduced in the previous section to get only gene clusters that best represent the gene content of their (approximate) occurrences. Based on the sum distance constraint this can be formalized as follows:

Problem 7 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s and a distance threshold δ_{sum}, find all $C \subseteq \Sigma$ with $|C| \geq s$ that are a median of sum distance constrained approximate common intervals in $\mathcal{S}$ with pairwise intersecting sets.*

We refer to character sets of the form of C as *median-based approximate gene clusters*, or *median gene clusters* for short.

As with the previous gene cluster models, we want to report also the approximate cluster occurrences. Since a median gene cluster is always defined for a specific combination of approximate common intervals, we report the respective k-tuples:

Problem 8 *Given a set $C \subseteq \Sigma$ that is a median gene cluster in $\mathcal{S} = \{S_1, \ldots, S_k\}$ for distance threshold δ_{sum}, list all pairwise set intersecting approximate common intervals $([i_1, j_1], \ldots, [i_k, j_k])$ for which C is a median.*

If we limit the search to intersecting medians, we can again make use of Observation 1 and report only C-optimal pairwise set intersecting approximate common intervals. This is feasible as optimizing interval boundaries with respect to the median of their character sets has no impact on the median itself. In Chapter 6, we show how median gene clusters can be computed based on approximate common intervals. Ways of dealing with redundancy in the output of Problem 8 will be discussed in Chapter 8.

2.5 Center based approximate gene clusters

The general approximate gene cluster discovery problem can as well be defined for the pairwise distance constraint:

Problem 9 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s and a distance threshold δ_{pw}, find all $C \subseteq \Sigma$ with $|C| \geq s$ that are a center of pairwise distance constrained approximate common intervals in $\mathcal{S}$ with pairwise intersecting sets.*

We refer to character sets of the form of C as *center-based approximate gene clusters*, or *center gene clusters* for short.

Concerning the output of approximate cluster occurrences, the same applies as for median gene clusters. A center gene cluster is defined for a specific interval combination:

Problem 10 *Given a set $C \subseteq \Sigma$ that is a center gene cluster in $\mathcal{S} = \{S_1, \ldots, S_k\}$ for distance threshold δ_{sum}, list all pairwise set intersecting approximate common intervals $([i_1, j_1]_{S_1}, \ldots, [i_k, j_k]_{S_k})$ for which C is a center.*

Limiting the search to intersecting centers is not a sufficient condition for applying Observation 1 to reduce the output to C-optimal interval combinations. This is because optimizing interval boundaries with respect to a set C can change the distribution of the pairwise distances such that C is no longer center of the modified intervals. To ensure that we report for each center gene cluster C at least one combination of approximate occurrences, we can modify Problem 9 such that there has to be a k-tuple of C-optimal approximate common intervals. In Chapter 7, we show how center gene clusters can be computed based on approximate common intervals. The issue of output redundancy will be addressed in Chapter 8.

2.6 Modeling missing gene cluster occurrences

So far, we introduced in this chapter models with varying capabilities of coping with incomplete gene cluster conservation. What all these models have in common is their immanent constraint that a gene cluster has to occur in each of the input genomes, albeit these occurrences can be up to a certain extent fragmentary under the approximate gene cluster models.

But what if a gene cluster is completely missing in one or a couple of input genomes? From a biological point of view, there is no reason why such a setting should be excluded from our models. From a practical point of view, this constraint makes the outcome of the gene cluster prediction overly dependent on the selection of input genomes. Of course, the problem of missing gene cluster occurrences can be evaded if gene clusters are also predicted in all $\Theta(2^k)$ subsets of the k input genomes. However, it appears to be more

elegant to integrate such a setting directly into the gene cluster models. For that purpose, the notion of a *quorum parameter* was introduced in several publications, e.g. [63, 78, 15, 60]. It refers to a user-defined threshold that determines the minimum number of genomes in which a gene cluster has to occur to be detected under a given gene cluster model. In the following, we call a gene cluster *q-covering* if it has an (approximate) occurrence in q out of k genomes, $2 \leq q \leq k$. The q-covering perfect gene cluster discovery problem reads as follows:

Problem 11 *Given a set $\mathcal{S} = \{S_1, \ldots, S_k\}$ of $k \geq 2$ strings and a quorum parameter q, $2 \leq q \leq k$, find all $C \subseteq \Sigma$ that are a character set of common intervals in a set $\mathcal{S}' \subseteq \mathcal{S}$, with $|\mathcal{S}'| \geq q$.*

When it comes to approximate gene clusters, we need to distinguish between models based on the sum distance constraint and models based on the pairwise distance constraint. In the latter case, the modification of the respective problem formulations is straight forward. For the reference gene cluster discovery problem we get:

Problem 12 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s, a distance threshold δ_{pw} and a quorum parameter q, $2 \leq q \leq k$, find all $C \subseteq \Sigma$ with $|C| \geq s$ that are an intersecting close set of pairwise distance constrained approximate common intervals in a subset $\mathcal{S}' \subseteq \mathcal{S}$, $|\mathcal{S}'| \geq q$ and have a perfect location in a $\mathcal{S}'$.*

The integration of the quorum parameter into the center gene cluster discovery problem reads as follows:

Problem 13 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s, a distance threshold δ_{pw} and a quorum parameter q, $2 \leq q \leq k$, find all $C \subseteq \Sigma$ with $|C| \geq s$ that are center of pairwise distance constrained approximate common intervals in a set $\mathcal{S}' \subseteq \mathcal{S}$, $|\mathcal{S}'| \geq q$ with pairwise intersecting sets.*

For the two sum distance constrained approximate gene cluster models, the introduction of a quorum parameter is slightly more involved. While in the original definitions, the sum distance threshold is given for a fixed number of approximate locations, it needs to be defined for a range of possible tuple sizes in the q-covering case. There are different ways of dealing with this problem. One extreme would be to simply ignore it, and use a fixed threshold for the whole range $2 \leq q \leq k$. In other words, this means that we would set the "cost" for a missing gene cluster occurrence to zero. The other extreme would be to charge the "complete cost" of a missing gene cluster C, i.e. $|C|$, the number of genes that got lost. Both approaches have undesirable effects. In the first, there is a tendency to use the quorum unnecessarily

in the sense that approximate common intervals are reported that cover less genomes than they could based on the sum distance threshold. In the second approach, there is the problem of balancing the distance constraint and the quorum parameter. If the distance constraint is too strict, no gene clusters are detected that cover only a subset of the genomes. If the distance constraint is too loose, we report gene cluster occurrences that have only single genes in common. A third approach is to subtract for each missing gene cluster occurrence its average share of the sum distance, i.e. $\frac{\delta_{sum}}{k}$. We focus in this thesis on this intermediate approach. Therefore, we define the q-covering sum distance constrained reference gene cluster discovery problem as follows:

Problem 14 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s, a distance threshold δ_{sum} and a quorum parameter q, find all $C \subseteq \Sigma$ with $|C| \geq s$ that are intersecting close set of sum distance constrained approximate common intervals in a subset $\mathcal{S}' \subseteq \mathcal{S}$, $|\mathcal{S}'| \geq q$, for distance threshold $\frac{|\mathcal{S}'|}{k} \cdot \delta_{sum}$ and have a perfect location in $\mathcal{S}'$.*

The integration of the quorum parameter into the median gene cluster discovery problem follows the same pattern:

Problem 15 *Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$, $k \geq 2$, a minimum cluster size s, a distance threshold δ_{sum} and a quorum parameter q, find all $C \subseteq \Sigma$ with $|C| \geq s$ that are median of sum distance constrained approximate common intervals in a set $\mathcal{S}' \subseteq \mathcal{S}$, $|\mathcal{S}'| \geq q$, with pairwise intersecting sets for distance threshold $\frac{|\mathcal{S}'|}{k} \cdot \delta_{sum}$.*

Chapter 3

Perfect gene clusters

As we have seen in Section 2.2, the search for perfect gene clusters and their occurrences in a set of genomes can be solved by the computation of common intervals. This problem has been studied extensively in the literature [85, 32, 31, 20, 78, 21, 5] and efficient algorithms for its solution have been proposed. Especially to mention in this context are the *Connecting Intervals* Algorithm (CI) by Schmidt and Stoye [78] and the algorithm by Didier et al. [20, 78, 21]. Both algorithms compute all common intervals in two strings of length at most n in $O(n^2)$ time, which is worst-case time optimal. For the CI Algorithm, also an extension to $k > 2$ strings is sketched which runs in $O(kn^2)$ time [78]. In this chapter, we revise this algorithm for the following reasons: Firstly, the algorithms for computing approximate gene clusters presented in the following chapters adopt its basic search strategy. Secondly, we show in Section 3.4 how its space complexity can be reduced from $O(kn^2)$ to $O(kn)$. For simplicity, we study the algorithm first for pairwise string comparison. The extension to multiple strings is shown in Section 3.6.

3.1 The basic CI Algorithm

To compute all common intervals of two strings S_1 and S_2 of length at most n, it is clearly sufficient to test each of the $O(n^4)$ combinations of maximal intervals of S_1 with maximal intervals of S_2 for having the same character set. This observation can be translated into a search strategy that considers explicitly only the maximal intervals of a selected *reference string*, in the following S_1, and finds all maximal locations of their character sets in the other string S_2.

The CI algorithm implements this idea by combining a systematic traversal of all maximal intervals $[i, j]$ of S_1, named *reference intervals* in the following, with a tracking and marking of maximal intervals in S_2 that consist only of characters occurring in $\mathcal{CS}(S_1[i, j])$. Common intervals can then be

Algorithm 1 Connecting Intervals (CI)

```
1: for i = 1, ..., |S_1| do
2:   j ← i
3:   while j ≤ |S_1| and [i, j] is left-maximal do
4:     c ← S_1[j]
5:     while [i, j] is not right-maximal do
6:       j ← j + 1
7:     end while
8:     for each position p in S_2 with S_2[p] = c do
9:       mark position p in S_2
10:      find maximal interval [ℓ, r] of marked positions that contains p
11:      if |CS(S_1[i, j])| = |CS(S_2[ℓ, r])| then
12:        output ([i, j], [ℓ, r])
13:      end if
14:    end for
15:    j ← j + 1
16:  end while
17:  unmark all positions in S_2
18: end for
```

retrieved by comparing the character sets of $[i, j]$ and the intervals tracked in S_2. Pseudocode for this procedure is given in Algorithm 1. The algorithm traverses intervals in S_1 with a common start position i one after the other for increasing end positions j (**while** loop, line 3) and tests for maximality (lines 3 and 5). Once left-maximality is lost for an interval with start position i, it can not be regained by shifting the end position j to the right. Hence, the iteration through the outer **while** loop is stopped for a start position i as soon as the first non-left-maximal interval occurs. Right-maximality is guaranteed implicitly by extending the processed intervals instantly until they become right-maximal (lines 5 to 7). This happens at the latest for $j = |S_1|$.

Processed in this way, reference intervals with a common start position i differ in their character sets exactly by one character from the preceding reference interval. With all reference intervals being maximal, it is easy to see that the character c in which two consecutive reference intervals differ corresponds to the right neighbor of the preceding reference interval and equals $S_1[j]$ before the inner **while** loop (line 5) is executed for the current reference interval. This observation becomes important for the next step of the algorithm, the tracking of maximal intervals consisting only of characters from the current $\mathcal{CS}(S_1[i, j])$. Instead of detecting these intervals from scratch, one can build on the intervals detected and marked beforehand for

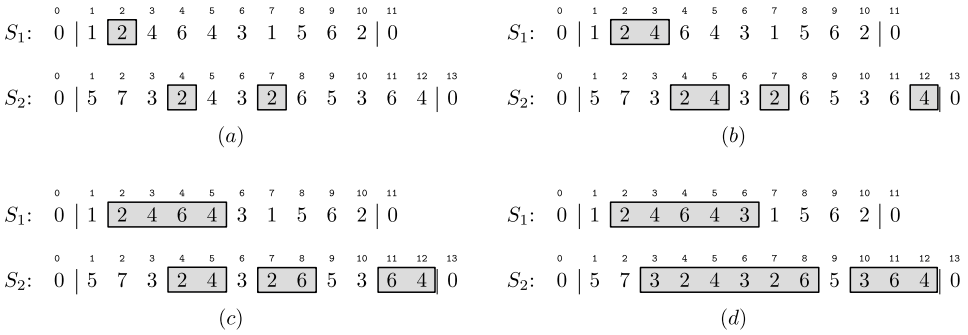


Figure 3.1: Iterative generation of reference intervals in S_1 for start position 2 and corresponding interval marking in S_2 for the strings of Example 8. (a) $[2,2]_{S_1}$, marking of two new intervals in S_2; (b) $[2,3]_{S_1}$, one interval extension, one new interval marking; (c) $[2,4]_{S_1}$ not right-maximal; $[2,5]_{S_1}$, two interval extensions; (d) $[2,6]_{S_1}$ one interval merging, one interval extension; no further extension $[2,j]_{S_1}$, $j > 6$, is left-maximal, next start position processed.

the previous reference interval. To update the interval marking in S_2, it is sufficient to add all occurrences of c in S_2 (line 9). This involves for each occurrence either an extension of a marked interval, the merging of two marked intervals or the generation of a new marked interval if both neighboring positions are not in $\mathcal{CS}(S_1[i,j])$. This process is illustrated in the following example.

Example 8 *Given two strings S_1, S_2 over alphabet $\Sigma = \{1,2,3,4,5,6,7\}$ with S_1 = 1 2 4 6 4 3 1 5 6 2 and S_2 = 5 7 3 2 4 3 2 6 5 3 6 4, the interval marking in S_2 for reference intervals in S_1 with start position 2 is visualized in Figure 3.1.*

Finally, the algorithm identifies the maximal marked intervals in S_2, i.e. intervals $[\ell, r]$ with unmarked neighboring positions $\ell - 1$ and $r + 1$, (line 10) and tests whether any of them have the same character set as $S_1[i,j]$. Here, it is sufficient to consider only those maximal intervals of S_2 that were extended by the latest new character of the current $S_1[i,j]$. All other intervals do not contain this character and therefore have a different character set. Since, by construction, the character sets of the marked intervals are subsets of $\mathcal{CS}(S_1[i,j])$, the test for character set equality can be replaced by a comparison of character set sizes (line 11). Intervals having the same character set are combined with $[i,j]$ to a pair of common intervals and reported (line 12). Once all reference intervals with a fixed left border i have been processed, i is shifted to the next position (outer **for** loop, line 1) and the iterative generation of reference intervals starts anew. For that purpose, all positions in S_2 are unmarked (line 17). The algorithm terminates when i reaches the end of S_1.

From the previous considerations, it is clear that we do not miss any common intervals in Algorithm 1 and that only interval combinations that are indeed common intervals are reported. Thus, we claim the correctness of the following theorem:

Theorem 1 *Let $\mathcal{S} = \{S_1, S_2\}$ be a set of two strings over a finite alphabet Σ. Algorithm 1 computes all common intervals in $\mathcal{S}$.*

3.2 Efficient data structures for the CI Algorithm

In the description of the basic CI Algorithm, we omitted details on how to efficiently implement several non-trivial operations: Testing maximality of intervals, enumerating character occurrences, tracking boundaries of maximal marked intervals and computing character set sizes of substrings. In [78], Schmidt and Stoye introduced several data structures to replace these costly operations by constant-time look-ups.

For testing maximality (lines 3 and 5), they use a dynamic bit array of size $|\Sigma|$ named Occ to keep track of all characters in the current reference interval. An interval $[i, j]$ is maximal if the entries in Occ corresponding to the neighboring characters $S_2[i-1]$ and $S_2[j+1]$ are zero. Maintenance of Occ is the following: Each time the left border of the reference interval is shifted, the entries of Occ are set to zero. When the interval is extended to the right, the value of the new character c, Occ$[c]$, is set to one. This array is linked with a counter |Occ| to track the number of ones in the current Occ. This value corresponds to $|\mathcal{CS}(S_1[i, j])|$ which is used in line 11 of the algorithm.

For the enumeration of all occurrences of a character c in S_2, a static array of length $|\Sigma|$ named Pos is precomputed that lists for each character $c \in \Sigma$ all positions p where c occurs in S_2. The look-up of the next position of the for-loop in line 8 can then be done in constant time.

Efficient maintenance of the maximal marked intervals in S_2 (lines 9, 10) is achieved by means of another dynamic array of size $|S_2|$, that stores for each maximal marked interval at its start and end positions, ℓ and r, the complementary boundary positions r, respectively ℓ. (This array is well-defined, as each position in S_2 can be element of at most one maximal marked interval at a time.) Using this array, we can test in constant time whether the neighbors of a newly marked position are themselves marked, and in case of an interval merging, we can identify the boundaries of the new interval in constant time.

The last crucial step in the algorithm is the determination of the value $|\mathcal{CS}(S_2[\ell, r])|$. For this task, Stoye and Schmidt suggest a pre-computed $|S_2| \times |S_2|$ table called Num, in which the entry at position $[\ell, r]$ corresponds to the number of different characters that occur in the substring $S_2[\ell, r]$, i.e. Num$[\ell, r] = |\mathcal{CS}(S_2[\ell, r])|$. Hence, the look-up of $|\mathcal{CS}(S_2[\ell, r])|$ in line 11 of

Pos[1] = ∅
Pos[2] = {4, 7}
Pos[3] = {3, 6, 10}
Pos[4] = {5, 12}
Pos[5] = {1, 9}
Pos[6] = {8, 11}
Pos[7] = {2}

Num:

$\ell \backslash r$	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2	3	4	5	5	5	6	6	6	6	6
2		1	2	3	4	4	4	5	6	6	6	6
3			1	2	3	3	3	4	5	5	5	5
4				1	2	3	3	4	5	5	5	5
5					1	2	3	4	5	5	5	5
6						1	2	3	4	4	4	5
7							1	2	3	4	4	5
8								1	2	3	4	5
9									1	2	3	4
10										1	2	3
11											1	2
12												1

Figure 3.2: Data structures Pos and Num for S_2 from Example 8. Pos[c] lists the occurrences of character c in S_2. Entry Num[ℓ, r] contains the value of $|\mathcal{CS}(S_2[\ell, r])|$.

Algorithm 1 is possible in constant time. An example of the two static data structures Pos and Num can be found in Figure 3.2.

3.3 Complexity analysis

In [78], Schmidt and Stoye show that the CI algorithm for pairwise string comparison runs in $O(n^2)$ time requiring $O(n^2)$ space. We reproduce this analysis in this section, as we build upon it later on in the complexity analysis of our algorithms for approximate gene cluster detection:

The outer **for** loop in Algorithm 1 is executed $|S_1|$ times. Shifting of j in the two **while** loops takes place at most $|S_1|$ times for each fixed i. Together with the constant time test for left- and right maximality, the generation of reference intervals is in $O(n^2)$. From the description in Section 3.2 and from $|\Sigma| \in \Theta(n)$, it follows that the maintenance of data structure Occ is also in $O(n^2)$. The crucial part in the analysis of the CI Algorithm is the **for** loop in line 16. Due to the maximality constraint for reference intervals, each character is at most $|S_1|$ times the latest new character c in $[i, j]$. Hence, each of the $|S_2|$ positions p in S_2 becomes at most $|S_1|$ times newly marked. Thus, there are at most $|S_1| \cdot |S_2|$ changes of marked intervals and the same number of character set comparisons. Using the data structures from Section 3.2, both operations are in $O(1)$. The computation of the static data structures Pos and Num takes place only once and can be done in time $O(n)$, respectively $O(n^2)$. The dynamic data structures are reset $O(n)$ times. This results in a total asymptotic runtime of $O(n^2)$.

Concerning space complexity, table Num is the most costly data structure. All other data structures require only linear space, such that the overall space requirement amounts to $O(n^2)$.

3.4 Space efficient CI Algorithm

In Section 3.2, we have seen how expensive operations of the CI Algorithm can be replaced by constant time look-ups. However, this advantage is gained at the expense of additional space requirements. In particular, the storage of table NUM can be prohibiting when studying long strings or several strings at a time, as it increases space complexity from $O(n)$ to $O(n^2)$. In the following, we show that the information of table NUM that is actually used in the algorithm can be stored in two dynamic arrays of length $|\Sigma|$ respectively $|S_2|$ that are updated after each shift of the left border i of the reference interval in S_1.

3.4.1 Data structures replacing NUM

The first array for replacing NUM is a ranking scheme similar to the one used in [21], named RANK in the following. In this data structure, we store for the characters of Σ their ranks based on the order of their first occurrence in a string S, respectively '∞' for characters not occurring in S. If it is not clear from the context which string a ranking is based on, or if we deal with different rankings at a time, we extend the notation RANK to RANK_S to indicate that RANK is based on S.

Example 9 *Character ranking based on substring $S_1[2, 10]$ of Example 8:*

c:	0	1	2	3	4	5	6	7
$\text{RANK}_{S_1[2,10]}[c]$:	∞	5	1	4	2	6	3	∞

For our space efficient version of the CI Algorithm, RANK is used as a dynamic data structure updated such that RANK equals $\text{RANK}_{S_1[i,|S_1|]}$ while reference intervals with left-most position i are processed.

Before we define the second data structure, we introduce an auxiliary construct named RANGE which is an array of size $|S_2|$. Given a ranking scheme RANK, we store in RANGE for each position p in S_2, $1 \leq p \leq |S_2|$, its *range interval*, which is the largest interval around p that consists only of characters with a rank of at most $\text{RANK}[S_2[p]]$, i.e. $\text{RANGE}[p] = [a, b]$ for $a \leq p \leq b$ if and only if:

(i) $\text{RANK}[S_2[q]] \leq \text{RANK}[S_2[p]]$ for all $a \leq q \leq b$, and

(ii) $\text{RANK}[S_2[a-1]] > \text{RANK}[S_2[p]]$ and

(iii) $\text{RANK}[S_2[b+1]] > \text{RANK}[S_2[p]]$.

This definition of range intervals is similar to the concept of rank intervals used in [21]. The connection between the rank and range intervals is visualized in Figure 3.3. We do not use them explicitly in our approach, but

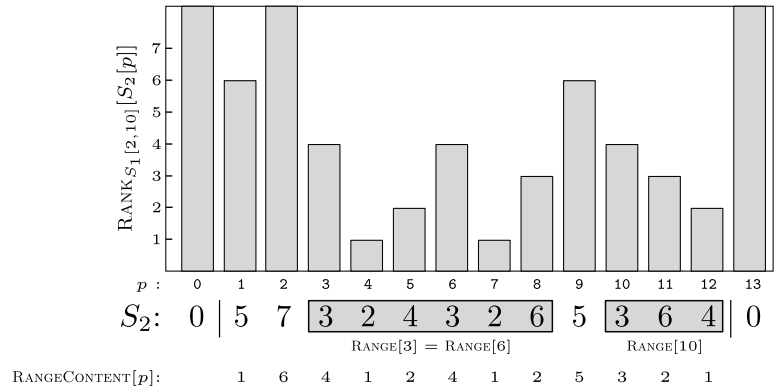


Figure 3.3: Connection between the character ranking based on $S_1[2, 10]$ and range intervals and range content in S_2.

the second data structure for replacing NUM, named RANGECONTENT, is based upon them. We define it as an array of size $|S_2|$ that contains for each position p in S_2 the *range content* of its range interval, namely the number of different characters in the range interval of p for the current character ranking, i.e. RANGECONTENT$[p] = |\mathcal{CS}(S_2[a, b])|$ for RANGE$[p] = [a, b]$. If in the course of the algorithm RANK and RANGECONTENT are updated in regard to shifts of i, we claim that the following observation holds:

Observation 2 *The CI Algorithm uses only values of table* NUM *that are also stored in array* RANGECONTENT.

To understand the correctness of this observation, let us have a closer look at the entries in NUM that are actually used during a run of the CI Algorithm. Table NUM is only used in line 13 of Algorithm 1 to look-up the size of character sets $\mathcal{CS}(S_2[\ell, r])$, where $[\ell, r]$ is a maximal marked interval in S_2. By construction, $[\ell, r]$ has the following properties: *(i)* It contains only characters from the reference interval $[i, j]$. *(ii)* Among the elements of $\mathcal{CS}(S_2[\ell, r])$, $S_2[p]$ has the right-most first occurrence in $S_1[i, j]$. *(iii)* The neighboring characters $S_2[\ell - 1]$ and $S_2[r + 1]$ do not occur in $\mathcal{CS}(S_1[i, j])$.

In terms of the character ranking based on the order of first occurrence in $S_1[i, |S_1|]$, this means that *(i)* the ranks of all characters occurring in $[\ell, r]$ fall into the range from 1 to $|\mathcal{CS}(S_1[i, j])|$, *(ii)* the character at position p has the highest rank within this interval, and *(iii)* the ranks of the neighboring characters are strictly higher than RANK$[S_2[p]]$. Hence, the interval $[\ell, r]$ equals the range interval of position p for the current ranking. From this, it follows immediately that the values of NUM$[\ell, r]$ and RANGECONTENT$[p]$ are equal. It remains to be shown that the arrays RANK and RANGECONTENT can be computed and updated efficiently.

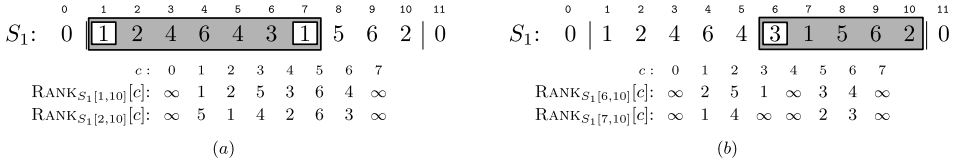


Figure 3.4: Update of rank after shift of left border of reference intervals. Only characters with occurrence in the grey shaded area change their rank.

3.4.2 Maintenance of RANK

Array RANK for $i = 1$ can be computed in linear time: After initializing all entries of RANK with ∞, we iterate through S_1 and, for each character read, we test if its rank is already set. If not, we assign it the next unused rank and increment a counter to keep track of the ranks already used.

To keep array RANK up to date when shifting i to $i' = i + 1$, we can simply compute it anew for $S_1[i', |S_1|]$ which is of course possible in time $O(n)$. Alternatively, we can perform a selective update that changes only the necessary positions. Although the latter approach yields no improvement in terms of worst-case time complexity, we study it in the following to get a deeper insight into the underlying process to simplify the understanding of updates of range intervals described later on.

For simplicity, we denote in the following RANK_{old} for $\text{RANK}_{S_1[i,|S_1|]}$ and RANK_{new} for $\text{RANK}_{S_1[i',|S_1|]}$. Clearly, the rank of a $c \in \Sigma$ with no occurrence in $S_1[i, |S_1|]$ remains ∞. Moreover, if character $c_{old} = S_1[i]$ has another occurrence right of i, the ranks of characters having their first occurrence in $S_1[i, |S_1|]$ right of this occurrence of c_{old} do not change. Hence, only the ranks of c_{old} and of characters c having their left-most occurrence between $i + 1$ and the next occurrence of c_{old}, respectively between $i + 1$ and $|S_1|$ remain to be updated.

Example 10 *In Figure 3.4, both types of rank updates are visualized:* (a) *position* $i = 1$ *drops out of reference interval. The next occurrence of* $S_1[1] = 1$ *is at position* 7*. The rank changes only for characters in interval* $[1, 7]$*.* (b) *position* $i = 6$ *drops out of reference interval. There is no further occurrence of* $S_1[6] = 3$ *in* S_1*. The rank changes for characters in interval* $[6, 10]$*.*

It is easy to see that for characters of the second type, the rank decreases by one, as exactly one character, namely c_{old}, does no longer occur before them. Moreover, for two such characters the following observation holds:

Observation 3 *For any pair of characters* $c \neq c_{old}$ *and* $c' \neq c_{old}$ *it holds that if* $\text{RANK}_{old}[c] > \text{RANK}_{old}[c']$*, then also* $\text{RANK}_{new}[c] > \text{RANK}_{new}[c']$*.*

This is true by construction: If the rank of any of these two characters c and c' needs to be updated, it is either the one with lower rank or both. And the

rank can only decrease. For c_{old}, on the contrary, the rank either increases by the number of different characters between its occurrences on position i and its next occurrence, or it becomes ∞ if no further occurrence exists.

3.4.3 Maintenance of RangeContent

Next, we study the maintenance of RangeContent for S_2. It is straight forward to see that the initialization of RangeContent for a given character ranking Rank can be done in quadratic time. We simply scan the neighborhood left and right of each position p in S_2 to identify its range interval $[a, b]$. Then we count the number of different characters in $[a, b]$ and set $\text{RangeContent}[p] = |\mathcal{CS}(S_2[a, b])|$. It remains to be shown, how we can update RangeContent efficiently when the start position of the reference interval shifts from i to $i' = i + 1$. To keep the total runtime of the algorithm in $O(n^2)$, we need to accomplish this in linear time, as the update is performed $O(n)$ times in total.

Let us first study the impact of Rank updates on range intervals in S_2: For positions that contain c_{old}, the range interval is likely to change completely due to a possibly large change of $\text{Rank}[c_{old}]$ when shifting i to $i' = i + 1$ in S_1. For other positions, however, possible changes are very limited: We have already seen that for two characters c and $c' \neq c_{old}$ with $\text{Rank}_{old}[c] > \text{Rank}_{old}[c']$ also $\text{Rank}_{new}[c] > \text{Rank}_{new}[c']$. Hence, it can neither happen that an occurrence of c becomes part of the range interval of an occurrence of c' only by decreasing the rank of c below the rank of c', nor can an occurrence of c' become detached from the the range interval of an occurrence of c only by increasing its rank above the rank of c. This means that positions p in S_2 that do not contain c_{old} have no impact on changes of the range intervals of each other. Therefore, it is sufficient to consider only occurrences of c_{old} for updating range intervals of such p. Moreover, occurrences of c_{old} can change range intervals only in one way: As their rank always increases in the update, they can shorten other range intervals but not extend them.

Example 11 *In Figure 3.5, the impact of the rank update from* $\text{Rank}_{S_1[2,10]}$ *to* $\text{Rank}_{S_1[3,10]}$ *on range intervals and range content in* S_2 *is visualized. There are two occurrences of* $c_{old} = 2$ *at positions* 4 *and* 7. *In the new ranking, they disrupt the former range intervals* $\text{Range}[3] = \text{Range}[6] = [3, 8]$. *Range interval* $\text{Range}[10] = [10, 12]$ *is not affected by the update.*

With these findings, we can now formulate our update strategy for array RangeContent. It consists of two phases: First, we recompute the value of RangeContent at positions containing c_{old}. Then, we update the remaining positions. For updating RangeContent at positions p with $S_2[p] = c_{old}$, we identify all range intervals $[a, b]$ in S_2 whose elements

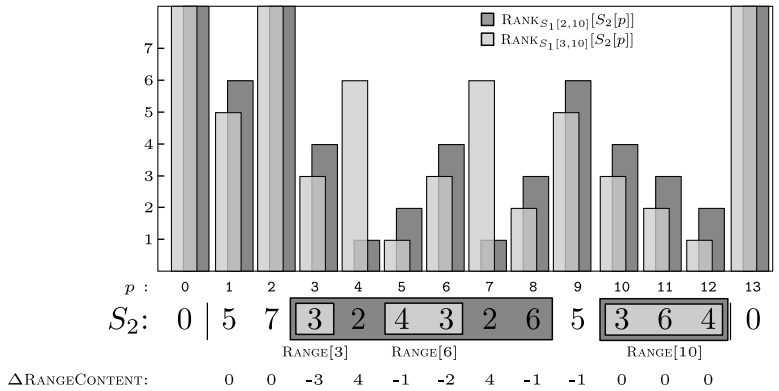


Figure 3.5: Rank update from $\text{RANK}_{S_1[2,10]}$ to $\text{RANK}_{S_1[3,10]}$ and impact on range intervals and range contents in S_2. (For the sake of clarity, range intervals are only drawn for occurrences of 3.)

have rank at most $\text{RANK}_{new}[c_{old}]$ and are confined by characters with ranks higher than $\text{RANK}_{new}[c_{old}]$. Then we compute for each such $[a, b]$ the size of its character content and set for each occurrence p of c_{old} in this interval $\text{RANGECONTENT}[p] = |\mathcal{CS}(S_2[a, b])|$. Since range intervals of two positions p and p' with $\text{RANK}[p] = \text{RANK}[p']$ are either congruent or do not intersect, we can perform this part of the update in linear time by a simple iteration through S_2 that is summarized in Algorithm 2. During the iteration (**for**-loop line 2), we track occurrences of c_{old} in the current interval (line 4). Once, the right neighbor of the current interval $[a, b]$ has a higher rank than c_{old} (line 6), we have reached the end of the next range interval spanning characters with rank at most $\text{RANK}_{new}[c_{old}]$. Hence, we set the range content of all occurrences of c_{old} in the current interval to $|\mathcal{CS}(S_2[a, b])|$ (line 8). We get this value for free if we track the number of different characters in the current interval during the iteration, for example using the combination of OCC and |OCC|, as introduced in Section 3.2. For finding the next range interval of this type, we continue the search to the immediate right of the finished interval (line 10). We iterate only once through S_2 in Algorithm 2 and set in the inner **for** loop the range content of every position p at most once. With all other operations in the algorithm being in $O(1)$, the total runtime is in $O(n)$.

For updating RANGECONTENT at positions p with $S_2[p] \neq c_{old}$, recall that the corresponding range interval can only be changed by occurrences of c_{old}. In other words: After the update, these range intervals have a common neighboring character, namely c_{old}. This observation is the key to understanding how RANGECONTENT can be updated at positions p with $S_2[p] \neq c_{old}$ in linear time. All we need to do is to scan S_2 for range intervals $[a, b]$ which are at least on one side delimited by an occurrence of c_{old}, i.e.

Algorithm 2 Update RANGECONTENT for occurrences of c_{old}

```
1: a ← 1, occSet ← ∅
2: for b = a, ..., |S_2| do
3:   if RANK[S_2[b]] = RANK[c_old] then
4:     occSet ← occSet ∪ {b}
5:   end if
6:   if RANK[S_2[b + 1]] > RANK[c_old] then
7:     for each p in occSet do
8:       RANGECONTENT[p] ← CS(S_2[a, b])
9:     end for
10:    a ← b + 1
11:    occSet ← ∅
12:  end if
13: end for
```

either $S_2[a-1] = c_{old}$ or $S_2[b+1] = c_{old}$, detect in each of them all positions for which it constitutes the range interval and update the range content of these positions accordingly.

In the first step, sketched in Algorithm 3, we focus on range intervals that are delimited on the left side by an occurrence of c_{old}. To update the range content in such intervals, we identify all occurrences of c_{old} and use the right neighbors of these occurrences as potential start positions a for range intervals in S_2. Beginning from each start position, we extend the corresponding interval step by step to the right as long as only characters with rank smaller than RANK$[c_{old}]$ are added (**while** loop, lines 4 to 16) and test after each interval extension whether the resulting interval $[a, b]$ constitutes a range interval. This can be accomplished simply by checking whether the right neighbor of the interval has a rank higher than the maximum rank of all characters in the current interval (line 8). If so, the range interval of each occurrence p of the character with maximum rank in the current interval is $[a, b]$ and we set RANGECONTENT$[p] = |\mathcal{CS}(S_2[a, b])|$ (lines 9 to 11). Again, we can use for example the combination of OCC and |OCC|, as introduced in Section 3.2, to track the character set size of the interval during the iteration.

It remains to be seen how we can track the maximum rank and the occurrences p of characters with maximum rank. Initially, $b = a$ holds. Therefore, we set the maximum rank in $[a, b]$ to the rank of $S_2[a]$ (line 3). This value is updated once a character with higher rank is about to be added. For technical reasons, we do so once the character is considered as right neighbor of the previous interval (line 12). This is feasible, as it is done only when all rank tests for the previous interval are completed. For tracking the occurrences of the character with maximum rank in the current interval, we simply check whether the newly added position contains

Algorithm 3 Update RANGECONTENT at remaining positions (Part1)

```
1: for each position a with S_2[a−1] = c_old do
2:    b ← a,   occSet ← ∅
3:    maxRank ← RANK[S_2[a]]
4:    while RANK[S_2[b]] < RANK[c_old] do
5:       if RANK[S_2[b]] = maxRank then
6:          occSet ← occSet ∪ {b}
7:       end if
8:       if RANK[S_2[b + 1]] > maxRank then
9:          for each p in occSet do
10:            RANGECONTENT[p] ← |CS(S_2[a, b])|
11:         end for
12:         maxRank ← RANK[S_2[b + 1]]
13:         occSet ← ∅
14:      end if
15:      b ← b + 1
16:   end while
17: end for
```

a character with the current maximum rank (line 5) and, if so, add it to a set with the other occurrences (line 6). Since, at this point, the maximum rank for the new interval is already updated, this works also for intervals with a new maximum rank. Occurrences are removed from this set once their range interval was found and the value of RANGECONTENT was set. This always happens before a new maximum rank occurs in the interval so that this set contains at any time only occurrences of the current rank maximum.

Clearly, we identify in this procedure all range intervals $[a, b]$ for which $S_2[a-1] = c_{old}$ holds, and update the range content of all positions for which an interval of the form $[a, b]$ is their range interval. An analogous procedure can be defined for range intervals that are delimited on the right side by occurrences of c_{old}. It is sketched in Algorithm 4.

In both algorithms, the intervals of S_2 starting right, respectively left, of occurrences of c_{old} are processed such that the extension stops before the start/end point of the next interval is reached. Also, we note that the range content of each position in S_2 is set at most twice: once in Algorithm 3 and once in Algorithm 4. Therefore, the complete update process is possible in $O(n)$. Altogether, we conclude the following:

Theorem 2 *Using the data structures described above, our version of the CI Algorithm computes all common intervals of two strings of length at most n in $O(n^2)$ time and $O(n)$ space.*

Algorithm 4 Update RANGECONTENT at remaining positions (Part 2)

```
1: for each position b with S_2[b+1] = c_old do
2:   a ← b,  occSet ← ∅
3:   maxRank ← RANK[S_2[b]]
4:   while RANK[S_2[a]] < RANK[c_old] do
5:     if RANK[S_2[a]] = maxRank then
6:       occSet ← occSet ∪ {a}
7:     end if
8:     if RANK[S_2[a−1]] > maxRank then
9:       for each p in occSet do
10:        RANGECONTENT[p] ← |CS(S_2[a, b])|
11:      end for
12:      maxRank ← RANK[S_2[a−1]]
13:      occSet ← ∅
14:    end if
15:    a ← a − 1
16:  end while
17: end for
```

3.5 Output format

The CI Algorithm is designed to report all combinations of common intervals of the given input strings. To solve Problems 1 and 2 formally, we need to derive therefrom a non-redundant representation of perfect gene clusters and its maximal locations on the input strings.

We show that this step becomes obsolete if we use a dense representation of common intervals. For a start, we observe that the current representation of common intervals has the potential to produce an enormous overhead: Imagine two strings $S_1 = (abx_1)^{n/3}$ and $S_2 = (abx_2)^{n/3}$ with $x_1 \neq x_2$. There are $O(n^2)$ combinations that form common intervals of at least two characters, but there is only a single underlying character set. For $n = 12$, such a setting is visualized in Figure 3.6. The situation becomes worse when considering $k > 2$ strings, as the number of possible common interval combinations increases exponentially with the number of studied strings. To avoid this redundancy, it seems reasonable to represent all common intervals of a character set C by a k-tuple of sets, in which the set at position ℓ, $1 \leq \ell \leq k$, contains all locations of C in S_ℓ. This reduces the output size drastically as the following example shows:

Example 12 *Common interval combinations of* $C = \{a, b\}$ *in* $\mathcal{S} = \{S_1, S_2\}$*:*
$([1,2],[1,2])$, $([1,2],[4,5])$, $([1,2],[7,8])$, $([1,2],[10,11])$, $([4,5],[1,2])$, $([4,5],[4,5])$, $([4,5],[7,8])$, $([4,5],[10,11])$, $([7,8],[1,2])$, $([7,8],[4,5])$,

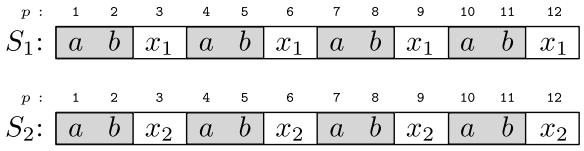


Figure 3.6: Example strings with redundant common interval combinations.

$([7,8],[7,8])$, $([7,8],[10,11])$, $([10,11],[10,11])$, $([10,11],[4,5])$, $([10,11],[7,8])$, $([10,11],[10,11])$.
Compact common intervals representation for C:
$S_1: [1,2],[4,5],[7,8],[10,11]$, $S_2: [1,2],[4,5],[7,8],[10,11]$.

The dense representation corresponds to the solution set of the perfect gene cluster problems stated in Chapter 2. Thus, it is reasonable to adapt Algorithm 1 such that it computes directly the compact representation without generating the k-tuples as an intermediate step.

In [78], Schmidt and Stoye solve this problem for two strings, S_1 and S_2, by running Algorithm 1 twice: First, they compute maximal locations with identical character content within S_1 by comparing S_1 against itself and flag all intervals having the same character content except for the left-most occurrence. Then, they compare S_1 and S_2 skipping reference intervals flagged in the first step. This approach requires additional space in the form of two $|S_1| \times |S_1|$ tables, each storing sets of intervals.

We show briefly that the compact common interval representation can be generated while the space complexity remains in $O(n)$. For that purpose, we adapt the algorithm such that we do not only mark intervals in S_2 but also in the reference string S_1. If, for a reference interval $[i,j]$, we find no perfect location of $\mathcal{CS}(S_1[i,j])$ in the substring $S_1[1,i-1]$ but one or more locations in S_2, we output $\mathcal{CS}(S_1[i,j])$ together with all perfect locations in $S_1[i,|S_1|]$ and S_2. In case we find a perfect location of $\mathcal{CS}(S_1[i,j])$ in $S_1[1,i-1]$, we output nothing for $[i,j]$, as common intervals for $\mathcal{CS}(S_1[i,j])$ were already reported for its left-most location in S_1. This approach outputs all common intervals in the compact representation. We do not use any of the additional $O(n^2)$ tables described in [78]. Instead, we need the data structures for marking intervals not only for S_2 but also for S_1. These are in $O(n)$ using the space efficient approach introduced in Section 3.4.

3.6 Extension to multiple Genomes

The extension of Algorithm 1 to $k > 2$ strings is straight-forward. Since Problem 1 requires a perfect location in each of the studied strings, it is still sufficient to consider explicitly only maximal intervals of the reference string S_1. To avoid output redundancy, we adapt the trick of the previous

Algorithm 5 Outline of parallel CI computation in multiple strings

```
1: for each maximal interval [i, j] in S_1 do
2:   if CS(S_1[i, j]) has no location in S_1[1, |S_1|] then
3:     locCount ← 1
4:     for each S_ℓ = S_2, ... S_k do
5:       if CS(S_1[i, j]) has a location in S_ℓ then
6:         locCount ← locCount + 1
7:       end if
8:     end for
9:     if locCount = k then
10:      output CS(S_1[i, j])
11:      for each S_ℓ = S_1, ... S_k do
12:        output all maximal locations [a, b] of CS(S_1[i, j]) in S_ℓ
13:      end for
14:    end if
15:  end if
16: end for
```

section and consider only reference intervals that are the left-most maximal location of their character set in the reference string. For these, we need to check whether they have a perfect location in each of the remaining strings $S_2, \ldots, S_k$. We can do this in two different ways: Iteratively or in parallel. In the iterative approach, we do a sequence of pairwise genome comparisons between the reference string and the other strings and in doing so track those reference intervals of S_1 which had a perfect location in each of the strings already processed. Alternatively, we can test candidate intervals in parallel for all strings $S_2, \ldots, S_k$. The advantage of the second approach is that the decision whether a reference interval yields a combination of common intervals can be made right after it was tested such that the tracking of intervals that are still candidates for common intervals can be omitted. However, this comes at the cost of additional space consumption as the data structures for tracking maximal marked intervals have to be stored for $k-1$ strings at a time. However, using the space efficient approach, space requirements increase only to $O(kn)$, while, in the iterative approach, up to $\Theta(n^2)$ pairs of interval boundaries need to be stored to track candidate reference intervals for common intervals. Concerning the asymptotic time complexity, both approaches are equivalent: Using the space efficient common interval representation, both run in time $O(kn^2)$. An outline of the parallel search strategy for the condensed output representation is given in Algorithm 5.

3.7 Quorum parameter

The introduction of a quorum parameter into common interval computation to represent q-covering perfect gene clusters is straight-forward and was already briefly discussed in [78]. To detect character sets that have a location in at least q out of k input strings, we need to adapt the search strategy such that $k - q + 1$ strings are considered as reference strings. (Otherwise, we would miss all character sets that have locations only in strings from which no reference intervals are taken.) During the search, we simply count the number of strings in which the character set of a reference interval has at least one location and output in the end only those for which we get a total of at least q. To avoid redundancy in the common interval representation, the character set of every reference interval should be checked for previous locations not only in the current reference string but also in the previous reference strings. If such an occurrence exists, the character set was already processed and the reference interval can be skipped.

The runtime of the algorithm increases to $O\big(k(k-q+1)n^2\big)$ when common intervals are computed with a quorum parameter q. This is simply because reference intervals are now taken from $k - q + 1$ strings instead of one, whereas the effort for processing the intervals of a single reference string stays basically unchanged. The space requirements remain in $O(kn)$.

Chapter 4

Computation of optimal δ-locations

To keep the following chapters on approximate gene cluster detection concise, we address at first an important subproblem which is the computation of optimal δ-locations. We will see later on which role this operation plays in approximate gene cluster detection. For the time being, it is only important to note that this operation is performed not for single sets but for the character sets of all maximal intervals of a string. Therefore, we study in this chapter the following problem:

Problem 16 *Given two strings S_1 and S_2 over an alphabet Σ and a distance threshold δ, find for each maximal interval in S_1 all optimal δ-locations of its character set in S_2.*

A similar problem setting, the computation of all δ-locations of a character set (not only optimal ones) in a set of k strings, was studied by Amir et al. [2]. They claim to have found an $O(kn^3 + \textit{output size})$ time and $O(kn^3)$ space algorithm to solve this problem. However, it is possible to construct a counter example for which the graph-based approach presented in [2] does not detect all δ-locations. (See the Appendix for details.)

We show in the following that the pairwise CI Algorithm can be extended to solve Problem 16 in time $O(n^2(\delta+1)^2)$ using $O(n^2)$ space. The structure of this chapter is as follows: We begin with a basic extension of the pairwise CI Algorithm in Sections 4.1 to 4.3. Then, we show in Section 4.4 how the worst-case time complexity of this algorithm can be reduced.

4.1 Adaptation of the CI Algorithm

We present the changes necessary to the Connecting Intervals Algorithm along the pseudocode of the extended algorithm given in Algorithm 6. Since we need to compute optimal δ-locations for all maximal intervals in S_1, we

Algorithm 6 Computation of optimal δ-locations

```
1: for i = 1, ..., |S_1| do
2:   j ← i , C ← ∅
3:   while j ≤ |S_1| and [i, j] is left-maximal do
4:     c ← S_1[j]
5:     while [i, j] is not right-maximal do
6:       j ← j + 1
7:     end while
8:     remove all intervals [a, b] from C with c ∈ CS(S_2[a, b])
9:     for each interval [a, b] ∈ C do
10:      remove [a, b] from C unless it is optimal δ-location of CS(S_1[i, j])
11:    end for
12:    for each position p in S_2 with S_2[p] = c do
13:      mark position p in S_2
14:      find unmarked positions l_1, ..., l_{δ+1} and r_1, ..., r_{δ+1} around p
15:      for each interval [l_x+1, r_y−1] with 1 ≤ x, y ≤ δ+1 do
16:        add [l_x+1, r_y−1] to C if it is optimal δ-location of CS(S_1[i, j])
17:      end for
18:    end for
19:    j ← j+1
20:  end while
21:  unmark all positions in S_2
22: end for
```

can adopt the iterative generation of reference intervals from the original CI Algorithm. Also the marking of intervals in S_2 that consist only of characters from the current reference interval $[i, j]$ in S_1 is useful for our purpose: Since approximate locations need to have character sets that intersect with $C = \mathcal{CS}(S_1[i, j])$, these intervals are good starting points for detecting C-optimal δ-locations. However, unlike with perfect locations, it is clearly not sufficient to consider only recently extended maximal marked intervals. Additionally, we need to consider intervals that are partially unmarked and/or contain no occurrence of c, the character most recently added to the current reference interval.

Example 13 *Let S_1 and S_2 be two strings over $\Sigma = \{1, \ldots, 15\}$ with:*

$S_1 = 6\ 2\ 1\ 4\ 3\ 9\ 4\ 5\ 3\ 8\ 7\ 12\ 2\ 10\ 6\ 3$ *and*
$S_2 = 11\ 14\ 7\ 12\ 7\ 1\ 5\ 3\ 8\ 13\ 4\ 7\ 1\ 12\ 15\ 3\ 4\ 9\ 1\ 15\ 8\ 1\ 5.$

The interval marking in S_2 for $\mathcal{CS}(S_1[3, 11]) = \{1, 3, 4, 5, 7, 8, 9\}$ is shown in Figure 4.1. Recently extended marked intervals are $[3, 3], [5, 9]$ and $[11, 13]$. Examples of optimal δ-locations for $\delta \geq 3$ are $[3, 14]$ and $[16, 23]$.

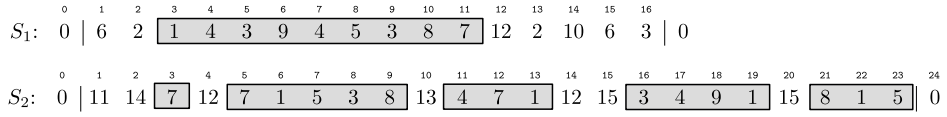


Figure 4.1: Example of interval marking in S_2 for reference interval $[3, 11]_{S_1}$. To find (optimal) δ-locations of $\mathcal{CS}(S_1[3, 11])$, also intervals that are partially unmarked need to considered.

In our search strategy, we distinguish two types of optimal δ-locations of C, namely intervals with no occurrence of c and intervals that contain such an occurrence. To detect intervals of the first type, we make use of the following observation:

Observation 4 *Every interval $[a, b]$ in a string S that is an optimal δ-location of a $C \subseteq \Sigma$ is also an optimal δ-location of $C' = C \setminus \{c\}$ for every $c \notin \mathcal{CS}(S[a, b])$.*

Proof: With $[a, b]$ being C-optimal, its neighboring characters $S[a-1]$ and $S[b+1]$ are not in C and therefore not in C'. Moreover, lacking an occurrence of c, it has the same left- and right-most essential position with respect to both C' and C. Therefore, $[a, b]$ is C'-optimal. Finally, the distance of $\mathcal{CS}(S[a, b])$ to C' equals its distance to C minus 1. Therefore, also the distance constraint is met which makes $[a, b]$ an optimal δ-location of C'. □

Customized to our problem, this means all optimal δ-locations of a reference interval $[i, j]$ with no occurrence of c can be deduced from the set of optimal δ-locations of the previous reference interval $[i, j']$ with $\mathcal{CS}(S_1[i, j']) = \mathcal{CS}(S_1[i, j]) \setminus \{c\}$. Therefore, in Algorithm 6, we store the optimal δ-locations in a set $\mathcal{C}$ which we pass on to the next reference interval unless a shift of the left border of the reference intervals occurred. In this case, the reference interval generation starts anew with a single character, and we empty $\mathcal{C}$ as there are no approximate locations to inherit (line 2). In all other cases, we filter $\mathcal{C}$ for intervals that contain no occurrence of c and test the remaining intervals directly for being optimal δ-locations of the new C (line 10). After that, $\mathcal{C}$ consists exactly of the optimal δ-locations of the new C with no occurrence of c.

Unfortunately, this strategy does not work for optimal δ-locations of the second type, intervals containing an occurrence of c. Some of them are shared among $[i, j']$ and $[i, j]$, and some are not, as the following example illustrates:

Example 14 *For the strings of Example 13, the character sets $\mathcal{CS}(S_1[3, 10])$ and $\mathcal{CS}(S_1[3, 11])$ have the following optimal δ-locations in S_2 for $\delta = 3$ with and without occurrences of $c = 7$:*

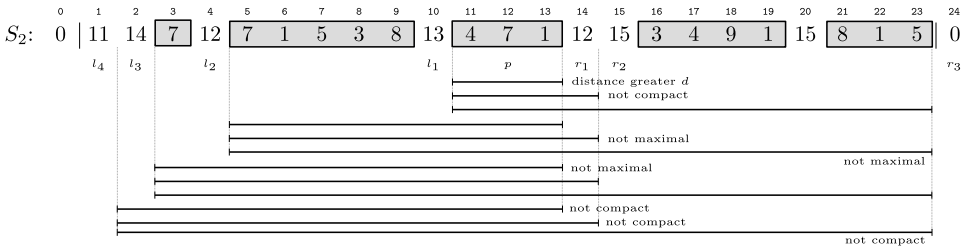


Figure 4.2: All intervals of the form $[l_x+1, r_y-1]$ for $p = 12$, $\delta = 3$. For intervals that are no optimal δ-location of $\mathcal{CS}(S_1[3, 11])$ the missing property is given.

	optimal δ-locations in S_2	
	containing 7	*not containing 7*
$\mathcal{CS}(S_1[3, 10])$:	$[5, 13]$	$[15, 23]$, $[16, 19]$, $[21, 23]$
$\mathcal{CS}(S_1[3, 11])$:	$[3, 9]$, $[3, 13]$, $[3, 23]$, $[5, 9]$, $[5, 13]$, $[11, 23]$	$[15, 23]$,$[16, 19]$

Therefore, we compute such intervals anew for each reference interval $[i, j]$ with $C = \mathcal{CS}(S_1[i, j])$. In order to do so, it is sufficient to identify for each occurrence of c in S_2 all C-optimal intervals around this position p that contain at most δ different unmarked characters. All other C-optimal intervals either contain no occurrence of c or have a distance to C greater than δ. To find candidates for optimal δ-locations of the second type, we compute positions to the left and right of p with increasing numbers $x, y \geq 1$ of unmarked characters (line 14):

$$l_x = l_x(p) = \max(\{l \mid S_2[l, p] \text{ contains } x \text{ different unmarked characters}\} \cup \{0\})$$
$$r_y = r_y(p) = \min(\{r \mid S_2[p, r] \text{ contains } y \text{ different unmarked characters}\} \cup \{|S_2|+1\}).$$

By definition, the substrings $S_2[l_x + 1, r_y - 1]$ contain at most $x + y - 2$ different characters not occurring in $S_1[i, j]$. (The number is smaller if the same unmarked characters occur left and right of p.) Clearly, not all of them are C-optimal or fulfill the distance constraint, as the example in Figure 4.2 illustrates. But together, they form a superset of the C-optimal δ-locations around p. This is true because any other interval $[a, b]$ that covers position p falls into one of the following two categories: Either it is contained in the superinterval $[l_{\delta+1} + 1, r_{\delta+1} - 1]$, i.e. $l_{\delta+1} < a \leq b < r_{\delta+1}$, but then it is not C-closed, or it is not contained within $[l_{\delta+1}, r_{\delta+1}]$, i.e. $a < l_{\delta+1}$ and/or $b > r_{\delta+1}$, and has therefore distance greater than δ to C. Thus, we only need to test every interval of the form $[l_x + 1, r_y - 1]$ for being an optimal δ-location for the reference interval $[i, j]$ and add it to $\mathcal{C}$ if it passes this test (line 16). Once all occurrences of c have been processed, $\mathcal{C}$ contains all optimal δ-locations of C. To avoid that intervals with multiple occurrences of c are redundantly inserted, we can simply add a rule by which

only intervals in which p is, for example, the left-most occurrence of c are added to $\mathcal{C}$. From the previous considerations, it follows directly that the presented algorithm solves Problem 16. Therefore, we claim the correctness of the following theorem:

Theorem 3 *Algorithm 6 computes for two strings S_1 and S_2 and a distance threshold δ for each maximal interval $[i, j]$ in S_1 the set of optimal δ-locations of $\mathcal{CS}(S_1[i, j])$ in S_2.*

4.2 Implementation details and data structures

So far, we focused on the conceptual extension of the CI Algorithm for detecting optimal δ-locations. Now, we have a closer look on how the different operations in Algorithm 6 can be implemented efficiently and which data structures can be employed.

For the algorithm part that is shared with the original CI algorithm, we simply adopt the data structures introduced in Section 3.2. These comprise array Occ of size $|\Sigma|$ for tracking the character content of the reference interval, array Pos of size $|\Sigma|$ for enumerating character occurrences and the array of size $|S_2|$ for tracking start/end positions of maximal marked intervals. We also use the $O(n^2)$ table Num to determine the character set sizes of intervals in S_2.

A novelty in the extended algorithm is the set of candidate intervals $\mathcal{C}$ for inheriting candidate δ-locations from previous reference intervals. The easiest way to implement this set is by means of a simple (unsorted) list. The first interesting step that involves $\mathcal{C}$ is the removal of elements that contain an occurrence of c, the new character in the current reference interval. This test can be performed by an iteration through $\mathcal{C}$ in which we check for each interval $[a, b]$ whether it contains an occurrence of c. Since these occurrences correspond to the entries in the list Pos$[c]$, we can accomplish this simply by iterating through this list for each $[a, b]$ and compare the interval boundaries with every p in Pos$[c]$. Once we find a combination for which p falls into $[a, b]$, we remove the interval from the list. The next step is to figure out which of the remaining elements actually are optimal δ-locations for the new reference interval. For that purpose, we make use of the following observation:

Observation 5 *Every interval $[a, b]$ in a string S that is an optimal δ-location of a $C' \subseteq \Sigma$ is an optimal δ-location of $C = C' \cup \{c\}$ for $c \notin C'$ and $c \notin \mathcal{CS}(S[a, b])$ if and only if $c \neq S[a-1]$, $c \neq S[b+1]$ and $D(C', \mathcal{CS}(S[a, b])) < \delta$.*

Proof: $\Rightarrow$: From $[a, b]$ being C-closed and from $c \in C$ follows directly that $S[a-1] \notin C$ and $S[b+1] \notin C$. Furthermore, we have $D(C, \mathcal{CS}(S[a, b])) \leq \delta$. Removing a single character from C that is not contained in $\mathcal{CS}(S[a, b])$ reduces the distance by one. Therefore, $D(C', \mathcal{CS}(S[a, b])) < \delta$ holds.

$\Leftarrow$: It follows from $c \neq S[a-1]$, $c \neq S[b+1]$ and $C = C' \cup \{c\}$ that $[a,b]$ is C-optimal. Furthermore, we have $D(C', \mathcal{CS}(S[a,b])) < \delta$. Adding a single character to C' increases this distance by at most one so that it can not exceed δ. Therefore, $[a,b]$ is an optimal δ-location of C. □

This means, we need to check every optimal δ-location of C' only for the following two properties: Is any of its neighboring positions an occurrence of c? Does its distance to C' equal δ? If at least one of these tests turns out positive, we remove the respective interval from $\mathcal{C}$. The first test can clearly be answered in constant time. Assuming that we store the distance to the current reference interval for every newly added interval in $\mathcal{C}$ and increment this value by one for each extension of the reference interval, also the second test is possible in constant time.

To identify optimal δ-locations that contain an occurrence of c, we compute positions $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$ around each occurrence p of c. This can be achieved with a simple scan of the neighborhood to the left and right of p until $\delta + 1$ different unmarked characters are found in each direction. Once these positions are found, we test for each combination $[l_x+1, r_y-1]$ with $1 \leq x, y \leq \delta+1$ whether it is an optimal δ-location of C. First, we test for interval maximality which is given if and only if $\textsc{Num}[l_x, r_y-1] \neq \textsc{Num}[l_x+1, r_y-1]$ and $\textsc{Num}[l_x+1, r_y] \neq \textsc{Num}[l_x+1, r_y-1]$ hold. Every maximal $[l_x+1, r_y-1]$ is automatically C-closed, because neither $S_2[l_x]$ nor $S_2[r_y]$ can be contained in C. Regarding C-compactness, remember from the previous chapter that this property is fulfilled for an interval $[a,b]$ if $\mathcal{CS}(S_2[a,b]) = \mathcal{CS}(S_2[a^*, b^*])$ with $a^*$and b^* being the left-most, respectively the right-most essential position in respect to C. This can be tested by comparing $\textsc{Num}[a,b]$ and $\textsc{Num}[a^*, b^*]$. It remains to be shown how we can identify efficiently $a^*$and b^* for intervals $[a,b]$ with $a = l_x+1$ and $b = r_y-1$. Clearly, a^* and b^* correspond to the left-most, respectively right-most, marked position in the interval $[a,b]$. It is also obvious that a^* only depends on a, while b^* only depends on b. Therefore, we can get this information almost for free, during the scan of the left and right neighborhood of position p (line 14 of Algorithm 6). We simply track the most recently passed marked position during the scans to the left and right of p and record it for each l_x, respectively r_y. In doing so, we have these values available when intervals of the form $[l_x+1, r_y-1]$ are tested for C-compactness. Intervals that pass this test are C-optimal, but not all of them are necessarily also δ-locations of C. To test the distance constraint, we compute for each of these intervals $[l_x+1, r_y-1]$ the distance $D(\mathcal{CS}(S_1[i,j]), \mathcal{CS}(S_2[l_x+1, r_y-1]))$. This is equal to the value of $|C| - |\mathcal{CS}(S_2[l_x+1, r_y-1])|$ plus twice the number of different unmarked characters in $S_2[l_x+1, r_y-1]$. The values of $|C|$ and $|\mathcal{CS}(S_2[l_x+1, r_y-1])|$ can be directly taken from $|\textsc{Occ}|$, respectively $\textsc{Num}[l_x+1, r_y-1]$. If we track the number of unmarked characters during the generation of candidate intervals $[l_x+1, r_y-1]$, we have this value available,

so that the complete distance computation can be performed in constant time. This tracking comes at no extra cost if we generate candidate intervals with a fixed left border l_x+1 and increase the right border r_y-1 step by step by incrementing y. In each step, at most one unmarked character is added to the interval, namely $S_2[r_{y-1}]$. If $\textsc{Num}[l_x+1, r_{y-1}-1] \neq \textsc{Num}[l_x+1, r_{y-1}]$ holds, a new unmarked character is added, otherwise it occurred already in the previous interval.

After this step, the test of intervals $[l_x+1, r_y-1]$ for being optimal δ-locations of C is completed. Therefore, all intervals that passed this final test are added to $\mathcal{C}$.

4.3 Complexity analysis

For simplicity, we start the analysis of Algorithm 6 with those algorithm parts that are directly adopted from the original CI Algorithm. These comprise the generation of reference intervals in S_1 and the tracking and marking of maximal intervals in S_2 that consist only of characters of the current reference interval. We have proven in Chapter 3 that this is in $O(n^2)$. Recall that we have also shown that every position p in S_2 is at most $O(n)$ times newly marked, such that in total the **for-**loop in line 12 is executed $O(n^2)$ times. The first interesting step within this **for-**loop is the detection of unmarked positions $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$. This involves a scan of the left and right neighborhood of p. In the worst case, we process the whole string adding the factor n to the complexity, while the test of a single character for being unmarked and new to the current interval is in $O(1)$. Thus, the total time spent on the detection of $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$ for all reference intervals is in $O(n^3)$. The second task in the **for-**loop is the collection of the candidate intervals of the form $[l_x+1, r_y-1]$. For each occurrence p, there are $O((\delta+1)^2)$ many of these intervals, while there are $|\textsc{Pos}[c]|$ many occurrences of each c in S_2. As every position p in S_2 is marked at most n times, $O\Big(n \cdot \sum_{c\in\Sigma} \big(|\textsc{Pos}[c]| \cdot (\delta+1)^2\big)\Big) = O(n^2(\delta+1)^2)$ intervals are added to $\mathcal{C}$ during the complete run of Algorithm 6. Using the results of the preceding scan, the generation of a single candidate takes constant time such that the total time spent on the generation of candidate intervals is also in $O(n^2(\delta+1)^2)$.

Before we continue with the analysis, we estimate how many optimal δ-locations a single reference interval $[i, j]$ can possibly have. In general, we observe that for any set of characters (no matter whether it is connected to a reference interval or not) this number is bounded in the following way:

Observation 6 *The number of optimal δ-locations of a character set $C \subseteq \Sigma$ in a string S of length n is in $O(n(\delta+1))$ for all $\delta \geq 0$.*

Proof: We count how many optimal δ-locations of C can have the same left-most position a in S. For that purpose, we show that every two such

intervals $[a, b_1]$ and $[a, b_2]$, $b_1 < b_2$, contain a different number of characters from $\Sigma \setminus C$. Clearly, we have $S[b_1+1] \notin C$ and $S[b_1+1] \notin \mathcal{CS}(S[a, b_1])$, while $S[b_1+1] \in \mathcal{CS}(S[a, b_2])$. However, it also holds that every $c \in \mathcal{CS}(S[a, b_1]) \setminus C$ is contained in $\mathcal{CS}(S[a, b_2])$. Therefore, the number of characters from $\Sigma \setminus C$ can not be the same in the two intervals. Moreover, a δ-location of C can have only between 0 and δ characters from $\Sigma \setminus C$. Thus, there are at most $\delta + 1$ optimal δ-locations of C with left-most position a and $O(n(\delta + 1))$ optimal δ-locations of C in the complete string. □

With this result, we can estimate the size of $\mathcal{C}$ which is important for the analysis of the two algorithm steps that involve a processing of $\mathcal{C}$. The first one is the removal of elements of $\mathcal{C}$ that contain an occurrence of c, the character most recently added to the reference interval (line 8). At this point, $|\mathcal{C}|$ is in $O(n(\delta+1))$, as $\mathcal{C}$ consists only of the optimal δ-locations of the previous reference interval. Therefore, the nested iteration through $\mathcal{C}$ and POS described earlier is in $O(n(\delta + 1) \cdot |\text{POS}[c]|)$ for one reference interval and in $O(n^3(\delta + 1))$ for the total algorithm. (The latter is true because $\sum_{c \in \Sigma} \text{POS}[c] = |S_2|$.) The second step that involves a processing of $\mathcal{C}$ is the **for**-loop in line 9 where the remaining intervals in $\mathcal{C}$ are tested for being optimal δ-locations. As we have seen in the previous section, this test can be performed in $O(1)$ time for each element, and therefore in $O(n(\delta+1))$ for the complete set. In total, there are $O(n^2)$ sets $\mathcal{C}$ to process in Algorithm 6, such that the complete time spent on this step is in $O(n^3(\delta+1))$. From the previous considerations it follows that this is also the time complexity of the complete algorithm. The space complexity of Algorithm 6 is in $\Theta(n^2)$ for the use of table NUM.

4.4 Faster computation of optimal δ-locations

For practical use, it would be desirable to decrease the asymptotic time complexity of our algorithm from cubic to quadratic dependence on n. With $\delta \ll n$, this is still true if this reduction comes at the cost of an additional factor δ in the complexity term. In the following, we show that such an improvement can be achieved with some modifications of the algorithm. Currently, there are two obstacles to the quadratic dependence on n in Algorithm 6: the scan of S_2 involved in the determination of unmarked positions $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$ surrounding each position p (line 14) and the two iterations through the interval set $\mathcal{C}$ to determine inheritable optimal δ-locations (lines 8 and 10).

4.4.1 Precomputation of $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$

Concerning the first problem, we observe that for each position p in S_2 the pattern of marked and unmarked positions in its neighborhood at the

time it gets newly marked depends only on the left border i of the current reference interval in S_1. Hence, the values $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$ can be precomputed and stored for all p, each time i is shifted in Algorithm 6. For the storage of these values, we use two tables, each of size $|S_2| \times (\delta + 1)$, which we name L and R in the following. There, the values $l_x(p)$ and $r_y(p)$ should be stored such that $L[p][x] = l_x(p)$ and $R[p][y] = r_y(p)$ hold for every position p in S_2 and all $1 \leq x, y \leq \delta + 1$ based on the current reference interval in S_1.

Example 15 *Table L for S_2, $\delta = 3$ and ranking* $\text{RANK}_{S_1[3,|S_1|]S_2}$ *for S_1, S_2 defined as in Example 13:*

p	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$L[1][p]$	0	0	2	2	4	5	5	7	4	0	10	10	12	10	0	15	16	15	18	0	20	21	21
$L[2][p]$	0	0	1	1	2	4	4	5	3	0	9	4	11	2	0	14	15	14	17	0	14	20	20
$L[3][p]$	0	0	0	0	1	3	3	4	2	0	8	2	10	1	0	12	14	12	16	0	12	18	14
$L[4][p]$	0	0	0	0	0	2	2	3	1	0	7	1	9	0	0	10	12	10	15	0	10	17	12

This precomputation alone does not improve the time complexity of the algorithm. However, we show that after the initial computation of L and R for $i = 1$, their update after each shift $i \to i+1$ is in time $O(n(\delta + 1))$ for both tables. To this end, we need a ranking scheme similar to the one used in Chapter 3 for computing the range content of intervals. The difference is that we rank the characters of Σ not only based on their first occurrence in $S_1[i, |S_1|]$ but based on their first occurrence in the concatenated string $S_1[i, |S_1|]S_2$. In doing so, we can distinguish the ranks of characters in S_2 with no occurrence in $S_1[i, |S_1|]$. In the following descriptions, we use the term rank not only for characters but also for positions in S_2. We refer by that to the rank of the character that occurs at this position. Since this character is uniquely defined, we do not introduce any ambiguities by that.

For the initialization of tables L and R, we scan the left, respectively right, neighborhood of each position p for the first $\delta + 1$ different characters with a rank greater than $\text{RANK}_{S_1S_2}[S_2[p]]$ and set the entries in L, respectively R, to the corresponding positions. It is easy to see that positions with rank higher than c correspond to unmarked positions in Algorithm 6 such that the entries in L and R equal $l_1, \ldots, l_{\delta+1}$, respectively $r_1, \ldots, r_{\delta+1}$, for each position p in S_2.

The interesting part is the update of L and R, when i is shifted to $i+1$. Recall from the description of rank updates in Chapter 3 that the rank of characters occurring between i and the next occurrence of $S_1[i]$, respectively the end of $S_1[i, |S_1|]S_2$, decreases by one, while the rank of $c_{old} = S_1[i]$ increases by the number of different characters between the two occurrences, or becomes ∞ if no further occurrence exists. Therefore, we update L and R in two separate steps, for occurrences of c_{old} and for all remaining positions. Since the entries in L and R can change strongly for positions of the first type, we compute them anew from scratch. For this purpose, we iterate through S_2 once from left to right and once from right to left, remembering the last $\delta + 1$ non-redundant occurrences of characters with rank greater

than the new rank of c_{old}. Once we reach an occurrence of c_{old}, we fill the corresponding entries in L, respectively R, with the currently remembered positions. This part of the update is in $O(n\delta)$, if the remembered positions are kept sorted (by index), a property that can be achieved with a queue-like data structure in which the oldest element is removed once the queue is full and a new element is to be added. We only need to watch out for duplicate occurrences of a character. If we are about to add such an occurrence, we need to remove the previous occurrence from the queue (although it might not be the oldest element).

For a position p with $S_2[p] \neq c_{old}$ the corresponding entries in L and R can only change if $\textsc{Rank}[S_2[p]]$ is smaller than $\textsc{Rank}[c_{old}]$. Also, there is only one way L, respectively R, can change for such a position p, namely if an occurrence of c_{old} is located close enough to p in S_2 to become an entry in L, respectively R. This is a direct consequence of Observation 3 in Chapter 3 which states that the rank difference of two characters other than c_{old} can not change its sign. To update L and R for these positions, we iterate through S_2 once from left to right and once from right to left remembering the latest occurrence of c_{old}. Each time, we reach a position with rank smaller than the rank of c_{old}, we go through its entries in L, respectively R, insert — if necessary — the last remembered occurrence of c_{old} at the appropriate position and shift the other entries accordingly. Clearly, this part of the update is also in $O(n\delta)$.

The initialization of L and R takes $O(n^2)$ time. Both parts of the update are in time $O(n\delta)$ and take place $O(n)$ times. Therefore, the total time spent on computing and maintaining tables L and R for Algorithm 6 is in time $O(n^2\delta)$. The additional space consumption is in $O(n\delta)$ which is subsumed by the overall space complexity of $O(n^2)$.

4.4.2 Precomputation of left-most and right-most essential positions

In Algorithm 6, we used the scan of the neighborhood of every position p in S_2 also to determine for the corresponding l_x and r_y the left-most, respectively right-most, essential position of intervals with left border l_x+1, respectively right border r_y-1. To precompute these values, we need two additional tables L' and R' that are of the same format as L and R. We study in the following only the computation and maintenance of L'. (The operations dealing with R' are analogous.) L' should be maintained such that for all $1 \leq p \leq |S_2|$ and $1 \leq x \leq \delta + 1$ entry $L'[p][x]$ corresponds to the closest position to the right of $L[p][x]$ that contains a character with rank smaller or equal to $\textsc{Rank}[S_2[p]]$. Clearly, we can initialize L' for $i = 1$ at no extra cost during the initialization of L which involves a scan of the neighborhood of each position in S_2. The update of L' at positions p with $S_2[p] = c_{old}$ is also straight-forward: During the scan of S_2 for updating L, we track not

only the last $\delta+1$ non-redundant occurrences of characters with rank greater than $\text{RANK}[c_{old}]$ but also for each of them the next position with rank at most $\text{RANK}[c_{old}]$. We find such a position at the latest when we reach p. Therefore, all entries $L'[p][x]$ for $S_2[p] = c_{old}$ are set during this process.

For all other p, the entries $L'[p][x]$ can only change if $\text{RANK}[c_{old}] > \text{RANK}[S_2[p]]$. To update L' at these positions, we precompute for all occurrences of c_{old} the $\delta + 1$ next positions to the right with strictly decreasing rank lower than $\text{RANK}[c_{old}]$. We store these positions in a list called *lower ranked neighbors* in the following. Later on, we check for each p that is not an occurrence of c_{old} every $L'[p][x]$ for being an occurrence of c_{old}, and replace each $L'[p][x]$ for which this is the case by the first lower ranked neighbor of $L'[p][x]$ that has rank at most $\text{RANK}[S_2[p]]$. We always find such a position among the lower ranked neighbors of $L'[p][x]$, because there can occur no more than δ different characters with rank greater than δ between $L[p][x]$ and p. Therefore, the update of L' is in $O(n\delta)$ if we can determine the lower ranked neighbors for all occurrences of c_{old} in time $O(n\delta)$. To achieve this, we scan S_2 and remember each occurrence of c_{old}. Each time we read a character with smaller rank at a position q, we test the most recent occurrence of c_{old} for having an uncompleted list of lower ranked neighbors. If this is the case, we compare the rank of $S_2[q]$ to the rank of the last element in the list. If it is smaller, we add q to the list and continue the testing with the previous occurrence of c_{old}. Once either of the two tests fails for an occurrence of c_{old}, it fails for all occurrences further to the left, so we can stop testing. In total, every list is changed at most $\delta + 1$ times for each update, and it happens only $O(n)$ times that we fail to insert an element which we find out in constant time. Hence, the total time for identifying lower ranked neighbors of c_{old} is in $O(n\delta)$.

Therefore, the complete computation and maintenance of L' also takes time $O(n\delta)$. For symmetry reasons, the same is true for R'. The space consumption of L' and R' is in $O(n\delta)$.

4.4.3 Better bounds for the total size of $\mathcal{C}$

The second efficiency problem in Algorithm 6 are the two iterations through the set $\mathcal{C}$ of optimal δ-locations of the previous reference interval (lines 8 and 9). We have seen earlier that the size of $\mathcal{C}$ is in $O(n(\delta+1))$ for each reference interval. To improve the upper bound of the algorithm complexity, we need to show that the average size of $\mathcal{C}$ over all steps of the algorithm, is lower than that.

To this end, we need to revisit the similarities between optimal δ-locations of successive reference intervals. So far, we reuse in Algorithm 6 only intervals with no occurrence of c, the character by which the two reference intervals differ. Now, we show that also the remaining optimal δ-locations can be partly inherited between successive reference intervals:

Observation 7 *Every interval $[a, b]$ in a string S that is an optimal δ-location of a $C' \subseteq \Sigma$ is an optimal δ-location of $C = C' \cup \{c\}$ for every $c \in \mathcal{CS}(S[a, b])$.*

Proof: Since $[a, b]$ is C'-optimal and contains an occurrence of c, it is also C-optimal. Also, the distance constraint holds because of $D(\mathcal{CS}(S[a, b]), C') \geq D(\mathcal{CS}(S[a, b]), C)$. □

However, we have seen already in Example 14 that the opposite is not true. This part of the relationship between optimal δ-locations of C and C' is characterized as follows:

Observation 8 *Every interval $[a, b]$ in a string S that is an optimal δ-location of a $C \subseteq \Sigma$ is an optimal δ-location of $C' = C \setminus \{c\}$ for $c \in \mathcal{CS}(S[a, b])$ and $c \in C$ if and only if (i) $D(\mathcal{CS}(S[a, b]), C) < \delta$ holds, and if (ii) there are positions p, p_ℓ and p_r in S with $a \leq p_\ell < p < p_r \leq b$, $S[p_\ell] \in C'$, $S[p_r] \in C'$ and $S[p] = c$.*

Proof: $\Rightarrow$: From $D(\mathcal{CS}(S[a, b]), C') \leq \delta$ it follows that $D(\mathcal{CS}(S[a, b]), C) < \delta$ because $c \in C$ and $c \in \mathcal{CS}(S[a, b])$, but $c \notin C'$. Let p_ℓ and p_r be left-most, respective right-most, essential position of $[a, b]$ with respect to C'. Condition (ii) then follows from the facts that $[a, b]$ is C'-optimal and $c \notin C'$.
$\Leftarrow$: $D(\mathcal{CS}(S[a, b]), C') \leq \delta$ holds, as $D(\mathcal{CS}(S[a, b]), C') = D(\mathcal{CS}(S[a, b]), C) + 1$ and $D(\mathcal{CS}(S[a, b]), C) < \delta$. It follows from (ii) that c is contained in the C'-essential subinterval of $[a, b]$. Since C and C' differ only in c, $[a, b]$ has to be C'-optimal □

We know from Observation 4 that optimal δ-locations containing an occurrence of c, the latest character in the reference interval $[i, j]_{S_1}$, are the only candidates for being non-inheritable from the previous reference interval. Observation 8 defines the only two types of these non-inheritable optimal δ-locations: The first type are intervals having exactly distance δ to $C = \mathcal{CS}(S_1[i, j])$, and the second type are intervals with left-most and/or right-most essential character c that contain no "inner occurrences" of c, i.e. positions that are separated from both interval boundaries by characters from C other than c.

Non-inherited optimal δ-locations of the second type are necessarily either of the form $[l_x + 1, r_1 - 1]$ or $[l_1 + 1, r_y - 1]$. Only such intervals can be C-optimal intervals with right-most essential position p, respectively left-most essential position p. Since there are only $\delta + 1$ many values of both, x and y, for every newly marked position, only $O(\delta+1)$ many of these intervals exist. Each of them can persist in $\mathcal{C}$ for at most $O(\delta + 1)$ iterations of the algorithm. Each time, either the new character of C is present in the interval which can happen at most δ times, otherwise the distance would have been above δ in the beginning, or the new character is not present. The latter can also happen at most δ times. Afterwards the distance is above δ due to missing characters alone. With $O(n^2)$ newly marked positions, the total number

of intervals of the second type inserted into $\mathcal{C}$ is bounded by $O(n^2(\delta+1))$. With an persistence time of $O(\delta)$, these intervals contribute $O(n^2(\delta+1)^2)$ to $\sum_{i\leq j}|\mathcal{C}_{i,j}|$, where $\mathcal{C}_{i,j}=\mathcal{C}$ at the time when reference interval $[i,j]$ is being processed.

Of the first interval type, we have $O((\delta+1)^2)$ many for each newly marked position and each can persist $O(\delta+1)$ iterations in $\mathcal{C}$. However, every such interval needs to be re-detected at least as often in line 15 of Algorithm 6 as it is inherited "silently", i. e. without being generated anew. This is because the distance to the reference character set increases by one, each time an interval is silently inherited. Since the initial distance is δ, we have at least one re-detection for every silent inheritance. Therefore, every optimal δ-location of the second type persists in $\mathcal{C}$ only $O(1)$ iterations before it is redetected. In total, these intervals contribute like the first type with $O(n^2(\delta+1)^2)$ to $\sum_{i\leq j}|\mathcal{C}_{i,j}|$.

Therefore the total time spent on the second iteration through $\mathcal{C}$ to remove non-optimal δ-locations (line 9) is in $O(n^2(\delta+1)^2)$. The first iteration through $\mathcal{C}$ (line 8) to remove intervals containing the latest character from the reference interval can be omitted if we test each optimal δ-location detected in line 16 for being already present in $\mathcal{C}$. This can be done efficiently if we first build a sorted list of the new elements and merge it to the old $\mathcal{C}$. Choosing the right order to process the intervals $[l_x+1, r_y-1]$ around a recently marked position p (**for**-loop of line 15), we obtain a sorted list at no extra cost for each p. Adding only those intervals for which p is the left-most occurrence of c, we can simply append the lists of different occurrences of c and gain a sorted redundancy-free list of all new optimal δ-locations that can be merged in linear time with $\mathcal{C}$. Repeating this process in every iteration of the algorithm keeps $\mathcal{C}$ sorted and free of redundancies. It follows that the total time spent on merging list is bounded by $\sum_{i\leq j}|\mathcal{C}_{i,j}|$, the total size of $\mathcal{C}$ summed over all iterations of the algorithm. Based on these findings, we claim the correctness of the following theorem:

Theorem 4 *Using the optimizations described in this section, Algorithm 6 can be improved to solve Problem 16 in time $O(n^2(\delta+1)^2)$ using $O(n^2)$ space.*

Chapter 5

Reference gene clusters

With the algorithmic results of the previous chapter, we are now in the position to develop efficient solutions to the problem of reference-based approximate gene cluster detection. In Section 2.3, we defined two variants of this problem that differ in the way how distances to approximate cluster occurrences are constrained, either separately for each input genome or collectively via their sum. Neither problem has been studied before in literature. Therefore, the aim of this chapter is to show that both problems can be solved efficiently in regard to both time and space complexity.

We formulate our search strategies for reference gene clusters and their approximate occurrences in a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ separately for the two distance constraining modes. We begin with sum distance constrained reference gene clusters and show later on how our approach can be adapted to the pairwise distance constraint. Finally, we show which changes are necessary in both modes to use the quorum parameter.

Throughout this chapter, we restrict the term "approximate occurrences of a reference gene cluster" to the intervals in the solution set of Problem 4, respectively Problem 6. These intervals are optimal with respect to their reference gene cluster C and occur in a combination of approximate common intervals for which C is an intersecting close set under the given distance constraint.

5.1 Computation of sum distance constrained reference gene clusters

Obviously, we can extract all sum distance constrained reference gene clusters in $\mathcal{S}$, as well as their approximate occurrences, from the corresponding set of approximate common interval combinations for the given distance threshold δ_{sum}. However, the generation of these combinations is rather inefficient, as it inherits the redundancy problem that already occurred with common intervals as seen in Section 3.5. Considering not only perfect but also ap-

proximate locations even increases the combinatorial overhead. We show that the detection of reference gene clusters and its approximate occurrences is feasible without intermediate generation of interval combinations.

5.1.1 Detection of reference gene clusters

Since a reference gene cluster has a perfect location in at least one input string, the search space for reference gene clusters is limited to the character sets of maximal intervals in $\mathcal{S}$. There are only $O(kn^2)$ character sets of this form, a small number compared to $O(n^{2k})$ possible approximate common interval combinations. We test each of these character sets for being a reference gene cluster. For that purpose, we compute for each candidate set C its optimal δ_{sum}-locations in $\mathcal{S}$. If there is at least one of these intervals in each string, we take the best from each, i.e. the one with lowest distance to C, and test if their total distance to C is at most δ_{sum}. If this is the case, C is clearly a reference gene cluster. Otherwise, it is not — for the following reasons: First, it follows from Observation 1 that there is always a C-optimal interval among the best approximate locations of C in a string. So, we can not build any better interval combinations using approximate locations that are not C-optimal. Second, intervals that are not even approximate locations of C are no alternative. Either their pairwise distance is already greater than δ_{sum}, or their intersection with C is empty which disqualifies them in the first place for building interval combinations with intersecting close set C.

5.1.2 Detection of approximate cluster occurrences

To detect the approximate occurrences of a reference gene cluster C, we simply need to identify among the C-optimal δ_{sum}-locations those combinable with others into k-tuples with a total distance to C of at most δ_{sum}. For this step, it is not necessary to actually build these combinations. We can simply pre-compute the minimal distance between C and the intervals in each string and then test for each candidate interval if its combination with the respective best approximate locations in the other $k-1$ strings meets the distance constraint. If it does, the candidate interval is an approximate occurrence of C. Otherwise, it is not for the same reason as above. We can not build a better approximate common interval combination for this candidate interval, at least none for which C is intersecting close set.

5.1.3 Algorithmic implementation

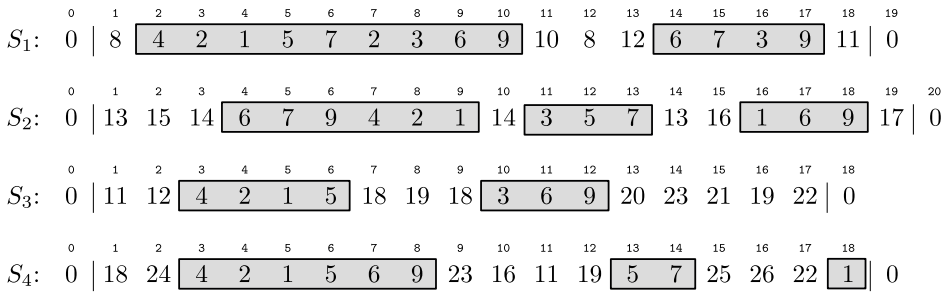
In practice, the computation of reference gene clusters and their approximate occurrences can be combined into a single process. Algorithm 7 contains pseudocode of such a combined search strategy which we explain in the following. We identify all candidate reference gene clusters one after the other by iterating through the maximal intervals in $\mathcal{S}$ (**for**-loops in lines 1

Algorithm 7 Sum distance constrained reference gene cluster computation in $\mathcal{S} = \{S_1, \ldots, S_k\}$ for distance threshold δ_{sum}

```
1: for each S_ℓ = S_1, ... S_k do
2:   for each maximal interval [i, j] in S_ℓ do
3:     C ← CS(S_ℓ[i, j])
4:     if C has no location in S_1, ..., S_{ℓ−1} and S_ℓ[1, i − 1] then
5:       for each S_ℓ' = S_1, ... S_k do
6:         C[ℓ'] ← set of optimal δ_sum-locations of C in S_ℓ'
7:       end for
8:       if C[ℓ'] ≠ ∅ for all C[ℓ'] = C[1], ... C[k] then
9:         for each ℓ' = 1, ..., k do
10:          minDist[ℓ'] ← min_{[a,b]∈C[ℓ']} {D(C, CS(S_ℓ'[a, b]))}
11:        end for
12:        minDist ← Σ_{1≤ℓ'≤k} minDist[ℓ']
13:        if minDist ≤ δ_sum then
14:          for each ℓ' = 1, ... k do
15:            maxDistShare[ℓ'] ← δ_sum − minDist + minDist[ℓ']
16:            remove from C[ℓ'] all [a, b] with D(C, CS(S_ℓ'[a, b])) >
               maxDistShare[ℓ']
17:          end for
18:          output C and C[1], ..., C[k]
19:        end if
20:      end if
21:    end if
22:  end for
23: end for
```

and 2). To ensure that every character set C is processed only once, we skip those with a previous perfect location in $\mathcal{S}$ (line 4). For the others, we need to detect the C-optimal δ_{sum}-locations in $\mathcal{S}$ (lines 5 to 7). This can be achieved by applying Algorithm 6 (Section 4.1) to all combinations of S_ℓ and S_1 to S_k. In this way, we get for every character set C all optimal δ_{sum}-locations in $\mathcal{S}$ sorted by their string of origin. This partitioning into sets, named $\mathcal{C}_1, \ldots, \mathcal{C}_k$ in the following, simplifies the subsequent test, whether there is at all a δ_{sum}-location in each string (line 8), as well as the identification of the distances between C and the best δ_{sum}-location in each string (line 10). If this distance complies with the distance threshold (line 13), C is a reference gene cluster, and we identify its approximate occurrences. For this purpose, we compute the value $\mathit{maxDistShare}[\ell'] = \delta_{sum} - \mathit{minDist} + \mathit{minDist}[\ell']$ for each $\mathcal{C}_{\ell'}$ (line 15). It corresponds to the maximum distance an element of $\mathcal{C}_{\ell'}$

Interval marking for $C = \mathcal{CS}(S_1[2,10])$ in strings $\mathcal{S} = \{S_1, S_2, S_3, S_3\}$:



List of optimal δ-locations of C for $\delta = 7$:

$\mathcal{C}[1]$: $[\mathbf{2},\mathbf{10}](\mathbf{0})$, $[2,17](3)$, $[14,17](4)$ — $minDist[1] = 0$

$\mathcal{C}[2]$: $[4,9](2)$, $[\mathbf{4},\mathbf{13}](\mathbf{1})$, $[4,18](3)$, $[11,13](5)$, $[11,18](4)$, $[15,18](5)$ — $minDist[2] = 1$

$\mathcal{C}[3]$: $[3,6](4)$, $[\mathbf{3},\mathbf{12}](\mathbf{3})$, $[10,12](5)$ — $minDist[3] = 3$

$\mathcal{C}[4]$: $[\mathbf{3},\mathbf{8}](\mathbf{2})$, $[3,14](5)$, $[13,14](6)$, $[18,18](7)$ — $minDist[4] = 2$

(Distance to C is given in parentheses after each interval)

Sum of minimum distances: $6 \Rightarrow C$ is reference gene cluster

---

Maximum distance share per string and list of approximate occurrences of C:

$maxDistShare[1] = 1$ — $\mathcal{C}[1]$: $[2,10]$ (0)

$maxDistShare[2] = 2$ — $\mathcal{C}[2]$: $[4,9]$ (2), $[4,13]$ (1)

$maxDistShare[3] = 4$ — $\mathcal{C}[3]$: $[3,6]$ (4), $[3,12]$ (3)

$maxDistShare[4] = 3$ — $\mathcal{C}[4]$: $[3,8]$ (2)

Figure 5.1: Extract from the computation of sum distance constrained reference gene clusters in a set of four strings $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$ (as defined in the figure): The character set of interval $[2,10]_{S_1}$ is tested for being a reference gene cluster for $\delta_{sum} = 7$. After computing the list $\mathcal{C}[\ell']$ of δ_{sum}-locations for each string, the sum of the minimum distances of the lists is compared to δ_{sum}. As the sum is below the threshold, $\mathcal{CS}(S_1[2,10])$ is a reference gene cluster and the approximate occurrences need to be identified. To this end, the maximum distance share for the intervals of each string is computed and the elements in the $\mathcal{C}[\ell']$ are filtered, based on these values.

can possibly have to be still combinable with elements of the other $k-1$ sets to a k-tuple of approximate common intervals with intersecting close set C for distance threshold δ_{sum}. All intervals that exceed the maximal distance share of their string of origin are removed from the respective $\mathcal{C}_{\ell'}$ (line 16). Therefore, the $\mathcal{C}_{\ell'}$ contain afterwards exactly the approximate occurrences of C and are reported together with C. An example of this procedure is shown in Figure 5.1.

5.1.4 Complexity analysis

Before we start with the complexity analysis of Algorithm 7, we need to recall two central results from Chapter 4: The first one is the runtime of Algorithm 6. We have shown that it is possible to compute for all maximal intervals in a string of length $O(n)$ the optimal δ-locations in another string of length $O(n)$ in time $O(n^2(\delta+1)^2)$. Since there are $O(n^2)$ maximal intervals in such a string, this averages to an amortized cost of $O((\delta+1)^2)$ for computing the optimal δ-locations of a single maximal interval. The second result is on the number of δ-locations a character set can possibly have in a string. We have shown that the sum of these numbers over the character sets of all maximal intervals in a string is in $O(n^2(\delta+1)^2)$. Therefore, we can apply again the amortization argument and say that the (amortized) number of δ-locations of a single character set is in $O((\delta+1)^2)$.

We now come to the actual complexity analysis of Algorithm 7: The iteration through all maximal intervals in $S_1, \ldots S_k$ is in time $O(kn^2)$. For each maximal interval, we compute 0-locations (line 4) and δ_{sum}-locations (lines 5 to 7) in $\mathcal{S}$ which is in time $O(k(\delta_{sum}+1)^2)$ according to the amortized analysis above. In terms of complexity analysis, the remaining operations of the algorithm (lines 8 to 18) can be categorized into two types. For operations of the first type, we iterate through the elements of the $\mathcal{C}[\ell']$ either to access the pairwise distance to C associated with each interval (lines 10 and 14) or to remove, respectively output, elements (lines 16 and 18). If the distances to C are stored for each interval in the $\mathcal{C}[\ell']$ (which is possible without extra cost as these values are computed anyway during Algorithm 6), the distance comparison takes time $O(1)$. Clearly, the removal and the reporting of a single element can also be performed in time $O(1)$. From the amortized size analysis of the $\mathcal{C}[\ell']$ follows immediately that these operations take time $O(k(\delta_{sum}+1)^2)$ for each reference interval. The second type of operations comprises computations, respectively tests, that involve only a single value, respectively look-up, for each $\mathcal{C}[\ell']$ (lines 8 and 10). These operations are obviously in time $O(k)$ for each reference interval. In total, we have $O(kn^2)$ reference intervals for which operations of amortized time $O(k(\delta_{sum}+1)^2)$ are performed. Thus, the total runtime of Algorithm 7 is in time $O(k^2n^2(\delta_{sum}+1)^2)$, while the space requirements are in $O(kn^2)$. Therefore, we claim the correctness of the following theorem:

Algorithm 8 Outline of pairwise distance constrained reference gene cluster computation in $\mathcal{S} = \{S_1, \ldots, S_k\}$ for distance threshold δ_{pw}

```
1: for each S_ℓ = S_1, ... S_k do
2:   for each maximal interval [i, j] in S_ℓ do
3:     C ← CS(S_ℓ[i, j])
4:     if C has no location in S_1, ..., S_{ℓ-1} and S_ℓ[1, i − 1] then
5:       for each S_ℓ' = S_1, ... S_k do
6:         C[ℓ'] ← set of optimal δ_pw-locations of C in S_ℓ'
7:       end for
8:       if C[ℓ'] ≠ ∅ for all C[ℓ'] = C[1], ... C[k] then
9:         output C and C[1], ..., C[k]
10:      end if
11:    end if
12:  end for
13: end for
```

Theorem 5 *Algorithm 7 computes in a set of k strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ all sum distance constrained reference gene clusters and their approximate occurrences for distance threshold δ_{sum} in time $O(k^2 n^2 (\delta_{sum} + 1)^2)$ using $O(kn^2)$ space.*

5.2 Computation of pairwise distance constrained reference gene clusters

In the following, we show that the detection of pairwise distance constrained reference gene clusters and their approximate occurrences is also possible in the same way, i.e. without intermediate generation of approximate common interval combinations.

5.2.1 Search strategy under the pairwise distance constraint

In Algorithm 8, we illustrate how the previous search strategy can be adapted to the pairwise distance constraint. The generation of reference intervals and the test of their character sets C for previous locations in $\mathcal{S}$ is analogous to Algorithm 7. Then follows the generation of the optimal δ_{pw}-locations of C. As their identification depends only on the value of the distance threshold, not on the underlying distance constraining mode, we can employ here the same method as for sum distance constrained reference gene clusters. The differences begin with the test of C for being a reference gene cluster which turns out to be much simpler under the pairwise distance constraint: We only need to check whether every string has an optimal

δ_{pw}-location of C (line 8). If this is the case, every combination of these intervals, one from each string, is a k-tuple of approximate common intervals with intersecting close set C for δ_{pw}. Therefore, it follows not only that C is a reference gene cluster in $\mathcal{S}$, but also that all elements of the $\mathcal{C}_\ell$ are approximate occurrences as defined in Problem 6. Passing this test is also the only way C can be a reference gene cluster. This follows immediately from Observation 1 and the fact that a reference gene cluster intersects with its approximate common intervals. Also, for obvious reasons, no interval not contained in the $\mathcal{C}_\ell$ can be in the solution set of Problem 6. The correctness of Algorithm 8 follows directly from these considerations.

5.2.2 Complexity analysis

We come now to the complexity analysis of Algorithm 8. The generation of optimal δ_{pw}-locations of the character sets of all reference intervals in $\mathcal{S}$ is in time $O(k^2n^2(\delta_{pw}+1)^2)$. Once these intervals are generated for a character set C, the most costly operation is to iterate through the corresponding $\mathcal{C}[\ell']$ (line 9) to output the elements in case that C is a reference gene cluster. Based on the amortization argument, this takes time $O(k(\delta_{pw}+1)^2)$ for each of the $O(kn^2)$ character sets. Therefore the asymptotic time complexity of the complete algorithm is in $O(k^2n^2(\delta_{pw}+1)^2)$. The space requirements are in $O(kn^2)$ due to the parallel computation of optimal δ-locations in k strings. Based on these considerations, we claim the correctness of the following theorem:

Theorem 6 *Algorithm 8 computes in a set of k strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ all pairwise distance constrained reference gene clusters and their approximate occurrences for distance threshold δ_{pw} in time $O(k^2n^2(\delta_{pw}+1)^2)$ using $O(kn^2)$ space.*

5.3 Detection of q-covering reference gene clusters

To compute reference gene clusters that occur only in a subset of the input strings, we need to introduce a quorum parameter q into our search strategy that defines the minimum number of strings that need to be covered by the cluster occurrences. For the pairwise distance constraint this adaptation is trivial: Instead of testing whether a candidate C has a C-optimal δ_{pw}-location in all k strings, we test if there is one in at least q out of k strings. If this is the case, C is a q-covering reference gene cluster for pairwise distance constraint and the elements of the $\mathcal{C}_\ell$ are its approximate occurrences.

The adaptation for the sum distance constraint is more involved due to the dependency of the distance threshold on the number of strings covered by approximate cluster occurrences. Recall that we decrease the distance threshold by $\frac{\delta_{sum}}{k}$ for each missing string. A simple approach to test a

character set C for being q-covering reference gene cluster is to check for each q', $q \leq q' \leq k$, whether there is an $\mathcal{S}' \subseteq \mathcal{S}$ with $|\mathcal{S}'| = q'$ in which C is a reference gene cluster for distance threshold $q'\frac{\delta_{sum}}{k}$. To perform this test, we can simply check if the q' smallest of the minimum distance values sum up to at most $q'\frac{\delta_{sum}}{k}$. If this holds for at least one q', C is a q-covering reference gene cluster, otherwise it is not. However, this test can be simplified: It is easy to see that this test fails for every $q' > q$ if it fails for $q' = q$. This is because the average distance available for each string, i.e. $\frac{\delta_{sum}}{k}$, is independent of q'. If this value is exceeded for the q smallest distances, it is exceeded for every $q' > q$. To test a candidate C for being a reference gene cluster, it is therefore sufficient to identify the q smallest minimum distance values and check if their sum is at most $q\frac{\delta_{sum}}{k}$. Only if this is the case, C is a q-covering reference gene cluster under the sum distance constraint.

For every C that is a q-covering reference gene cluster, we need to identify its approximate occurrences. These correspond to C-optimal intervals $[a,b]_{S'_\ell}$ that occur in approximate common interval combinations for an $\mathcal{S}' \subseteq \mathcal{S}$, $q \leq |\mathcal{S}'| \leq k$, with intersecting close set C for distance threshold $|\mathcal{S}'|\frac{\delta_{sum}}{k}$. Clearly, we get the smallest sum distance for $|\mathcal{S}'|$-tuples containing $[a,b]_{S'_\ell}$ if we combine this interval with the $|\mathcal{S}'|-1$ best approximate locations of the $|\mathcal{S}|-1$ strings other than $S_{\ell'}$. Since $D(C, \mathcal{CS}(S_{\ell'}[a,b]))$ is not necessarily among the q smallest minimum distance values, it is not sufficient here to consider only $|\mathcal{S}'| = q$. Instead, we adapt the average distance trick as follows: We partition the minimum distances of the $k-1$ strings other than S_ℓ into values of at most $\frac{\delta_{sum}}{k}$ and bigger values. Then, we combine $D(C, \mathcal{CS}(S_{\ell'}[a,b]))$ with all r values of at most $\frac{\delta_{sum}}{k}$ plus possibly the next $(q-(r+1))$ bigger values (to cover at least q strings). Let q' be the number of distances in this combination. If their sum is at most $q'\frac{\delta_{sum}}{k}$, $[a,b]_{S'_\ell}$ is an approximate occurrence of C. Moreover, one can easily verify that in case this threshold is exceeded, $[a,b]_{S'_\ell}$ can not be an approximate occurrence of the q-covering reference gene cluster C.

Obviously, the asymptotic time complexity of Algorithm 7 does not change for the pairwise distance constraint when a quorum parameter is used. We show that the same is true for the sum distance constraint: For testing a character set for being a reference gene cluster, we only need to identify the q smallest values in a set of k distance values in the range $0, \ldots, \delta_{sum}$ and compute their sum, which is possible in time $O(k(\delta_{sum}+1))$ using a bucket sort on the distances. To detect the approximate occurrences of a q-covering reference gene cluster, we need to build the partitioning based on $\frac{\delta_{sum}}{k}$, find the corresponding q' and compute the sum of the q' smallest distances. All three operations take only $O(k)$ time, once the distance values are sorted. Therefore, the time spent on each candidate interval is in $O(k(\delta_{sum}+1))$ which is subsumed by the time used for detecting the $\mathcal{C}[\ell']$.

Chapter 6

Median gene clusters

We now come to the first one of the two general approximate gene cluster discovery problems studied in this thesis, the computation of median gene clusters. Recall from Chapter 2 that this model differs from the sum distance constrained reference gene clusters in the fact that no reference occurrence of the gene cluster is required in the input genomes. This generalization expands the search space for gene cluster detection drastically: the number of candidate gene clusters is no longer polynomially bounded in the genome lengths, but increases exponentially with the alphabet size, i.e. the number of different gene family ids. Therefore, a fundamentally different search strategy needs to be employed.

In [71], Rahmann and Klau solve the more general weighted median gene cluster problem using an ILP approach. However, the performance of their program is rather low even for the unweighted case and only two input strings. The algorithms presented in the previous chapters solve these types of problem instances in polynomial time. This is possible because the problem of median gene cluster detection in two strings reduces essentially to the computation of optimal δ-locations. See Chapter 9 for a performance comparison.

In this chapter, we follow a different approach to solve Problems 7 and 8 of Section 2.4 for multiple strings. Since median gene clusters are not just sets of genes that are conserved at a certain rate in the given genomes but also gene sets that best represent a specific combination of their approximate occurrences, we begin our search not with subsets of the alphabet but with pairwise set intersecting interval combinations and then test whether their median complies with the given distance constraint. Clearly, the number of interval combinations increases exponentially with the number of input strings. However, for large alphabet sizes not all of them will be pairwise set intersecting and even less will have a median that complies with the given distance threshold, at least not if this value is chosen reasonably. Therefore,

our goal is to define a filter approach that rules out a large number of interval combinations before they are actually built and explicitly tests the remaining ones.

The structure of this chapter is as follows: First, we formulate our general search strategy for median gene cluster detection (Section 6.1), then we give algorithmic solutions to the different steps of our method (Sections 6.2 to 6.4), and show what changes are necessary to employ the quorum parameter (Section 6.5). Finally, we present some optimizations that do not affect the asymptotic time complexity of our approach but have the potential to achieve notable speed-ups in terms of practical running times (Section 6.6).

6.1 General search strategy

Our strategy for detecting median gene clusters is based on the observation that whenever a combination $(\mathcal{CS}(S_1[i_1, j_1]), \ldots, \mathcal{CS}(S_k[i_k, j_k]))$ of character sets of intervals from a set of $k \geq 2$ strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ has a median C for a given distance threshold δ, not only the total distance to the median is constrained but also the distances between the k character sets. The first observation follows immediately from the triangle inequality of the symmetric set distance:

Observation 9 *For any two character sets C_1, $C_2 \subseteq \Sigma$ with $D(C_1, C_2) > \delta$, there is no $C \subseteq \Sigma$ with $D(C, C_1) + D(C, C_2) \leq \delta$.*

For our search strategy this means that no two intervals need to be combined whose character sets have a pairwise distance greater than δ. No matter which additional $k-2$ intervals are added to complete the k-tuple, the sum distance to the median exceeds the threshold. To get a more powerful filter of the search space, we take a closer look on how the sum distance to the median can be distributed over the k character sets. Clearly, there is at least one pairwise distance between C and these character sets that is at most $\frac{\delta}{k}$, otherwise the sum of the distances would exceed δ. Therefore, the distance between such a character set C' and the $k-1$ other character sets is limited by the following upper bound:

Lemma 1 *Let $\mathcal{I} = ([i_1, j_1], \ldots, [i_k, j_k])$ be a k-tuple of intervals, $k \geq 2$, such that for a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ over alphabet Σ and a distance threshold $\delta \geq 0$ there exists a $C \subseteq \Sigma$ with $\sum_{\ell=1}^{k} D(C, \mathcal{CS}(S_\ell[i_\ell, j_\ell])) \leq \delta$. Then, $\mathcal{I}$ contains at least one $[i_m, j_m]$, $1 \leq m \leq k$, with $C' = \mathcal{CS}(S_m[i_m, j_m])$ and*

$$\sum_{\ell=1}^{k} D\big(C', \mathcal{CS}(S_\ell[i_\ell, j_\ell])\big) \;\leq\; 2\frac{k-1}{k}\delta. \tag{6.1}$$

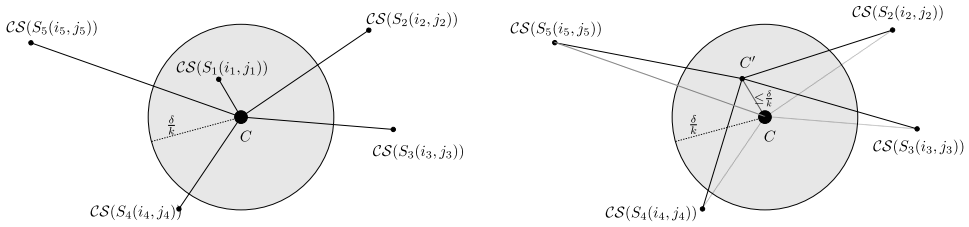


Figure 6.1: The sum distance to the median of character sets can be estimated via the sum distance to the cluster filter.

Proof: Choose $[i_m, j_m]$, $1 \leq m \leq k$ from the intervals in $\mathcal{I}$, such that $D(C, \mathcal{CS}(S_m[i_m, j_m])) \leq \frac{\delta}{k}$. Let $C' = \mathcal{CS}(S_m[i_m, j_m])$. From the triangle inequality we infer:

$$\begin{aligned}
\sum_{\ell=1}^{k} D\big(C', \mathcal{CS}(S_\ell[i_\ell, j_\ell])\big) &\leq \sum_{\ell \neq m} \Big(D(C', C) + D\big(C, \mathcal{CS}(S_\ell[i_\ell, j_\ell])\big)\Big) \\
&\leq (k-2)\frac{\delta}{k} + \sum_{\ell=1}^{k} D\big(C, \mathcal{CS}(S_\ell[i_\ell, j_\ell])\big) \\
&\leq (k-2)\frac{\delta}{k} + \delta \;\leq\; 2\frac{k-1}{k}\delta.
\end{aligned}$$

□

The connection between the sum distances to C and C' is visualized in Figure 6.1. Apparently, only interval combinations that contain an element whose character set is of the form of C' have the potential to yield a median gene cluster for distance threshold δ. Unfortunately, this condition is only necessary and not sufficient for the existence of a median gene cluster. Nevertheless, we begin our search with the detection of these character sets, named *(median) cluster filters* in the following. Afterwards, we test only interval combinations that contain a perfect location of at least one cluster filter for having a median that complies with distance threshold δ. A more detailed description of our approach is as follows:

1. First, compute all cluster filters C' in $\mathcal{S} = \{S_1, \ldots, S_k\}$. For that purpose, test all intervals in the strings $S_1, \ldots, S_k$ for having a character set that meets the conditions of a cluster filter.

2. Second, compute for each C' the pairwise set intersecting k-tuples of the form $([i_1, j_1], \ldots, [i_k, j_k])$ where all $[i_\ell, j_\ell]$, $1 \leq \ell \leq k$, are δ-locations of C' and at least one is a location of C' and inequality (6.1) holds.

3. Finally, compute for each k-tuple from Step 2 the corresponding median(s). Report all medians along with the k-tuple if they comply with the sum distance threshold δ and have the minimum cluster size s.

It follows immediately from the previous considerations that this approach solves Problems 7 and 8 stated in Section 2.4. The only flaw is that medians may be reported redundantly if they originate from multiple interval combinations. However, this problem can be easily resolved in a post-processing step. In the following sections, we have a closer look on how the three steps of the approach can be implemented.

6.2 Computation of cluster filters (Step 1)

For the first step of our search strategy, we observe that every cluster filter in $\mathcal{S}$ is in fact a reference gene cluster for the relaxed cluster filter distance threshold $\delta_{cf} = 2\frac{k-1}{k}\delta$. Therefore, we can apply Algorithm 7 to perform the first step of our search strategy. However, this algorithm is designed to compute not only reference gene clusters but also optimal δ_{cf}-locations which are not needed for detecting cluster filters. Since these intervals are not useful in the following steps (we will see later on why), we present a simplified approach that generates δ_{cf}-locations only as far as necessary for testing a candidate character set for being a reference gene cluster. Recall that for this purpose it is sufficient to determine only the minimum distance between the tested character set and the intervals of each string.

The pseudocode in Algorithm 9 implements the simplified search strategy. Its basic structure is equivalent to Algorithm 6 for detecting δ-locations, except for the two extra **for** loops in lines 1 and 11 which are needed to extend the approach to multiple strings. The generation of reference intervals (lines 2 to 9) and the generation of candidate intervals of approximate locations (lines 11 to 15) are also borrowed from Algorithm 6. They are only performed now for more than one string: the former successively for all strings in $\mathcal{S}$, and the latter in parallel for all strings except for the current reference string.

We also adopt the distinction between intervals that contain an occurrence of c, the latest character in the reference interval $[i,j]_{S_\ell}$, and intervals that do not contain such an occurrence. To compute the minimum distance between $\mathcal{CS}(S_\ell[i,j])$ and intervals of the form $[l_x+1, r_y-1]$, as defined in line 15, we simply compute each pairwise distance and compare it to the running minimum for the respective string (line 17). Due to Observation 1, it is not even necessary to test the $[l_x+1, r_y-1]$ for being optimal with respect to $\mathcal{CS}(S_\ell[i,j])$. Of course, it is possible that the minimum distance to $\mathcal{CS}(S_\ell[i,j])$ occurs with an interval that does not contain c. But for intervals of this type, the minimum distance to $\mathcal{CS}(S_\ell[i,j])$ is bounded by their minimum distance to the previous reference interval plus 1 for the missing occurrence of c. We account for this in line 10 by increasing for each string the respective minimum distance value by 1 after each extension of the reference interval. In doing so, it does not matter whether the old minimum

distance value actually originated from an interval containing no occurrence of c. This is because for other intervals, the true distances to $\mathcal{CS}(S_\ell[i,j])$ are computed anew in line 16. Therefore, no approximate locations need to be inherited between successive reference intervals to determine the minimum distances between $\mathcal{CS}(S_\ell[i,j])$ and the intervals in the different strings. To determine the minimum total distance, we simply add up the minimum pairwise distances for the k strings and compare the sum to the relaxed distance threshold δ_{cf} (line 21). In case the distance constraint is met, we report $\mathcal{CS}(S_\ell[i,j])$ as cluster filter. To avoid the redundant detection of cluster filters that have more than one perfect location in the input strings, we can reuse an idea from the previous chapter, where we skipped all intervals for which we found a perfect location either in a previous reference string, or earlier in the current reference string.

It follows from the previous considerations that Algorithm 9 detects all cluster filters in the given strings for the relaxed distance constraint δ_{cf}. The complexity analysis of Algorithm 9 can be mostly adopted from the previous chapters. We are doing at no point substantially more work than in Algorithm 7. In fact, the opposite is true: We save the effort spent on dealing with lists of inherited approximate locations and on testing the optimality of intervals. However, these changes have no impact on the asymptotic time complexity, as the **for**-loop in line 15 is executed $O(k^2n^2(\delta+1)^2)$ times. The space complexity is $O(kn^2)$ due to table NUM.

6.3 Generation of k-tuples from maximal δ-locations (Step 2)

To perform the second step of our algorithm, we need to build for each cluster filter C' all k-tuples $([i_1,j_1],\ldots,[i_k,j_k])$ of maximal δ-locations with pairwise intersecting character sets that have a perfect location of C' among them and satisfy inequality (6.1). To this end, it makes no difference whether we build all combinations for a single cluster filter in a row, or if we build them piecewise as its perfect locations become reference intervals in Algorithm 9. Therefore, our strategy is to build those k-tuples containing a perfect location $[i_m,j_m]_{S_m}$ of a cluster filter C' at the time $[i_m,j_m]_{S_m}$ is the reference interval in Algorithm 9. In doing so, our dynamic data structures, RANK, L and R, are in the right state at the time maximal δ-locations of $C' = \mathcal{CS}(S_m[i_m,j_m])$ need to be collected in the remaining $k-1$ strings to build the corresponding k-tuples.

We only need to adapt our redundancy test for cluster filters to this new work flow: Since interval combinations are now built on the basis of a specific cluster filter location, it is no longer possible to skip automatically intervals that are not the first location of the cluster filter. Otherwise, we are likely to miss some interval combinations for the cluster filter. However, what we

Algorithm 9 Computation of cluster filters for median gene clusters detection in $\mathcal{S} = \{S_1, \ldots, S_k\}$ for distance threshold $\delta_{cf} = 2\frac{k-2}{k}\delta$

1: **for each** $\ell = 1, \ldots, k$ **do**
2: **for** $i = 1, \ldots, |S_\ell|$ **do**
3: $minDist[\ell'] \leftarrow 0$ for all $1 \leq \ell' \leq k$
4: $j \leftarrow i$
5: **while** $j \leq |S_\ell|$ **and** $[i, j]_{S_\ell}$ is left-maximal **do**
6: $c \leftarrow S_\ell[j]$
7: **while** $S_\ell[i, j]$ is not right-maximal **do**
8: $j \leftarrow j + 1$
9: **end while**
10: $minDist[\ell'] \leftarrow minDist[\ell'] + 1$ for all $1 \leq \ell' \leq k$
11: **for each** $\ell' = 1, \ldots, \ell-1, \ell+1, \ldots, k$ **do**
12: **for each** position p in $S_{\ell'}$ with $S_{\ell'}[p] = c$ **do**
13: mark position p in S_2
14: find positions $l_1, \ldots, l_{\delta+1}$ and $r_1, \ldots, r_{\delta+1}$
15: **for each** interval $[l_x + 1, r_y - 1]$ with $1 \leq x, y \leq \delta + 1$ **do**
16: $dist \leftarrow D(\mathcal{CS}(S_{\ell'}[l_x + 1, r_y - 1]), \mathcal{CS}(S_\ell[i, j]))$
17: $minDist[\ell'] \leftarrow \min\{minDist[\ell'], dist\}$
18: **end for**
19: **end for**
20: **end for**
21: **if** $\sum_{\ell'=1}^{k} minDist[\ell'] \leq \delta_{cf}$ **then**
22: output $\mathcal{CS}(S_\ell[i, j])$
23: **end if**
24: $j \leftarrow j + 1$
25: **end while**
26: **end for**
27: **end for**

can do to avoid redundancy is to combine perfect locations of a cluster filter only then into a k-tuple, when the first location in $\mathcal{S}$ is the reference interval.

We split the description of Step 2 into two substeps. First, we show how all maximal δ-locations of the character set of a reference interval can be extracted efficiently from $\mathcal{S}$. Then we show how the corresponding k-tuples can be enumerated.

6.3.1 Collection of δ-locations of a cluster filter

When searching for maximal δ-locations of a cluster filter C', the techniques developed in Chapter 4 for computing optimal δ-locations can not be reused. This is for two reasons: In general, maximal intervals do not follow the "$[l_x+1, r_y-1]$ scheme" used previously for generating candidates of optimal δ-locations, but may start/end within marked blocks, i.e. the neighboring characters may be elements of C. Additionally, maximal δ-locations need to be generated only for those character sets of reference intervals that are a cluster filter. Therefore, the continuity in inheriting (maximal) δ-locations between successive reference intervals is lost. Instead, we compute maximal δ-locations anew from scratch for each cluster filter C' using the current state of the data structures RANK, L and R each time a perfect location $[i_m, j_m]_{S_m}$ is processed as reference interval. Pseudocode of this approach is given in Algorithm 10.

To ensure that all δ-locations are detected, we iterate for each $c \in C'$ through all its occurrences p in $S_{\ell'}$ (lines 1 and 2). Then we find borders $l_{\delta+1}(p)$ and $r_{\delta+1}(p)$ as in Algorithm 9 either by direct search or by a look-up in tables L and R. We iterate through all maximal intervals between $l_{\delta+1}(p) + 1$ and $r_{\delta+1}(p) - 1$ that contain position p (**for** loops in lines 4 and 11) and keep track of the number of the characters not occurring in C' (lines 8 and 15). Finally, we compute the distance of the current interval to C' (line 18) which equals $|\mathcal{CS}(S_m[i_m, j_m])| - |\mathcal{CS}(S_{\ell'}[l, r])| + 2d_{\text{total}}$ to test whether $[l, r]_{S_{\ell'}}$ meets the distance constraint. The tests in lines 5, 7, 12 and 14 ensure that no interval is considered more than once by stopping the interval extension once we either read another character from C' with higher rank than $S[p]$ or find another occurrence of $S[p]$ further left than p.

We analyze the asymptotic runtime of Algorithm 10 for *one* cluster filter: When tables L, R and NUM are available, the runtime of Algorithm 10 is $O(n^2)$, since each substring of S_2 is processed at most once and in constant time. For multiple strings, the computation of maximal δ-locations has to be done for the $k-1$ strings that are not the origin of the cluster filter. This can be done independently for each of these strings, so that the time complexity is in $O(kn^2)$. The practical running times should mainly depend on the number of maximal δ-locations contained in the strings and the fraction of the strings covered by them. Both numbers will be rather small for large alphabet sizes and reasonable values of δ.

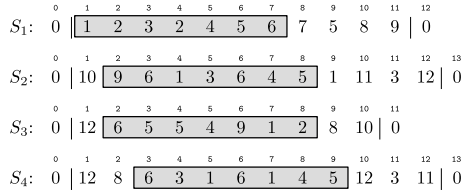
Algorithm 10 Detection of maximal δ-locations of $C' = \mathcal{CS}(S_m[i_m, j_m])$ in $S_{\ell'}$ for a reference interval $[i_m, j_m]_{S_m}$ of Algorithm 9 if C' is cluster filter

```
1: for each c ∈ C' do
2:   for each position p in S_ℓ' with S_ℓ'[p] = c do
3:     d_left ← 0
4:     for l = p, ..., L[p][δ + 1] + 1 do
5:       if S_ℓ'[l] ∈ C' and RANK[S_ℓ'[l]] ≥ RANK[c] and l ≠ p then
6:         break
7:       else if S_ℓ'[l] ∉ C' and S_ℓ'[l] ∉ CS(S_ℓ'[l + 1, p]) then
8:         d_left ← d_left + 1
9:       end if
10:      d_total ← d_left
11:      for r = p, ..., R[p][δ + 1] − 1 do
12:        if S_ℓ'[r] ∈ C' and RANK[S_ℓ'[r]] > RANK[c] then
13:          break
14:        else if S_ℓ'[r] ∉ C' and S_ℓ'[r] ∉ CS(S_ℓ'[l, r − 1]) then
15:          d_total ← d_total + 1
16:        end if
17:        dist ← |CS(S_m[i_m, j_m])| − |CS(S_ℓ'[l, r])| + 2d_total
18:        if dist < δ then
19:          δ-loc[ℓ'][dist] ← δ-loc[ℓ'][dist] ∪ [l, r]
20:        end if
21:      end for
22:    end for
23:  end for
24: end for
```

6.3.2 Generation of k-tuples from δ-locations

For enumerating all candidate k-tuples for a cluster filter C', assume that the information about its δ-locations is stored in a table δ-loc of size $k \times (\delta + 1)$ that stores at position δ-loc$[\ell][dist]$ the list of δ-locations in S_ℓ, $1 \leq \ell \leq k$, that have exactly distance $dist$ to C'. For the reference string S_m, the origin of C', the list at position δ-loc$[m][0]$ contains exactly one element $[i_m, j_m]$ with $\mathcal{CS}(S_m[i_m, j_m]) = C'$. All other lists δ-loc$[m][dist]$, $dist > 0$, are empty. We can then use a recursive traversal of δ-loc to build all combinations of δ-locations of C' that contain exactly one δ-location from each of the k strings. As mentioned before, we should build k-tuples with multiple perfect locations only for the first one processed as reference interval. We can implement this simply by keeping the lists δ-loc$[\ell'][0]$ always empty once reference strings S_m with $m > \ell'$ are processed.

Due to inequality (6.1), we can use a branch-and-bound technique in the recursion that starts backtracking once the accumulated distances to C' in



	1	2	3	4	5	6	7	8	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0	0	0
$\mathcal{CS}(S_2[2,9])$	1	0	1	1	1	1	0	0	1
$\mathcal{CS}(S_3[2,8])$	1	1	0	1	1	1	0	0	1
$\mathcal{CS}(S_4[3,9])$	1	0	1	1	1	1	0	0	0
median	1	T	1	1	1	1	0	0	T

Figure 6.2: Majority vote over the character sets of a 4-tuple of intervals from a set of strings $\mathcal{S} = \{S_1, S_2, S_3, S_4\}$ (as defined in the figure). There are four alternative medians due to the ties for characters 2 and 9.

the current branch of the traversal exceed $\delta_{cf} = 2\frac{k-1}{k}\delta$. However, in the worst case even a single cluster filter may yield $\Theta(n^{2k})$ k-tuples. As we have seen in Example 12, this is already possible for $\delta = 0$. However, for strings of gene family ids where $|\Sigma|$ is in $\Theta(n)$ the actual numbers are likely to be much smaller, at least for reasonable values of δ. We will see in Chapter 9 for which parameter ranges our approach is feasible in practice.

6.4 Computation of the median(s) for k-tuples of intervals (Step 3)

Once an interval combination is generated from a cluster filter C', we only need to compute its median(s) and test whether the corresponding sum distance complies with the median gene cluster distance threshold δ. Only then the median(s) should be reported as median gene cluster(s).

6.4.1 Basic median computation

To compute the median(s) of a k-tuple of character sets $\mathcal{C} = (C_1, \ldots, C_k)$, we simply need to perform a majority vote over the elements of $\Sigma_{\mathcal{C}} = \bigcup_{1 \leq \ell \leq k} C_\ell$. This means that characters occurring in more than half of the tuple elements are in the median, while characters occurring in less than half of the elements are not, and characters that occur in exactly half of the elements may or may not be added to the median yielding different median variants. An example of this multiplicity is given in Figure 6.2. To avoid this redundancy, one can either define that all borderline elements are added to the median or not, or introduce an additional tie state T for these characters as used in Figure 6.2. In either case, we get a unique median representation. We refer to the median that contains all borderline elements as the *maximal* median.

Since all medians have the same sum distance to $C_1, \ldots, C_k$, we need to test only once whether this value complies with the median gene cluster distance threshold. The computation of this distance can be done character-wise based on the following observation:

Observation 10 *Given a k-tuple of sets $(C_1, \ldots, C_k)$ over an alphabet $\Sigma_{\mathcal{C}}$ for which $M \subseteq \Sigma_{\mathcal{C}}$ is a median, let p_c be the number of sets C_ℓ, $1 \leq \ell \leq k$, that contain character $c \in \Sigma$, and let n_c be the number of sets that do not contain c, then $p_c + n_c = k$ holds. Furthermore, let $d_c := \min\{p_c, n_c\}$, then it holds:*

$$\sum_{\ell=1}^{k} d(M, C_\ell) = \sum_{c \in \mathcal{C}} d_c. \tag{6.2}$$

The time complexity of median computation is in $O(k|\Sigma_{\mathcal{C}}|)$, as the computation of both the median representation and the total distance can be done by a simple iteration through the characters $\Sigma_{\mathcal{C}}$.

At this point, we have introduced all three steps of our approach for median gene cluster detection. Their execution is visualized in Figure 6.3 exemplarily for a single cluster filter. We show finally that a combination of the Steps 2 and 3 can be used to improve the search space pruning.

6.4.2 Iterative median computation

While a median computation itself is not very time-consuming, it is the sheer number of k-tuples to be tested that make this step of the algorithm expensive. Remember that the number of k-tuples that originate from a single cluster filter can be exponential in k. However, these k-tuples are not all disjoint but share some elements as they originate from the recurrence through the table δ-loc that stores the maximal δ-locations as described in Section 6.3. We can take advantage of this observation by computing medians iteratively, adding one element of the k-tuple after the other and computing intermediate medians after each step. Updating the distance to the intermediate median can then be done according to the following lemma:

Lemma 2 *Let $C_1, \ldots, C_k$ be character sets over Σ and let M_j be the maximal median of $\{C_1, \ldots, C_j\}$, $1 \leq j \leq k$. Define $p_{c,j} = |\{\ell \mid 1 \leq \ell \leq j, c \in C_\ell\}|$ and $n_{c,j} = |\{\ell \mid 1 \leq \ell \leq j, c \notin C_\ell\}|$. Then the following equation holds:*

$$\sum_{\ell=1}^{j} d(M_j, C_\ell) = \sum_{\ell=1}^{j-1} d(M_{j-1}, C_\ell) + |C_j \setminus M_{j-1}| + |\{c \in M_{j-1} \setminus C_j : n_{c,j-1} \neq p_{c,j-1}\}|. \tag{6.3}$$

Proof: Based on equation (6.2), the proof can be done separately for each character $c \in \mathcal{C}$. Let $d_{c,j} = min\{n_{c,j}, p_{c,j}\}$. There are four cases to be considered: (i) $c \notin M_{j-1}$ and $c \notin C_j$, (ii) $c \in M_{j-1}$ and $c \in C_j$, (iii) $c \notin M_{j-1}$ and $c \in C_j$ and (iv) $c \in M_{j-1}$ and $c \notin C_j$.
(i) It holds that $n_{c,j-1} > p_{c,j-1}$ and therefore $d_{c,j-1} = p_{c,j-1}$. From $c \notin C_j$, we get $n_{c,j} = n_{c,j-1} + 1$ and $p_{c,j} = p_{c,j-1}$ so that $d_{c,j} = d_{c,j-1}$.
(ii) It holds that $p_{c,j-1} > n_{c,j-1}$ and therefore $d_{c,j-1} = n_{c,j-1}$. From $c \in C_j$,

Step 1: Testing $C' = \mathcal{CS}(S_1[1,7])$ for being cluster filter in $\mathcal{S} = \{S_1, S_2, S_3\}$ for $\delta = 3$:

	0	1	2	3	4	5	6	7	8	9	10	11	12		
S_1:	0	1	2	3	2	4	5	6	7	5	8	9	0		$minDist[1] = 0$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
S_2:	0	10	9	6	1	3	6	4	5	1	11	3	12	0	$minDist[2] = 1$
	0	1	2	3	4	5	6	7	8	9	10	11			
S_3:	0	12	6	5	5	4	9	1	2	8	10	0			$minDist[3] = 2$

Sum of minimum distances: $\sum_{\ell=1}^{3} minDist[\ell] = 3 \leq 2\frac{k-1}{k}\delta \Rightarrow C'$ is cluster filter

---

Step 2(a): Computation of maximal δ-locations of C'

S_2:	0-location:	-
	1-location:	[3,9]
	2-location:	[5,8], [6,9], [2,9], [3,11], [3,7]
	3-location:	[5,7], [6,10], [6,8], [1,9], [2,11], [2,7], [3,6], [7,11], [7,9], [3,12]
S_3:	0-location:	-
	1-location:	-
	2-location:	[2,8]
	3-location:	[1,8], [2,9], [2,7], [2,5], [3,8]

Step 2(b): Combinations of maximal δ-locations that fulfill inequality (7.1)

distance to C': 3	distance to C': 4
$\begin{pmatrix} [1,7] \\ [3,9] \\ [2,8] \end{pmatrix}$	$\begin{pmatrix} [1,7] \\ [3,9] \\ [1,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,9] \\ [2,9] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,9] \\ [2,7] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,9] \\ [2,5] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,9] \\ [3,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [5,8] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [6,9] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [2,9] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,11] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,7] \\ [2,8] \end{pmatrix}$

Step 3: Computation of median gene clusters

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[3,9])$	1	**0**	1	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	**1**
median	**1**	**1**	**1**	**1**	**1**	**1**	0

total distance: 3

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[3,9])$	1	**0**	1	1	1	1	0
$\mathcal{CS}(S_3[1,8])$	1	1	**0**	1	1	1	**1**
median	**1**	**1**	**1**	**1**	**1**	**1**	0

total distance: 3

	1	2	3	4	5	6	8	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0	0
$\mathcal{CS}(S_2[3,9])$	1	**0**	1	1	1	1	0	0
$\mathcal{CS}(S_3[2,9])$	1	1	**0**	1	1	1	**1**	**1**
median	1	1	1	1	1	1	0	0

total distance: 4

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	**1**	1	1	1	1	0
$\mathcal{CS}(S_2[3,9])$	1	0	1	1	1	1	0
$\mathcal{CS}(S_3[2,7])$	1	0	**0**	1	1	1	**1**
median	**1**	0	**1**	**1**	**1**	**1**	0

total distance: 3

	1	2	3	4	5	6
$\mathcal{CS}(S_1[1,7])$	1	**1**	1	1	1	1
$\mathcal{CS}(S_2[3,9])$	1	0	1	1	1	1
$\mathcal{CS}(S_3[2,5])$	**0**	0	**0**	1	1	1
median	**1**	0	**1**	**1**	**1**	**1**

total distance: 3

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[3,9])$	1	**0**	1	1	1	1	0
$\mathcal{CS}(S_3[3,8])$	1	1	**0**	1	1	**0**	**1**
median	1	1	1	1	1	1	0

total distance: 4

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[5,8])$	**0**	**0**	1	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	**1**
median	1	1	1	1	1	1	0

total distance: 4

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	**1**	1	1	1	0
$\mathcal{CS}(S_2[6,9])$	1	**0**	0	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	0	1	1	1	**1**
median	**1**	**1**	0	**1**	**1**	**1**	0

total distance: 3

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	**0**
$\mathcal{CS}(S_2[2,9])$	1	**0**	1	1	1	1	1
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	1
median	**1**	**1**	**1**	**1**	**1**	**1**	**1**

total distance: 3

	1	2	3	4	5	6	9	11
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0	0
$\mathcal{CS}(S_2[3,11])$	1	**0**	1	1	1	1	0	**1**
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	**1**	1	0
median	1	1	1	1	1	1	0	0

total distance: 4

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[3,7])$	1	**0**	1	1	**0**	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	**1**
median	1	1	1	1	1	1	0

total distance: 4

Median gene clusters for $\delta = 3$: {1,2,3,4,5,6}, {1,3,4,5,6}, {1,2,4,5,6}, {1,2,3,4,5,6,9}

Figure 6.3: Extract from median gene cluster computation for $\delta = 3$ in $\mathcal{S} = \{S_1, S_2, S_3\}$ (as defined in the figure): The 3-step approach is performed for the character set $\mathcal{CS}(S_1[1,7])$. Among the four detected median gene clusters, $\{1,2,3,4,5,6,9\}$ is the only one with no reference occurrence in $\mathcal{S}$.

we get $p_{c,j} = p_{c,j-1} + 1$ and $n_{c,j} = n_{c,j-1}$ so that $d_{c,j} = d_{c,j-1}$.
(iii) It holds that $n_{c,j-1} > p_{c,j-1}$ and therefore $d_{c,j-1} = p_{c,j-1}$. From $c \in C_j$, we get $p_{c,j} = p_{c,j-1} + 1$ and $n_{c,j} = n_{c,j-1}$ so that

$$d_{c,j} = \begin{cases} p_{c,j} & \text{if } p_{c,j-1} < n_{c,j-1} - 1 \\ n_{c,j} & \text{if } p_{c,j-1} = n_{c,j-1} - 1 \end{cases}$$

For both cases, we get $d_{c,j} = d_{c,j-1}+1$ since the distance increases by 1 either because $c \notin M_j$ or, in case $c \in M_j$, it follows that $d_{c,j} = n_{c,j} = d_{c,j-1} + 1$.
(iv) It holds that $p_{c,j-1} \geq n_{c,j-1}$ and therefore $d_{c,j-1} = n_{c,j-1}$. From $c \notin C_j$, we get $n_{c,j} = n_{c,j-1} + 1$ and $p_{c,j} = p_{c,j-1}$ so that

$$d_{c,j} = \begin{cases} n_{c,j} & \text{if } n_{c,j-1} < p_{c,j-1} \\ p_{c,j} & \text{if } n_{c,j-1} = p_{c,j-1} \end{cases}$$

In the first case, we get $d_{c,j} = d_{c,j-1} + 1$ as $c \in M_j$ but $\notin C_j$. For the second case with $c \in M_{j-1}$ but $c \notin M_j$, we get $d_{c,j} = p_{c,j} = p_{c,j-1} = n_{c,j-1} = d_{c,j-1}$. Since there are no other cases, the value of $d_{c,j}$ increases by one if and only if $c \in C_j \setminus M_{j-1}$ or $c \in M_{j-1} \setminus C_j$ with $n_{c,j-1} \neq p_{c,j-1}$. In all other cases $d_{c,j} = d_{c,j-1}$ holds. □

We can use the iterative distance computation of Lemma 2 to speed up the recursion through the lists of δ-locations of the k strings. When adding a new interval $[i, j]$, we additionally compute how the distance to the intermediate median changes and start backtracking once this distance is bigger than δ. To compute the changes in the intermediate distance efficiently, we keep track of n_c and p_c for each character c that occurs in at least one of the intervals already added to the k-tuple. Then, we can test for each character in constant time whether one of the two cases occurs where the intermediate distance increases. It is not necessary to generate the intermediate medians explicitly. Instead, only the final median is computed once the k-tuple is completely filled.

The speed-up of the iterative median computation is two-fold: Firstly, we can stop adding new δ-locations once the distance to an intermediate median is greater than δ, since by adding another character set the distance never decreases. Secondly, for k-tuples that share the first $\ell < k$ elements, the first ℓ intermediate distance computations are performed only once.

6.5 Introduction of a quorum parameter

Up to now our search strategy finds only median gene clusters that have an approximate occurrence in each input string. To evade this constraint, we can employ a quorum parameter, $2 \leq q \leq k$, as we already did for perfect and reference gene clusters. Once again, it is desirable to integrate this parameter directly into the search strategy rather than running the algorithms

separately for each $\mathcal{S}' \subseteq \mathcal{S}$. We show how this can be accomplished in our search strategy for median gene clusters.

The basic idea is as follows: During the recursive generation of interval combinations, we add up to $k-q$ empty intervals that contribute each $\frac{\delta}{k}$ to the sum distance to the median. A convenient way of implementing these changes is to insert empty intervals, one for each input string, into the δ-loc tables. Then we can basically adopt the branch and bound strategy for generating interval combinations. However, we should take care that empty intervals are not used for intermediate median computation. Also, we should avoid that more than $k-q$ empty intervals are combined into a k-tuple. To this end, we can simply track the number of empty occurrences in the current branch and start backtracking, once we accumulated too many of them. Besides this new bound, we can reuse the bound for the sum distance to the median. As we are subtracting the average distance share for each empty cluster occurrence, it does not matter for this bound how many intervals in a k-tuple are actually empty.

For the cluster filter threshold, we need to consider the following things: When a single string is not covered by a gene cluster, the threshold $\delta_{cf} = 2\frac{k-1}{k}\delta$ changes to $\delta'_{cf} = 2\frac{k-2}{k-1}(\delta - \frac{\delta}{k})$ for the remaining strings. One can easily verify that the difference between these values equals $2\frac{\delta}{k}$:

$$\begin{aligned}
\delta_{cf} - \delta'_{cf} &= 2 \cdot \delta \cdot \left(\frac{k-1}{k} - \left(\frac{k-2}{k-1} \cdot \left(1 - \frac{1}{k}\right) \right) \right) \\
&= 2 \cdot \delta \cdot \left(\frac{k-1}{k} - \left(\frac{(k-2)\cdot k}{(k-1)\cdot k} - \frac{k-2}{(k-1)\cdot k} \right) \right) \\
&= 2 \cdot \delta \cdot \left(\frac{k-1}{k} - \left(\frac{(k-2)\cdot (k-1)}{(k-1)\cdot k} \right) \right) \\
&= 2 \cdot \delta \cdot \left(\frac{(k-1)-(k-2)}{k} \right) \\
&= 2 \cdot \frac{\delta}{k}
\end{aligned}$$

Moreover, it follows from induction over the number of omitted strings that this difference holds not only for the first but for every missing string. Therefore, we change the cluster filter bound such that for each empty interval in the current search branch, we subtract $2\frac{\delta}{k}$ from the remaining cluster filter distance.

The worst case time complexity is not affected by the quorum parameter, as at most $O(kn^{2k-2})$ additional interval combinations need to be built for a cluster filter which is subsumed by the overall time complexity $O(n^{2k})$. However, in practice the extra work can be significant, as we can no longer skip automatically cluster filters lacking a δ-location in at least one input string.

6.6 Algorithm optimizations

Due to the enumeration of interval combinations in Step 2, our approach to median gene cluster computation has exponential worst-case time complexity. Although, in practice, the number of generated k-tuples is likely much lower than the $\Theta(n^{2k})$ possible interval combinations, it is still far from being polynomially bounded. Nevertheless, we can try to optimize our algorithm such that a substantial number of genomes can be processed in reasonable running time. There are four main targets to optimizing our approach: (i) We can try to rule out cluster filters that can not yield a median gene cluster. (ii) We can try to rule out intervals from δ-loc tables that can not be combined to any k-tuple that yields a median gene cluster. (iii) We can try to improve pruning criterions for the branch and bound phase. (iv) We can preprocess the input genomes to identify segments that should be either completely contained in a cluster filter location or not at all. In this section, we study optimizations from all four categories.

6.6.1 Minimum cluster filter size

Our first optimization helps us to rule out cluster filters that are too small to yield a median gene cluster of minimum size s. Clearly the minimum size of a cluster filter C' has to be $s - \frac{\delta}{k}$. Otherwise,

$$D(C', C) \geq \max\{\underbrace{|C|}_{\geq s} - \underbrace{|C'|}_{< s - \frac{\delta}{k}}, \underbrace{|C'|}_{< s - \frac{\delta}{k}} - \underbrace{|C|}_{\geq s}\} > \frac{\delta}{k}$$

holds for all median gene clusters C, a contradiction to C' being a cluster filter. Therefore, we can skip in the first place the enumeration of interval combinations for such small C'.

6.6.2 Infrequent characters in cluster filters

Our second idea to rule out cluster filters is to count characters that occur not frequently enough in the input strings to ever be an element of a median. This is always the case if a character occurs in less than half of the input strings. We call such characters *infrequent* in $\mathcal{S}$. Every majority vote will exclude such characters from the median. Hence, no character set may contain more than $\frac{\delta}{k}$ characters of this type to be a cluster filter. To test this property efficiently for the character sets generated in Step 1, the character frequencies should be precomputed in advance for all elements of Σ.

6.6.3 Suboptimal interval borders

When we restrict our search to intersecting median gene clusters, which are the only ones meaningful in a biological context, we can rule out even more

cluster filters. Recall from Section 2.4 that for intersecting median gene clusters, it is sufficient to consider only combinations of intervals that are all optimized with respect to the same median. So far, we did not make use of this observation, as the prospect medians are unknown at the time interval combinations are built. However, we can still take advantage of this restriction:

Assume a k-tuple of intervals that are all optimized with respect to an intersecting median C. Then, the left-most and right-most essential character in each interval may not be infrequent in $\mathcal{S}$. Otherwise, the interval would not be optimized with respect to the median. Therefore, if we find out for an interval that for every interval combination its left-most and/or right-most essential characters with respect to all intersecting medians are infrequent, we can completely skip this interval in Step 2, or if the interval is the perfect location of a cluster filter, we can skip Step 2 entirely. Figuring out whether this holds may seem unrealistic at first glance, as the left-most and right-most essential character of an interval depend on the composition of a specific median which is unknown at the time it is needed for determining left-most and right-most essential positions. In general, this is true. However, for intervals $[i_\ell, j_\ell]_{S_\ell}$ in which $c = S_\ell[i_\ell]$ has no other occurrence than position i_ℓ, this is feasible. This is because either c occurs in the respective median, or it does not. In the former case, c is the left-most essential character, and in the latter case, $[i_\ell, j_\ell]_{S_\ell}$ is not compact. In either case, we can skip the interval under the premise that c occurs in less than half of the input strings. Analogous considerations hold for $S_\ell[j_\ell]$. Since multiple character occurrences in substrings are not the rule for large alphabet sizes, this seems to be worth testing, since it can rule out a complete execution of Step 2, while the test itself takes only constant time given that character frequencies are precomputed.

In fact, we can make even more use of this partial determination of the median content: When building interval combinations in Step 2, there is always one fixed element $[i_m, j_m]_{S_m}$ which is the origin of the cluster filter. If $S_m[i_m]$ and $S_m[j_m]$ occur uniquely in this interval, we know that $S_m[i_m]$ and $S_m[j_m]$ are contained in every median C for which an interval combination containing $[i_m, j_m]_{S_m}$ is optimized. (Otherwise, $[i_m, j_m]_{S_m}$ is not C-compact.) Moreover, we know that $S_m[i_m - 1]$ and $S_m[j_m + 1]$ are not contained in any of these medians. (Otherwise, $[i_m, j_m]_{S_m}$ is not closed.) Therefore, we test every interval $[i_\ell, j_\ell]_{S_\ell}$ in δ-loc for the following properties:

(*i*) $S_\ell[i_\ell - 1] \neq S_m[i_m] \wedge S_\ell[j_\ell + 1] \neq S_m[i_m]$,

(*ii*) $S_\ell[i_\ell - 1] \neq S_m[j_m] \wedge S_\ell[j_\ell + 1] \neq S_m[j_m]$.

Only if all these tests turn out positive, $[i_m, j_m]_{S_m}$ and $[i_\ell, j_\ell]_{S_\ell}$ need to be combined in the enumeration phase in Step 2. Otherwise, we can remove $[i_\ell, j_\ell]_{S_\ell}$ from δ-loc. If only one of the characters $S_m[i_m]$ or $S_m[j_m]$ has a

unique occurrence in $[i_m, j_m]_{S_m}$, we perform only the first, respectively the second test. We get additional tests if $S_\ell[i_\ell]$ and $S_\ell[j_\ell]$ are unique occurrences in $[i_\ell, j_\ell]_{S_\ell}$:

(iii) $S_\ell[i_\ell] \neq S_m[i_m - 1] \wedge S_\ell[i_\ell] \neq S_m[j_m + 1]$,

(iv) $S_\ell[j_\ell] \neq S_m[i_m - 1] \wedge S_\ell[j_\ell] \neq S_m[j_m + 1]$.

For these tests the same applies as above: If any of them fails, we can remove $[i_\ell, j_\ell]_{S_\ell}$ from δ-loc. Again, we can only use a part of the tests if only one of the characters $S_\ell[i_\ell]$ or $S_\ell[j_\ell]$ is unique in $[i_\ell, j_\ell]_{S_\ell}$.

The same test can also be used to decide if two intervals from table δ-loc can be combined into an optimized k-tuple. This does not reduce the size of the table, as such non-combinable intervals may still be combinable with other intervals. Instead we can precompute the pairwise combinability of the elements of δ-loc and use this information later in the branch and bound phase.

6.6.4 Lower bounds for sum distances to cluster filters

In the branch and bound approach used for enumerating interval combinations, we use two pruning criteria based on the accumulated distances to the cluster filter and the (iterative) median. We show that the bound for the first criterion can be easily improved.

So far, we start backtracking once we accumulate a sum distance to the cluster filter that exceeds $2\frac{k-1}{k}\delta$. However, after combining $l < k$ intervals there are still $k - l$ intervals to be added from the remaining strings $S_{\ell+1}, \ldots, S_k$. Their distances to the cluster filter are not necessarily zero, and a lower bound on them could be used to start backtracking earlier. Clearly, after combining ℓ intervals, the sum distance to these intervals plus the lower bound for the remaining strings may not exceed $2\frac{k-1}{k}\delta$. If it does, we can start backtracking. The lower bounds can be computed easily: We simply precompute for each $1 \leq \ell < k$ the sum of the minimum distances to the intervals from each $S_{\ell+1}, \ldots, S_k$. This needs to be done only once for each table δ-loc.

6.6.5 Lower bounds for distances to prospective medians

Improving the second pruning criterion is slightly more involved, as there is no fixed median during the branch and bound phase.

At first, we derive a lower bound on the distance between each element $[i_\ell, j_\ell]_{S_\ell}$ of δ-loc with $C^* = \mathcal{CS}(S_\ell[i_\ell, j_\ell])$ and any prospective median C of interval combinations from δ-loc containing $[i_\ell, j_\ell]_{S_\ell}$. The Venn diagram in Figure 6.4 visualizes all possible relationships between these two sets and the corresponding cluster filter. Let f be the number of infrequent characters in $C^* \cap C'$. Then the following inequality holds:

$$
\begin{aligned}
D(C^*, C) &= |C^* \setminus C| + |C \setminus C^*| \\
&= |(C^* \cap C') \setminus C| + |(C^* \setminus C') \setminus C| + |(C \cap C') \setminus C^*| + |(C \setminus C') \setminus C^*| \\
&= (6) + (7) + (1) + (4) \\
&= (6) + (7) + (1) + (4) + (3) - (3) + (5) - (5) + (6) - (6) + (4) - (4) \\
&= \big((7) + (3) + (1) + (5)\big) - \big((3) + (5) + (6) + (4)\big) + 2\big((6) + (4)\big) \\
&= \underbrace{D(C^*, C')}_{known} - \underbrace{D(C', C)}_{\leq \frac{\delta}{k}} + 2\Big(\big(\underbrace{(C^* \cap C')}_{\geq f} \setminus C\big) + \big(\underbrace{(C \setminus C') \setminus C^*}_{\geq 0}\big)\Big) \\
&\geq D(C^*, C') - \frac{\delta}{k} + 2f.
\end{aligned}
$$

An alternative estimation of $D(C^*, C)$ is: $D(C^*, C) \leq f'$, where f' is the number of infrequent characters in C^*. Therefore, the complete estimation of the lower bound reads:

$$D(C^*, C) \geq \max\left\{D(C^*, C') - \tfrac{\delta}{k} + 2f, f'\right\}$$

In case $|C^*| < s$, we can extend f' to $f' + (s - |C^*|)$, as there must be at least $s - |C^*|$ characters in each median that do not occur in the respective interval.

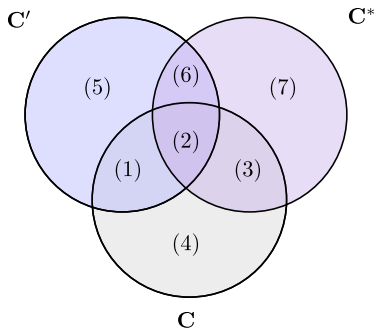
This estimation can be used for two purposes. First, we can rule out elements of δ-loc that have distance greater than δ to every prospect median. Second, if we compute for every level of δ-loc the sum of the minimum distances between the prospect median and the intervals with the best lower bounds, $\max\left\{D(C^*, C') - \frac{\delta}{k} + 2f, f'\right\}$, in the remaining strings, we can improve the iterative median computation if we add these minimum distances to those distances accumulated for the strings processed already.

6.6.6 Unfragmented entities

The last optimization technique presented in this section is a preprocessing of the input strings. We identify intervals that are unfragmented in the sense that the contained characters occur elsewhere in the strings only in form of the complete substring, i.e. there are no fragmentary occurrences and within each occurrence there are no changes in the character order or the character copy numbers. We call such segments *unfragmented entities*.

From a practical point of view it makes no sense to study gene clusters whose occurrences cover only a part of unfragmented entities. Such a gene cluster is always suboptimal in the sense that its character set, as well as its approximate occurrences, can be extended to yield a bigger gene cluster with longer occurrences at no additional cost.

From a computational point of view, this modification of the gene cluster model has the potential to save a lot of effort in processing intervals that



(1): $(C \cap C') \setminus C^*$
(2): $(C \cap C') \cap C^*$
(3): $(C \cap C^*) \setminus C'$
(4): $(C \setminus C') \setminus C^*$
(5): $(C' \setminus C) \setminus C^*$
(6): $(C^* \cap C') \setminus C$
(7): $(C^* \setminus C') \setminus C$

Figure 6.4: Venn diagram showing all possible relations between the three set types involved in median gene cluster computation: a cluster filter C', a prospective median C and character sets $C^* \neq C'$ of intervals $[i_\ell, j_\ell]_{S_\ell}$ contained in a k-tuple generated for C' in Step 2.

start/end within unfragmented entities. However, the actual benefit depends on the number and length of unfragmented entities in the input strings.

It remains to be shown how we can implement this optimization. One approach is to collapse each unfragmented entity into a single character and compute gene clusters in the resulting strings. But then we have to keep in mind that there are characters of length greater than one and take the actual sizes into account when computing character set sizes or distances to intervals. Otherwise, we can substantially change the results. Alternatively, we can precompute the locations of unfragmented entities in the original strings and store them in a table. Then we can simply look-up whether the processed intervals cut an unfragmented entity and skip them in case they do.

As none of the optimizations presented in Section 6.6 has an impact on the worst-case time complexity of our approach, we study their practical running times on real genomes in Chapter 9.

Chapter 7

Center gene clusters

We now come to the last approximate gene cluster model studied in this thesis, the center gene clusters. This model differs from median gene clusters in the way representatives of approximate cluster occurrences are defined. Remember that under the sum distance constraint as used for median computation, pairwise distances to a consensus set are not directly constrained but only via their sum. Therefore, when it comes to median gene cluster computation, a rather large distance to a single character set can be compensated by less than average distances to others. Apparently, this effect grows stronger for increasing numbers of input strings up to the point where approximate occurrences consisting only of single genes may be included in approximate common interval combinations. To avoid such distorted conservation patterns, in the first place, we can use the center gene cluster model which differs from median gene clusters exactly in the point that pairwise distances to approximate cluster occurrences are constrained individually. If we additionally allow for missing occurrences in a small amount of input genomes using a quorum parameter, we should be able to detect the same gene clusters at lower distance thresholds.

The problem of center gene cluster detection has not been addressed before in the literature. However, a related problem, the identification of character sets with δ-locations covering a certain number of input strings, was studied by Chauve et al. [15]. The model presented there differs from ours in the absence of an optimality criterion, i.e. no center is computed for a certain approximate common interval combination. Instead all subsets of the alphabet Σ should be reported along with their δ-locations, in case these are distributed over a sufficient number of genomes. The authors present three equivalent formulations of this problem and sketch an algorithmic solution for each. All three solutions have in common that their starting points for gene cluster detection are not interval combinations but subsets from Σ that are within a certain distance range of the character sets of the substrings of the input strings.

In this chapter, we study how our approach for median gene cluster detection based on building interval combinations can be adapted to center gene clusters and how the computational complexity is affected thereby. After giving an overview of the general search strategy to solve Problems 9 and 10 of Section 2.5, we study in more detail the changes necessary to the different algorithm steps (Sections 7.2 to 7.5) and discuss the issue of the quorum parameter (Section 7.6). We conclude this chapter with a survey of the optimization techniques of the previous chapter to study which of them can be reused for center gene cluster computation (Section 7.7).

7.1 General search strategy

For the detection of center gene clusters, we can largely employ the cluster filter approach introduced in the previous chapter for median gene clusters. We only need to adapt our filter criterion to the new distance constraint.

First of all, we observe that in a combination of intervals with a center gene cluster for distance threshold δ no two elements have character sets with pairwise distance greater than 2δ:

Observation 11 *Let $\mathcal{I} = ([i_1, j_1], \ldots, [i_k, j_k])$ be a k-tuple of intervals, $k \geq 2$, such that for a set of strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ over alphabet Σ and a distance threshold $\delta \geq 0$ there exists a $C \subseteq \Sigma$ with*

$$\max_{1 \leq \ell \leq k} D(C, \mathcal{CS}(S_\ell[i_\ell, j_\ell])) \leq \delta. \tag{7.1}$$

Then, it holds for every interval $[i_m, j_m]$ in $\mathcal{I}$, $1 \leq m \leq k$, with character set $C' = \mathcal{CS}(S_m[i_m, j_m])$ that

$$\max_{1 \leq \ell \leq k} D(C', \mathcal{CS}(S_\ell[i_\ell, j_\ell])) \leq 2\delta. \tag{7.2}$$

The correctness of this relation follows immediately from the triangle inequality. We name character sets of the form of C' *center cluster filters*, or simply "cluster filters" if the underlying gene cluster model is clear from the context. The relation between a center cluster filter and a center gene cluster is visualized in Figure 7.1.

In our new filter approach, we use inequality 7.2 to replace the two distance thresholds introduced for median cluster filters, namely δ for the pairwise distance to single intervals and $\delta_{cf} = 2\frac{k-1}{k}\delta$ for the total distance to its best approximate locations in the $k-1$ remaining strings. At first glance this may seem as a weakening of the filter condition. However, if we take into account that the threshold δ $(= \delta_{sum})$ used for median gene clusters constrains sum distances while the threshold δ $(= \delta_{pw})$ used for center gene clusters constrains pairwise distances, it becomes clear that, in fact, the opposite is true. For detecting gene clusters with the same degree of

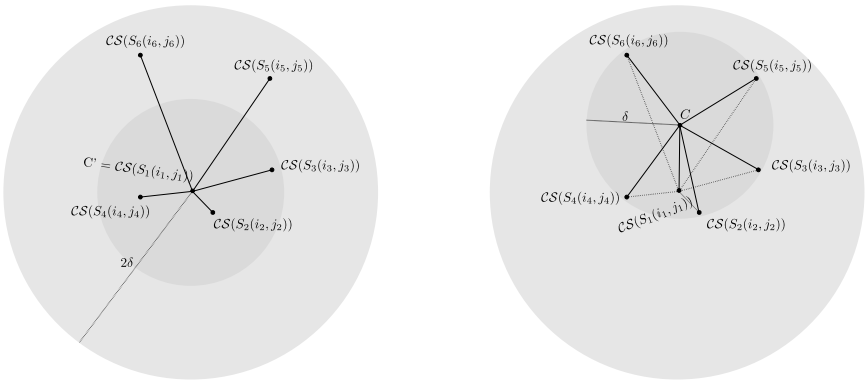


Figure 7.1: Relation between a center cluster filter C' and a center gene cluster C. Interval combinations that have a center C with maximum pairwise distance δ are in a (2δ)-range of a center cluster filter C'.

conservation under both the center and the median gene cluster model, we need to chose the corresponding distance thresholds such that approximately $\delta_{sum} = k \cdot \delta_{pw}$ holds. With regard to that, the constraints for the sum of $k-1$ pairwise distances to a cluster filter are the same for both median and center: $2\frac{k-1}{k}\delta_{sum} = (k-1) \cdot 2\frac{\delta_{sum}}{k} = (k-1) \cdot 2\delta_{pw}$. The constraint for pairwise distances, however, is stricter for the center as it is based on the average, and not the total distance available: $2\delta_{pw} < \delta_{sum}$ for all $k > 2$. Moreover, it follows from Observation 11 that in every k-tuple of intervals whose center meets a given distance threshold δ, all elements of the tuple are center cluster filters for δ. Therefore, we can detect the gene cluster and the corresponding k-tuple starting from each of the k intervals.

These findings give rise to the following approach for detecting center gene clusters: (1) Compute all cluster filters from a selected reference interval (for instance S_1), identify for each of them the maximal (2δ)-locations in the other $k-1$ strings and build all possible k-tuples. (2) Test each of these interval combinations for having a center that fulfills inequality 7.1, and in case they do, report them along with their center(s). There is only one problem with this approach: Unlike median computation which we have seen is trivial, the problem of center computation was proven to be NP-complete [28, 49]. However, it was recently shown that the problem is not only fixed parameter tractable for parameter δ [29, 55] but also in polynomial time if $\delta \in O(\log(\Sigma_{\mathcal{C}}))$ holds [55]. Although the latter restriction can not be guaranteed in gene cluster computation, these results suggest that center computation should be tractable for the instances occurring in our approach. Still, we would be in a better position if we kept the number of instances for which we need to perform this costly computation as low as possible. A simple method to reduce this number is to filter the generated k-tuples first

for having a median and compute centers only for the remaining instances. The soundness of this filter is guaranteed by the following observation:

Observation 12 *Given a k-tuple of character sets $\mathcal{C} = \{C_1, \ldots, C_k\}$ over an alphabet Σ. If $\mathcal{C}$ has a center $C \subseteq \Sigma$ with*

$$\max_{1 \leq \ell \leq k} \{D(C, C_\ell)\} \leq \delta$$

for $\delta \geq 0$, then there is also a median $M \subseteq \Sigma$ of $\mathcal{C}$ with

$$\sum_{\ell=1}^{k} D(M, C_\ell) \leq k \cdot \delta.$$

Proof: Clearly, $\sum_{\ell=1}^{k} D(C, C_\ell) \leq k \cdot \delta$ holds. Therefore, either C is a median of $\mathcal{C}$, or there is another $M \subseteq \Sigma$ with $\sum_{\ell=1}^{k} D(M, C_\ell) < \sum_{\ell=1}^{k} D(C, C_\ell)$ that is a median of $\mathcal{C}$. □

Adding this additional filter, we obtain a 4-step approach to the computation of center gene clusters that can be summarized as follows:

1. Compute all center cluster filters $C' \subseteq \Sigma$ in a fixed reference string $S_{ref} \in \mathcal{S} = (\{S_1, \ldots, S_k)\}$ for $\delta_{cf} = 2\delta$.

2. Compute for each C' the pairwise set intersecting k-tuples of the form $([i_1, j_1], \ldots, [i_k, j_k])$ where all $[i_\ell, j_\ell]$, $1 \leq \ell \leq k$, are maximal (2δ)-locations of C' and at least one is a perfect location of C'.

3. Discard all k-tuples generated in Step 2 that have no median that fulfills the sum distance threshold $k\delta$.

4. Finally, compute centers for the remaining k-tuples. Report all centers along with their k-tuple that comply with the sum distance threshold δ and have at least the minimum cluster size s.

The correctness of this approach follows from the previous considerations. However, like with median gene clusters, we have the problem that center gene clusters may be reported redundantly if they are found for different interval combinations. As with median gene clusters, we deal with this issue in a post-processing step. In Figure 7.2, the complete 4-step approach to center gene cluster computation is applied to a set of example strings.

In the following sections, we have a closer look at the different steps of our approach focusing, where applicable, on the differences to the corresponding steps for median gene cluster computation.

Step 1: Testing C' for being a center cluster filter in $\mathcal{S} = \{S_1, S_2, S_3\}$ for $\delta = 1$:

	0	1	2	3	4	5	6	7	8	9	10	11	12		
S_1:	0	1	2	3	2	4	5	6	7	5	8	9	0		$minDist[1] = 0$
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
S_2:	0	10	9	6	1	3	6	4	5	1	11	3	12	0	$minDist[2] = 1$
	0	1	2	3	4	5	6	7	8	9	10	11			
S_3:	0	12	6	5	5	4	9	1	2	8	10	0			$minDist[3] = 2$

Maximum of minimum distances: $\max_{1 \le \ell \le 3} \{minDist[\ell]\} = 2 \le 2\delta \Rightarrow C'$ is cluster filter

---

Step 2(a): Computation of maximal 2δ-locations of C'

S_2:	0-location:	-
	1-location:	[3,9]
	2-location:	[5,8], [6,9], [2,9], [3,11], [3,7]
S_3:	0-location:	-
	1-location:	-
	2-location:	[2,8]

Step 2(b): Combinations of maximal δ-locations

sum distance to C': 3	sum distance to C': 4
$\begin{pmatrix} [1,7] \\ [3,9] \\ [2,8] \end{pmatrix}$	$\begin{pmatrix} [1,7] \\ [5,8] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [6,9] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [2,9] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,11] \\ [2,8] \end{pmatrix}, \begin{pmatrix} [1,7] \\ [3,7] \\ [2,8] \end{pmatrix}$

Step 3: Computation of medians

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[3,9])$	1	**0**	1	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	**1**
median	**1**	**1**	**1**	**1**	**1**	**1**	0

total distance: 3

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[5,8])$	**0**	**0**	1	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	**1**
median	1	1	1	1	1	1	0

total distance: 4

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	**1**	1	1	1	0
$\mathcal{CS}(S_2[6,9])$	1	**0**	0	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	0	1	1	1	**1**
median	**1**	**1**	0	**1**	**1**	**1**	0

total distance: 3

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	**0**
$\mathcal{CS}(S_2[2,9])$	1	**0**	1	1	1	1	1
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	1
median	**1**	**1**	**1**	**1**	**1**	**1**	**1**

total distance: 3

	1	2	3	4	5	6	9	11
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0	0
$\mathcal{CS}(S_2[3,11])$	1	**0**	1	1	1	1	0	**1**
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	**1**	1	0
median	1	1	1	1	1	1	0	0

total distance: 4

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[3,7])$	1	**0**	1	1	**0**	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	**1**
median	1	1	1	1	1	1	0

total distance: 4

Distance threshold for median filter: $k\delta = 3 \Rightarrow$ three k-tuples are discarded.

Step 4: Computation of centers

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	0
$\mathcal{CS}(S_2[3,9])$	1	**0**	1	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	**1**
center	**1**	**1**	**1**	**1**	**1**	**1**	0

maximum pairwise distance: 2

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	**1**	1	1	1	0
$\mathcal{CS}(S_2[6,9])$	1	**0**	0	1	1	1	0
$\mathcal{CS}(S_3[2,8])$	1	1	0	1	1	1	**1**
center	**1**	**1**	0	**1**	**1**	**1**	0

maximum pairwise distance: 1

	1	2	3	4	5	6	9
$\mathcal{CS}(S_1[1,7])$	1	1	1	1	1	1	**0**
$\mathcal{CS}(S_2[2,9])$	1	**0**	1	1	1	1	1
$\mathcal{CS}(S_3[2,8])$	1	1	**0**	1	1	1	1
center	**1**	**1**	**1**	**1**	**1**	**1**	**1**

maximum pairwise distance: 1

Center gene clusters for $\delta = 1$: {1,2,4,5,6}, {1,2,3,4,5,6,9}

Figure 7.2: Extract from center gene cluster computation for $\delta = 1$ in $\mathcal{S} = \{S_1, S_2, S_3\}$ (as defined in the figure): The 4-step approach is performed for the character set $C' = \mathcal{CS}(S_1[1,7])$.

7.2 Computation of cluster filters (Step 1)

It follows from Observation 11 that center cluster filters are equivalent to pairwise distance constrained reference gene clusters for distance threshold $\delta_{cf} = 2\delta$. Hence, we can use Algorithm 8 for their computation. However, like with median cluster filters, it is needless to track the complete set of optimal δ_{cf}-locations. Therefore, we can simplify the procedure as was done in Algorithm 9 for the computation of median cluster filters. In fact, Algorithm 9 needs to be changed only at three points to compute center cluster filters instead of median cluster filters:

- First, we need to identify in line 15 intervals of the form $[l_x + 1, r_y - 1]$ for the parameter range $1 \leq x, y \leq 2\delta + 1$, instead of $1 \leq x, y \leq \delta + 1$. This is due to the changed cluster filter property as defined in Observation 11.
- Second, we need to change the test for the cluster filter property in line 21 from $\sum_{\ell'=1}^{k} \mathit{minDist}[\ell'] \leq \delta_{cf}$ to $\max_{1 \leq \ell' \leq k} \{\mathit{minDist}[\ell']\} \leq \delta_{cf}$. This is again due to the fact that we restrict now pairwise distances to the cluster filter.
- The third change is in the outermost **for**-loop (line 1) where we iterate through all reference intervals. As mentioned in the previous section, every element of a k-tuple that yields a center gene cluster has the properties of a center cluster filter. Therefore we can fix the reference interval, e.g. $S_\ell = S_1$, and omit the **for**-loop.

It follows from the previous considerations and the explanations given in Section 7.2 that the resulting algorithm computes all center cluster filters in the reference string. The time complexity of cluster filter detection decreases to $O(kn^2(\delta + 1)^2)$ as it is performed now only for one reference string.

7.3 Generation of k-tuples from maximal δ-locations of a cluster filter (Step 2)

To perform Step 2 of our approach, we need to compute for each cluster filter C' all k-tuples $([i_1, j_1], \ldots, [i_k, j_k])$ of maximal (2δ)-locations with pairwise intersecting character sets that have among them a perfect location of C'. Analogous to Step 2 of median gene cluster computation, we do this computation separately for each perfect location in the reference string. However now only for one designated reference string, not successively for all k strings.

Once a location of C' is fixed in the reference string, we identify at first its maximal (2δ)-locations in the other $k-1$ strings and store them in a (2δ)-loc table using Algorithm 10 of the previous chapter. Then, we enumerate the k-tuples using basically the same approach as in Section 6.3. There are only

two differences to consider. Since we use only one reference string, the filter against redundant generation of k-tuples with multiple perfect locations of C' becomes useless. Also the bound for the sum distance to the cluster filter does not pay off anymore. However, this is not really a loss: As we learned in Section 7.1, the equivalence to the median cluster filter threshold $2\frac{k-1}{k}\delta$ under the pairwise distance constraint is $2(k-1)\delta$, and intervals that could be combined to exceed this threshold are not even element of table (2δ)-loc.

The worst case time complexity of Step 2 is not affected by these changes: The generation of table (2δ)-loc takes time $O(kn^2)$ and the enumeration of all k-tuples takes time $O(n^{2k})$. Both analyses are for the worst case, which is unlikely to be achieved with real instances for gene cluster discovery.

7.4 Filtering k-tuples by median distances (Step 3)

Also, the third step of our approach is largely equivalent to the corresponding step in Chapter 6, only the objective is different now: While median computation used to be the final step to determine whether a k-tuple yields a median gene cluster, it is now a filter to reduce the number of instances for which the center needs to be computed. As we have seen before, a k-tuple can only have a center that complies with distance threshold δ if it has a median with sum distance at most $k\delta$. Therefore, we compute in this step the median distance of each k-tuple generated in Step 2 and compare it to $k\delta$. Only if this threshold is not exceeded, we pass the corresponding k-tuple on to Step 4 for center computation.

The median computation itself, including the iterative computation scheme, can be adopted one-to-one from Section 6.4. Therefore, the time complexity of this step is in $O(k|\Sigma_{\mathcal{C}}|)$ for each k-tuple generated in Step 2.

7.5 Computation of center gene clusters from remaining k-tuples (Step 4)

We now come to the final step of our approach, the determination of the centers of the remaining k-tuples to figure out which of these interval combinations yield a center gene cluster for the given distance threshold δ. As mentioned before, center computation is an NP-complete problem that is fixed parameter tractable. In this section, we review two alternative approaches to center computation. The fixed-parameter tractable algorithm by Ma and Sun [55] and an alternative approach developed by Léon Kuchenbecker for his bachelor thesis [48] which was co-supervised by myself in the scope of this PhD project.

7.5.1 Preliminary remarks

As the algorithms presented in this section do not directly solve the problem of center computation but the equivalent CLOSEST STRING Problem, respectively the more general NEIGHBOR STRING problem, we need to introduce some additional theory.

In the following, we represent a set of character sets $\mathcal{C} = \{C_1, \ldots, C_k\}$ via the corresponding bit strings $T_1, \ldots, T_k$ of length $m = |\Sigma_{\mathcal{C}}|$. These strings are defined such that in every string T_ℓ the *character state* of position p, i.e. $T_\ell[p]$, is set to one, if the pth character of $\Sigma_{\mathcal{C}}$ occurs in C_ℓ, and to zero otherwise. The symmetric set distance $D(C_\ell, C_{\ell'})$ between two character sets from $\mathcal{C}$ is then equivalent to the Hamming distance of the respective bit strings: $D_H(T_\ell, T_{\ell'}) = |\{p \mid T_\ell[p] \neq T_{\ell'}[p]\}|$. The formal definition of the CLOSEST STRING Problem (on general alphabets) is as follows:

Problem 17 *Given a set of strings $\mathcal{T} = \{T_1, \ldots, T_k\}$ of length m over an alphabet Σ, find a string T such that $\max_{\ell=1}^{k} D_H(T, T_\ell)$ is minimized.*

Clearly, for $\Sigma = \{0, 1\}$, CLOSEST STRING is equivalent to the problem of center computation. A related problem is NEIGHBOR STRING:

Problem 18 *Given a set of strings $\mathcal{T} = \{T_1, \ldots, T_k\}$ of length m over an alphabet Σ and distance thresholds $\delta_1, \ldots, \delta_k$, find a string T such that $D_H(T, T_\ell) \leq \delta_\ell$ holds for all $1 \leq \ell \leq k$.*

To simplify the description of the following algorithms, we introduce additional terminology: Given two bit strings T_ℓ and $T_{\ell'}$ of length m, we say a position p, $1 \leq p \leq m$, is a *match* between T_ℓ and $T_{\ell'}$ if $T_\ell[p] = T_{\ell'}[p]$ holds, otherwise it is a *mismatch*. Furthermore, we need the concept of *string partitions*: Every string T of length m can be partitioned into two smaller strings using a set $P = \{i_1, \ldots, i_{m'}\} \subseteq \{1, \ldots, m\}$. Let $Q = \{j_1, \ldots, j_{m''}\}$ be the complementary set of P, i.e. $Q = \{1, \ldots, m\} \setminus P$. We denote the subsequence of T defined by the index positions contained in P as $T|_P = T[i_1]T[i_2]\ldots T[i'_m]$. Analogously, we denote $T|_Q = T[j_1]T[j_2]\ldots T[j''_m]$. Clearly, it holds for any two strings T and T' of length m that $D_H(T, T') = D_H(T|_P, T'|_P) + D_H(T|_Q, T'|_Q)$.

7.5.2 The StringSearch Algorithm for center computation

Given an instance of the NEIGHBOR STRING problem $\big((T_1, \delta_1), \ldots, (T_k, \delta_k)\big)$, the StringSearch algorithm compares a fixed reference string T_1 to another string T_ℓ, $2 \leq \ell \leq k$ for which T_1 is not a neighbor string, i.e. $D_H(T_1, T_\ell) > \delta_\ell$, and determines all positions p where the two strings disagree, $T_1[p] \neq T_\ell[p]$. Let P be the set of these mismatch positions. The algorithm generates then all character state combinations a potential neighbor string T could have at these positions such that the distance thresholds δ_1 and δ_ℓ are not exceeded

for $T_1|_P$ and $T_\ell|_P$. For each possible combination, the algorithm is called recursively for the complementary string partition that corresponds to the yet undefined positions in the neighbor string, decreasing in each search branch the distance thresholds according to the mismatches between the k strings and the respective character state configuration in the recently fixed partition.

The recursion terminates successfully once there is no further T_ℓ for which $D_H(T, T_\ell) > \delta_\ell$ holds for the undetermined partition of T. In this case, the respective positions can be set to the character states of T_1. The resulting string is always a solution for the given instance of the NEIGHBOR STRING problem. In case no neighbor string exists, the algorithm terminates after all branches of the search tree have been tracked up to the point where the first remaining distance for a string partition becomes negative. An example of this procedure is given in Figure 7.3.

The key observation to prove the fixed-parameter tractability of this algorithm is that the remaining distance to T_1 is cut by half for each recursive call of StringSearch. (See [55] for details.) Based on this finding, it can then be shown that the time complexity of the algorithm is in $O(km + k\delta \cdot 2^{4\delta}(|\Sigma| - 1)^\delta)$, which reduces to $O(km + k\delta \cdot 2^{4\delta})$ for binary alphabets.

A way to solve the CLOSEST STRING Problem with this algorithm is to run it several times for increasing δ until the first neighbor string is found. To detect all centers, the algorithm needs to be modified such that it does not terminate after the first solution is found but works through the remaining branches of the search tree.

7.5.3 The MismatchCount Algorithm

The second algorithm for center computation studied in this thesis is the MismatchCount Algorithm introduced in [48]. We present in the following a simplified version of this approach that works directly on the binary strings (in contrast to the use of "transition matrices" in the original algorithm.)

Given a set of binary strings $\mathcal{T} = \{T_1, \ldots, T_k\}$ of length m and a distance threshold δ, the MismatchCount Algorithm solves the CLOSEST STRING Problem for binary strings using a simple enumeration scheme that tests all $\sum_{d=1}^{\delta} \binom{m}{d}$ strings T with distance at most δ to a selected element of $\mathcal{T}$. (Without loss of generality, we always choose T_1 in the following.) During the enumeration, the algorithm tracks the distances between the current T and each T_ℓ, $1 \leq \ell \leq k$, remembering the smallest maximum distance found so far along with the T for which it was detected. The enumeration scheme is chosen as follows: The T are enumerated separately for the different distance values to T_1. For each distance $d \in \{0, 1, \ldots, \delta\}$ the strings are generated recursively by fixing the character states in T from left to right, such that once the states of positions $1, \ldots, p$ are fixed containing $d' \leq d$ mismatches

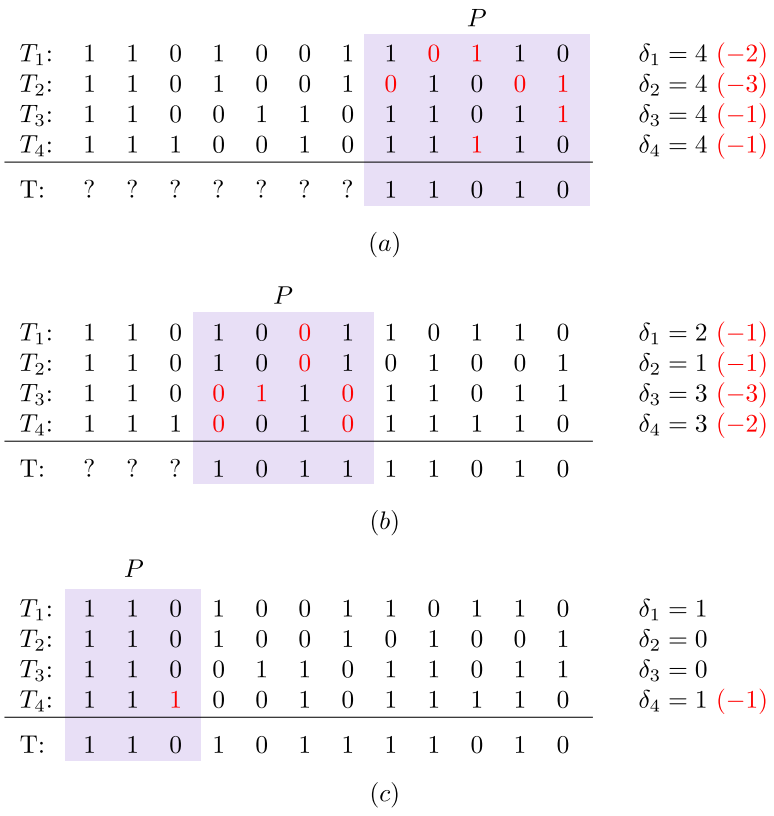


Figure 7.3: Search branch of the StringSearch Algorithm for neighbor string computation on the problem instance $((T_1, \delta_1), (T_2, \delta_2), (T_3, \delta_3), (T_4, \delta_4))$ as defined in the figure: (a) $D_H(T_1, T_2) > \delta_2$ holds, therefore T is partitioned based on the mismatches between T_1 and T_2 and a possible configuration of T is fixed for the mismatch partition P; (b) StringSearch is called recursively for the remaining positions. Since $D_H(T_1, T_3) > \delta_3$ holds, the partitioning of T is continued based on the mismatches between T_1 and T_3; (c) for the last string T_4, $D_H(T_1, T_4) \leq \delta_4$ holds. Therefore, the remaining positions of T are set to the respective character states of T_1.

$D_H(T,T_1)=0$	$D_H(T,T_1)=1$	$D_H(T,T_1)=2$	$D_H(T,T_1)=3$
0 0 0 0 0	1 0 0 0 0	1 1 0 0 0	1 1 1 0 0
	0 1 0 0 0	1 0 1 0 0	1 1 0 1 0
	0 0 1 0 0	1 0 0 1 0	1 1 0 0 1
	0 0 0 1 0	1 0 0 0 1	1 0 1 1 0
	0 0 0 0 1	0 1 1 0 0	1 0 1 0 1
		0 1 0 1 0	1 0 0 1 1
		0 1 0 0 1	0 1 1 1 0
		0 0 1 1 0	0 1 1 0 1
		0 0 1 0 1	0 1 0 1 1
		0 0 0 1 1	0 0 1 1 1

Figure 7.4: Enumeration scheme for all strings T with Hamming distance at most 3 to a bit string T_1 of length 5. The '0's denote matches between T and T_1 at the respective positions, while '1's denote mismatches.

compared to $T_1[1,p]$, the algorithm works through all possible character state combinations for the remaining string $T[p+1,m]$ with $d-d'$ mismatches compared to $T_1[p+1,m]$ before the character state of p is changed again. An example of such an enumeration scheme that sets the character state of every position — as long as possible — first to a mismatch compared to T_1 is visualized in Figure 7.4. After each change of T, we need to update the distance values to each $T_1,\ldots,T_k$. To this end, it is sufficient to consider only positions at which character states changed in relation to the previous T and compute for each $T_1,\ldots,T_k$ how the number of mismatches changed at these positions. Based on these considerations, the time complexity of this approach is in $O(kf)$, where f is the total number of character state changes during the enumeration of all possible configurations of T. For the presented enumeration scheme, it was conjectured in [48] that $f \in O(2^m)$ holds. We sketch a proof of this assertion: According to the enumeration scheme, every position p in T is set once to 0 and once to 1 for every possible character state configuration of $T[1,p-1]$ with up to δ mismatches to $T_1[1,p-1]$. There are $\binom{p-1}{d}$ such configurations for every $d=0,1,\ldots,\delta$. Summing over all possible values of p and d, we get the upper bound:

$$f \leq \sum_{d=0}^{\delta}\sum_{p=1}^{m} 2\binom{p-1}{d} = 2\sum_{p=1}^{m}\sum_{d=0}^{\delta}\binom{p-1}{d} \leq 2\sum_{p=1}^{m} 2^{p-1} \leq 2\cdot 2^m \in O(2^m).$$

Thus, the worst-case time complexity of the MismatchCount Algorithm is in $O(k2^m)$. To accomplish Step 4 of our approach to center gene cluster computation, we need to detect all centers of a k-tuple of intervals, and not only one representative. Therefore, we need to modify the MismatchCount Algorithm such that either all T with the current minimum distance are tracked (and discarded if a T with lower maximal distance is found), or run the algorithm twice, the first time to determine the smallest maximal distance to the center(s), and the second time to extract the closest strings.

What makes the MismatchCount Algorithm especially interesting are two optimizations that were shown to speed up the algorithm to an extent that its practical runtimes are lower than those of the StringSearch Algorithm on a broad range of random instances [48]. The first optimization is based on the following observation: Once a configuration of T is generated and the maximum pairwise distance $d_{max} = max_{1 \leq \ell \leq k} \{D(T, T_\ell)\}$ exceeds δ, at least $\lceil \frac{1}{2}(d_{max} - \delta) \rceil$ positions in T need to be changed before the maximum distance constraint can be met. Therefore, the enumeration of the following configurations can be skipped up to the point where $\lceil \frac{1}{2}(d_{max} - \delta) \rceil$ changes were made to the current T. In the recursive enumeration scheme, this corresponds to a backtracking to the point where the $\left(d - \lceil \frac{1}{2}(d_{max} - \delta) \rceil\right)$-th mismatch compared to T_1 was set. Respectively, if the backtracking involved more than d mismatches, the complete search branch for strings T with d mismatches to T_1 can be skipped.

The second optimization makes use of the fact that for positions p having the same character state in all strings, i.e. $T_1[p] = \ldots = T_k[p]$, it makes no sense to consider a string T with the opposite character state at position p for being closest string. This is because a closer string can always be constructed from such a string by inverting the character state of this position.

Despite its higher asymptotic time complexity, the MismatchCount Algorithm was chosen for the implementation of the presented approach to center gene cluster computation. This decision was based on the observation that its practical runtimes are lower on problem instances for center gene cluster computation compared to the StringSearch Algorithm [48].

7.6 Introduction of a quorum parameter

To detect also center gene clusters that occur only in a subset of the input genomes, we can integrate a quorum parameter q into our search strategy, as we did for the previous gene cluster models. The basic idea to accomplish this generalization is the same as for median gene clusters: During the recursive generation of interval combinations, we add up to $(k{-}q)$ empty intervals. For the basic enumeration, we would not need to modify any distance constraints to cope with empty intervals. However, as we perform an iterative median computation for filtering purposes, we need to subtract δ from the derived median distance threshold $k\delta$ for each empty interval added to a k-tuple. Also, we need to take care that the empty character sets of empty intervals are not used for median computation.

The main change due to the quorum parameter is in the cluster filter computation. Since a gene cluster can now have empty occurrences, it is no longer sufficient to consider only a single reference interval. Instead $(k{-}q{+}1)$ strings need to be processed as reference intervals. This modification ensures that all gene clusters covering at least q strings are detected. The cluster

filter threshold which constrains pairwise distances is not affected by the quorum parameter. However, the use of multiple reference strings brings us back the problem of redundant k-tuple generation for different locations of a cluster filter, which occurred initially with median gene clusters. Using the filters described in Section 6.3, the redundancy can be largely reduced.

Concerning the algorithmic complexity, only Step 1 of center gene cluster computation is affected in terms of worst case time complexity by the quorum parameter. As $(k-q+1)$ reference strings need to be processed now, the complexity increases to $O(k^2n^2(\delta+1)^2)$. For all other steps, the extra work is subsumed by the respective complexity classes. However, each of them is likely to be executed for more instances than without the quorum parameter.

7.7 Algorithm optimizations

Besides center computation, the enumeration of interval combinations is the second step in center gene cluster computation that is not polynomially bound. We show in the following that we can borrow most of the optimization techniques used in median gene cluster computation to speed up practical runtimes of this costly operation.

7.7.1 Minimum cluster filter size

Like with median gene clusters one can define a minimum size threshold on cluster filters. Clearly, only center cluster filters C' with $|C'| \geq s - \delta$ can yield a center gene cluster of size at least s. Therefore, we can skip Steps 2 to 4 for cluster filters that are smaller than that.

7.7.2 Infrequent characters in cluster filters

The trick to count infrequent characters in cluster filters does not work for center gene clusters. As a center is not determined by a majority vote, it may well contain infrequent characters.

7.7.3 Suboptimal interval borders

As already stated in Section 2.5, not every center gene cluster is necessarily a center of an interval combination that is optimized with respect to it. Unlike with median gene clusters, this is not even true if only intersecting center gene clusters are considered. However, one can easily verify that for every center gene cluster of this type, an equivalent or even better center gene cluster can be found for which the same or a very similar interval combination is optimal: Given an interval combination with a center gene cluster C it is not optimized for, we can replace every interval by the corresponding C-optimized interval and test if C is still center. If not there is another center

with lower maximum distance to the new intervals. In case the intervals are not optimized with respect to the new center, we replace them again by the corresponding optimized intervals and so on. Since after each replacement either the distance to the center decreases or a center-optimized interval combination is found, this process converges. Hence, for practical purposes, it should be sufficient to report only center gene clusters that are center of an interval combination, that is optimized with respect to it.

If we choose to do so, we can partially anticipate suboptimal interval borders in the same way we did for median gene clusters (Section 6.6.3). Of course, it is not possible to skip intervals with infrequent left-most or right-most essential characters, as they may be contained in centers. However, we can rule out intervals that can not be combined with a certain cluster filter location due to conflicting interval borders using the tests described in Section 6.6.3.

7.7.4 Lower bounds for sum distances to cluster filters

As we have seen in Section 7.3, it is not necessary during the enumeration of interval combinations to use an upper bound for the accumulated distances to the respective center cluster filter. This is because by construction no combination of intervals from the table (2δ)-loc can exceed the respective threshold $2(k-1)\delta$. Therefore, the anticipation of pairwise distances yet to be added to the sum, as described in Section 6.6.4, is of no use in the computation of center gene clusters.

7.7.5 Lower bounds for distances to prospective medians

On the contrary, the lower bounds for distances to prospective medians are useful for center gene cluster computation: When filtering center instances for having a median that complies with median distance threshold $k\delta$ (Step 3), these bounds can be applied as usual to stop branching as soon as a partly generated k-tuple can no longer yield a median that meets the distance constraint. We only need to adapt the bound to the corresponding median distance threshold which results in the following bound:

$$D(C^*, C) \quad \geq \max\{D(C^*, C') - \delta + 2f, f'\}\,.$$

(Recall that f is number of infrequent characters shared among C^* and C', while f' is the number of infrequent characters in C^*.)

7.7.6 Unfragmented entities

The last optimization technique presented in the previous chapter, the consideration of *unfragmented entities* can be adopted for center gene clusters. Clearly, if approximate cluster occurrences are no longer allowed to start/end

within such well-conserved segments, suboptimal center gene clusters will be ruled out as was already the case for median gene clusters. But practically no gene cluster is lost, as they are all contained in "better" gene clusters with more genes and longer approximate locations with the same number of missing or intermitting genes.

7.7.7 Upper bound on pairwise distances between intervals contained in table (2δ)-loc

An optimization that is only useful for center gene cluster computation consists of an upper bound on the pairwise distances between intervals that are combined in Step 2 to form k-tuples. By construction, these intervals are in a (2δ)-range of the center cluster filter. Therefore, their pairwise distance is bounded by 4δ. However, only for distances up to 2δ their combination may yield a center gene cluster for distance threshold δ. Hence, we can stop branching in the k-tuple enumeration once two intervals exceeding this upper bound are combined. To test this bound efficiently, the pairwise distances should be precomputed for all interval pairs.

Note that, in principle, a corresponding bound is applicable to median gene cluster computation, as the pairwise distances between intervals that have a median gene cluster are bounded by the sum distance constraint δ. However, each time this threshold is exceeded for a pair of intervals, also the median distance threshold is exceeded for the median of the partially filled k-tuple such that we gain no additional speed-up from this bound for median gene cluster computation .

Chapter 8

Statistical evaluation of gene cluster predictions

Up to now, we focused in this thesis on the development of efficient approaches to gene cluster computation under approximate common intervals based models. Clearly, in terms of practical applicability other issues apart from runtime efficiency need to be considered. In particular, we need to shift our attention to the question of whether gene clusters predicted by these approaches reflect indeed conserved ancestral gene order, or occur simply by chance. While the final decision on this point can of course not be done by purely computational means, we can use a statistical evaluation scheme to assess the likelihood of both explanations, or at least give a ranking of the predicted clusters to point out the most promising candidates for further experimental evaluation. Moreover, the knowledge on what type of gene clusters would be significant in a set of studied genomes can be used to rule out uninformative parameter ranges in the first place and thereby reduce the computational effort.

In the literature, many statistical models for gene cluster evaluation have been proposed. See [70] for a recent overview. These methods are typically designed for a special gene cluster model and are not generally applicable to other models including the ones presented in this thesis. Hence, we develop in this chapter a new approach that is suited for approximate common intervals based gene clusters.

We begin with an overview of the basic challenges involved in statistical gene cluster evaluation and a classification of the problem we are trying to solve (Section 8.1). Then we derive a new statistical framework for approximate common intervals based gene clusters in two steps. At first, we show how the significance of a δ-location can be computed (Section 8.2). Then we study the problem of approximate gene cluster significance (Section 8.3). We conclude the chapter with a critical reflection on the statistical power of the presented evaluation scheme (Section 8.4).

8.1 Basic challenges and related work

In comparative genomics, evidence of gene cluster conservation is typically explained as remnant ancestral gene order that was preserved up to present either by lack of divergence time or due to selective constraints. However, as gene order and gene content of related genomes diverge progressively over time, a third explanation becomes more and more important, namely that the seemingly conserved structures occur merely by chance. To rule out this possibility, it is necessary to test a predicted gene cluster against the null hypothesis of random gene order. Designing suitable test statistics for gene cluster significance is a challenging task, as many parameters need to be considered: the underlying gene cluster model, the cluster size, the rate of conservation, the size of the search space and the sizes of the involved gene families.

In their seminal work on gene cluster statistics, Durand and Sankoff [23] distinguish two basic statistical questions that occur in gene cluster evaluation:

- **Individual clusters**: Given a particular set of genes, is it significant to find them clustered in at least two genomic regions?
- **Whole genome clustering:** Given two genomes in which gene clusters have been predicted under a given cluster model, is the number of observed gene clusters significant?

Clearly, for our purpose, mainly the first question is of interest. Durand and Sankoff [23] also showed that this question should be answered in the context of the search strategy applied, as it determines the size of the search space considered for gene cluster detection. They distinguish the following three approaches:

- **Reference region**: Given a set of predefined reference genes, scan a genome for regions that contain all or a subset of these genes.
- **Window sampling**: Given two regions either on one or two different genomes, do they share a certain number of homologous genes?
- **Whole genome comparison**: Given two or more genomes, find all sets of genes that occur clustered in both genomes.

As our approximate common intervals based gene clusters are detected via the comparison of whole genomes, they clearly fall into the third category. Durand and Sankoff proposed a combinatorial framework for this and the other problem settings under the r-window model. However, as the approximate gene cluster occurrences in our models require no fixed interval length, we can not directly employ these test statistics to our problem. However, we will see in the following how it can be adapted to our purposes.

8.2 Significance of δ-locations

Before we deal with the problem of gene cluster significance, we study at first a related question: What is the probability that a genomic region is an approximate occurrence of a specific set of genes? Or stated more formally: Given a character set $C \subseteq \Sigma$, a distance threshold δ and an interval $[i, j]$ on a random permutation of a string S over an alphabet Σ, what is the probability that

$$D(\mathcal{CS}(S[i, j]), C) \leq \delta$$

holds? For simplicity, we assume that each character from the alphabet occurs in S with frequency 0 or 1. In the following, we set $\ell = j - i + 1$, $c = |C|$ and $h = |C \cap \mathcal{CS}(S[i, j])|$. We refer to h as the number of *hits*, as it denotes the number of characters from C that are contained in the interval. All other characters in $[i, j]$ are referred to as *non-hits*. Having excluded duplicate occurrences of genes, we get

$$D(\mathcal{CS}(S[i, j]), C) = c + \ell - 2h, \tag{8.1}$$

and we can reformulate our problem as follows: What is the probability that at least $h_{min} = \frac{1}{2}(c + \ell - \delta)$ hits from C occur in $[i, j]_S$? This is equivalent to a problem under the r-window gene cluster model studied by Durand and Sankoff [23]: What is the probability that h out of c' genes occur in a window of length ℓ, where $c' = |C \cap \mathcal{CS}(S)|$? The probability that exactly h genes fall into the window is given by the hypergeometric distribution. For the complete probability one only needs to sum over all possible values of h:

$$q(n, \delta, c', \ell) = \sum_{h=\frac{1}{2}(c+\ell-\delta)}^{\min(c',\ell)} \frac{\binom{c'}{h}\binom{n-c'}{\ell-h}}{\binom{n}{\ell}}. \tag{8.2}$$

This formula computes the probability that a randomly chosen interval from S constitutes a δ-location of C.

To assess the significance of a δ-location found through whole genome comparison, we need to include the search space size into our evaluation. Each of the $\Theta(n^2)$ intervals of S with sufficient length can be a δ-location of C. As these intervals are strongly overlapping, summing over the probabilities obtained for the individual intervals would systematically underestimate the significance of an observed δ-location. In the r-windows model, this problem is less prominent. With a fixed interval length of r, only $n - r + 1$ intervals are candidates for approximate cluster locations. We can achieve a similar situation for approximate common intervals based gene clusters if we do not rely on the probability that a specific interval $[i, j]$ is a δ-location, but on the probability that a certain position i is a starting point of a δ-location. One can easily verify that the length of a δ-location in S is bounded by $c' + \delta$. So we only need to compute the probability that an interval $[i, j]$ of length

$c' + \delta$ or any prefix interval $[i, j']$, $j' \leq j$, thereof is a δ-location of C. To obtain this value, we count the hit/non-hit configurations of an interval $[i, j]$ with h hits for which at least one $[i, j']$, $j' \leq j$, is a δ-location of C. The number of these configurations can be defined recursively as follows:

$$f(n, h, c', \ell) = \begin{cases} 0 & \text{if } \ell < h \text{ or } \ell = 0 \\ \binom{c'}{h}\binom{n-c'}{\ell-h} & \text{if } h \geq h_{min}[\ell] \\ f(n, h, c', \ell-1) + f(n, h-1, c', \ell-1) & \text{else} \end{cases} \tag{8.3}$$

The recursion contains two base cases. At first, we test if the current interval and therewith all its prefixes are too small to contain the required number of hits. If so, we return zero, as there is no valid configuration. The second base case occurs if the interval itself is a δ-location. Then, we count the number of ways to distribute h hits over ℓ positions. The third case applies if the interval itself is no δ-location but is large enough that one of its prefixes can possibly be a δ-location. Then, we set the state of the last position in the interval, once to a hit and once to a non-hit, and sum the valid configurations in the next smaller prefix over both cases. Assuming that the computation of the binomial coefficient is a constant time operation, this computation can be performed in time $O(\ell h)$ using a dynamic programming approach.

To obtain, from the number of valid configurations, the probability that a δ-location starts at a certain position, we only need to sum over all possible values of h and divide the result by the number of all possible configurations of the interval:

$$q(n, \delta, c, c') = \frac{\sum_{h=c-\delta}^{c'} f(n, h, c', c' + \delta)}{\binom{n}{c'+\delta}}. \tag{8.4}$$

The probability of finding at least one δ-location of C in S can then be bounded above by sampling all start positions in S that contain a gene from C:

$$P(n, \delta, c, c') \leq c' \cdot q(n, \delta, c, c'). \tag{8.5}$$

8.3 Significance of individual gene clusters

Based on the results of the previous section, we can now estimate the probability of observing a gene cluster in a set of strings with randomized gene order. At first, we study the pair-wise distance constraint, and then the sum distance constraint.

8.3.1 Significance under the pair-wise distance constraint

Assume we are given a set of k strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ with randomized gene order, a pair-wise distance threshold δ and a fixed set of genes C of size c. Then, the probability that C is a gene cluster in $\mathcal{S}$ can be obtained simply by multiplying over the probabilities of observing a δ-location in each string:

$$P_{pw}(k, \delta, c) = \prod_{k'=1}^{k} P(n_{k'}, \delta, c, c'_{k'}), \tag{8.6}$$

where n'_k is the length of $S_{k'}$ and $c'_{k'} = |C \cap \mathcal{CS}(S_{k'})|$.

Using a quorum parameter q, the situation becomes more involved. To obtain an exact probability, one needs to sum over all possible combinations of covered and non-covered strings in which at least q strings of $\mathcal{S}$ are covered. A simpler approach is to substitute the strings by a uniform string, that is defined such that the rejection of the null hypothesis of random gene order is the least likely, i.e. the string has to be designed such that the observation of an approximate location is the least likely. This is achieved by setting its length n to $\min\{n_1, \ldots, n_k\}$ and c' to $\min\{c'_1, \ldots, c'_k\}$. Then, the probability to find δ-locations is at least q out of k of these modified strings is:

$$P_{pw}(k, \delta, c, q) = \sum_{q'=q}^{k} \binom{k}{q'} P(n, \delta, c, c')^{q'} \big(1 - P(n, \delta, c, c')\big)^{k-q'}. \tag{8.7}$$

For the original strings this probability constitutes a lower bound.

8.3.2 Significance under the sum distance constraint

To obtain equivalent equations for the sum distance constraint, we need to sum over all possible distributions of the sum distance δ over the k strings. Unlike the pairwise distances above, the individual distances in these distributions are no upper bound but exact distances. Therefore, the equations derived for δ-locations can not be used in this evaluation. We need to adapt them to the probability of observing a δ-location that is no $(\delta-1)$-location. This probability, denoted as $q'(n, \delta, c, c')$, corresponds to the difference between the probability of observing a δ-location and the probability of observing a $(\delta-1)$-location:

$$q'(n, \delta, c, c') = \begin{cases} q(n, \delta, c, c') - q(n, \delta-1, c, c') & \text{if } \delta > 0 \\ q(n, \delta, c, c') & \text{if} \delta = 0 \end{cases} \tag{8.8}$$

The probability of observing at least one δ-location with exact distance δ in S can again be found by summing over all c' possible start positions:

$$P'(n, \delta, c, c') \leq c' \cdot q'(n, \delta, c, c'). \tag{8.9}$$

We have now all prerequisites to define the probability of observing a sum distance constrained approximate gene cluster. As mentioned above, we need to sum the probabilities over all possible distributions of δ over the k strings. This sum can be defined recursively:

$$P_{sum}(k, \delta, c) = \sum_{d=0}^{\delta} P'(n_k, d, c, c'_k) \cdot P_{sum}(k-1, \delta - d, c), \tag{8.10}$$

where $P_{sum}(0, \delta, c) = 1$ is the base case. For the integration of the quorum parameter into this calculation, we add a parameter that counts the number of covered strings:

$$\begin{aligned} P_{sum,q}(k, \delta, c, q) = &\sum_{d=0}^{\delta} P'(n_k, d, c, c'_k) \cdot P_{sum,q}(k-1, \delta-d, c, q-1) \\ &+ (1 - P(n_k, \delta, c, c'_k)) \cdot P_{sum,q}(k-1, \delta, c, q). \end{aligned} \tag{8.11}$$

To ensure that only combinations covering at least q strings contribute to the probability, we add a new base case, $P_{sum,q}(k, \delta, c, q) = 0$ if $q > k$. The second base case is $P_{sum,q}(0, \delta, c, q) = 1$ if $q \leq 0$.

8.3.3 Gene cluster significance in whole genome comparison

Up to now, we assumed that a gene cluster for which we determine the significance is a single pre-specified set of genes. However, this is not the typical scenario in gene cluster detection. Usually gene cluster predictions are obtained via a discovery, not a search strategy. Therefore, the number of sets of genes that were tested during the process influences the significance of observing a gene cluster with a certain size and degree of conservation. Unfortunately, we observe that candidate gene clusters, no matter whether they have a reference occurrence in the genomes or not, are largely intersecting. This makes it very hard to design a statistical framework that considers the search space size. Durand and Sankoff [23] proposed a solution for a reference based r-window approach where probability functions are corrected for overlapping reference intervals. For approximate common intervals based gene clusters, finding an efficient framework for whole genome comparison is more involved, if possible at all. This is because of the higher complexity of the model, the variable cluster size and, in case of median and center gene clusters, the fact that there is no reference occurrence required. Therefore, we pose as an open question whether such a framework can be found and how it would look like.

8.4 Discussion

We have designed a statistical framework to assess the significance of a pre-defined gene cluster. For gene clusters found through a discovery strategy, we have seen that the design of such a framework is a challenging task that would go beyond the scope of this thesis. If we use the statistical framework for pre-defined gene clusters to evaluate the predictions of our discovery algorithms, we will systematically overestimate their significance. However, to assess the relative significance of gene cluster predictions obtained in a single discovery process, the presented framework can still be useful — for example to obtain a ranking of the predictions that helps to identify the most promising candidates for further evaluation or to select the most significant representative among a possibly large number of redundant variants of a single gene cluster.

A problem that could arise in the first application is that a gene cluster consisting mainly of frequent genes can possibly get the same score as a gene cluster that consists mainly of non-duplicated genes. If these clusters have the same size and degree of conservation the observation of the second cluster is clearly more significant. However, as our method of genome randomization does not allow for gene families, this will not be reflected in the obtained ranking. Hence, the integration of gene families into our framework could improve the power of the evaluation scheme.

Apart from these technical difficulties, there is also a systematic problem connected to the evaluation of gene cluster significance. Usually the predictions are evaluated against the null-hypothesis of totally randomized gene order. However, we are not aware of any studies on the evolutionary distance that is necessary between species to assume random gene order. For closely related gene clusters, remnants of ancestral gene order may be a more likely explanation for some gene clusters than functional selection. Therefore, it might be interesting to incorporate such information in the evaluation of gene cluster significance.

Chapter 9

Experimental Results

We now come to the experimental evaluation of the presented approaches to gene cluster detection. The purpose of this chapter is two-fold: Our first goal is to evaluate the practical runtimes of our filter based approaches. Secondly, we will have a closer look at some of our gene cluster predictions to assess their quality from a biological point of view and thereby assess the utility of our approximate common intervals based gene cluster models. Unless stated otherwise, all computations described in this chapter were performed on an 8×2.6 GHz AMD Opteron 8218 Dual-Core processor with 32 GB main memory.

9.1 Data preparation

We used the genomes of five γ-proteobacteria for our experimental study which are summarized in Table 9.1. The datasets were downloaded from the NCBI database [69]. For grouping genes into homology families, we employed the GhostFam tool [79]. Using the standard parameters for sequence comparison, this program distributed the $11,184$ genes occurring in the studied genomes into 5086 gene families of sizes between 1 and 63. We discovered 36 unfragmented entities in the dataset with maximal length 10. To avoid that these segments are split up during computations, we flagged them in a preprocessing step. Computational results without this preprocessing can be found in Section 9.2.2.

9.2 Performance evaluation

For a general performance evaluation, we study, at first, the runtime of our algorithms for a broad range of parameter settings. Then we study the dependency of the runtime on the employed optimization techniques and finally compare the performance of these approaches to the ILP approach of Rahmann and Klau [71].

species name	refSeq number	# genes
Buchnera aphidicola APS	NC_002528	564
Escherichia coli K12	NC_000913	4183
Haemophilus influenzae Rd	NC_000907	1709
Pasteurella multocida Pm70	NC_002663	2015
Xylella fastidiosa 9a5c	NC_002488	2680

Table 9.1: Data sets used in the experimental evaluation.

9.2.1 Runtime dependency on parameter settings

To evaluate practical runtimes of our algorithms in dependency to cluster size and distance threshold, we conducted a series of test runs for different parameter settings for both median and center gene cluster detection.

Results for the median mode are given in Tables 9.2 for small gene cluster sizes and in Table 9.3 for larger size thresholds. For low distance thresholds, we observe that median gene cluster computation is quite fast. However, at a certain point that is dependent on the minimum cluster size, the combinatorial explosion kicks in which is caused by the recursion through the δ-loc tables. The number of possible k-tuples is an indicator of the hardness of this step. However, for a broad range of parameters, computation times for solving the general median gene cluster discovery problem are still manageable. For extreme parameter settings where runtimes for median gene cluster computation are prohibitive, practical results can be obtained under the reference based approximate gene cluster models. The time spent on the generation of δ-loc tables, as given in the tables, is an upper-bound of the computation time of the equivalent reference based gene cluster discovery problem.

We performed a similar series of experiments for center gene cluster computation. The results of these tests are summarized in Table 9.4. Here, we observe the same combinatorial explosion as described above, albeit it seems to occur later than for the median gene cluster approach if we adjust pairwise and sum distances. Consider, for example, the parameter setting $s = 6$ and $\delta_{pw} = 2$. Comparing 5 genomes, this corresponds to a sum distance threshold of $\delta_{sum} = 10$. For this setting median gene cluster computation took more than two hours, while center gene cluster computation finished within 32 seconds. However, as a substantial number of interval combinations with valid median have no center for the corresponding pairwise distance threshold, only a subset of the median gene clusters can be detected with a center based approach. For example, in the previous setting only 9 out of 17 median gene cluster classes are detected with the center approach.

Exemplarily, we studied also the impact of the quorum parameter on the computation times. Results for q-covering median gene cluster computation

	$\delta = 0$	$\delta = 1$	$\delta = 5$	$\delta = 8$	$\delta = 10$
$s = 4$					
running time	7s	7s	28s	1h 39m	-
time for generation of δ-loc	100%	100%	28%	$\ll 1\%$	(3m 45s)
# cluster filters	55	100	4107	8823	$1.7 \cdot 10^4$
# possible k-tuples	11	5464	$5.1 \cdot 10^9$	$3.0 \cdot 10^{11}$	$8.0 \cdot 10^{12}$
# completely generated k-tuples	11	65	$1.0 \cdot 10^7$	$1.6 \cdot 10^9$	-
# optimal k-tuples	11	33	1304	$1.1 \cdot 10^4$	-
# reference based k-tuples	11	33	1246	8574	-
# non nested k-tuples	6	13	404	3895	-
# gene cluster classes	6	7	36	43	-
$s = 5$					
running time	7s	7s	9s	1m 7s	35h 40m
time for generation of δ-loc	100%	100%	96%	14%	$\ll 1\%$
# cluster filters	35	59	1293	4366	$1.3 \cdot 10^4$
# possible k-tuples	7	1902	$6.3 \cdot 10^8$	$3.1 \cdot 10^{10}$	$3.8 \cdot 10^{12}$
# completely generated k-tuples	7	37	$5.0 \cdot 10^4$	$1.9 \cdot 10^7$	$2.0 \cdot 10^{10}$
# optimal k-tuples	7	18	478	3636	9889
# reference based k-tuples	7	18	445	2720	6006
# non nested k-tuples	5	10	129	748	1476
# gene cluster classes	5	5	13	25	26
$s = 6$					
running time	7s	7s	8s	13s	2h 14m
time for generation of δ-loc	100%	100%	98%	67%	$\ll 1\%$
# cluster filters	20	33	513	1518	8959
# possible k-tuples	4	492	$2.6 \cdot 10^8$	$8.8 \cdot 10^9$	$6.8 \cdot 10^{11}$
# completely generated k-tuples	4	21	$1.5 \cdot 10^4$	$5.6 \cdot 10^5$	$1.9 \cdot 10^9$
# optimal k-tuples	4	8	218	1347	3706
# reference based k-tuples	4	8	203	1071	2642
# non nested k-tuples	3	6	52	254	612
# gene cluster classes	3	3	6	17	17

Table 9.2: Experimental results for median gene cluster computation in five γ-proteobacteria. Computation times and results are shown for different combinations of s and δ. The fraction of the total running time spent on the generation of δ-loc tables corresponds approximately to the time needed for solving the equivalent reference based approximate gene cluster discovery problem. We also give information on intermediate results: the number of cluster filters, the number of k-tuples that can be generated from the δ-loc table (# possible k-tuples), the number of k-tuples actually generated during iterative median computation (# generated k-tuples), the number of k-tuples that are optimal with respect to a median gene cluster (# optimal k-tuples), the number of k-tuples for which the median has a reference occurrence (# reference based k-tuples), the number of median gene cluster k-tuples not nested within a better k-tuple (# non nested k-tuples) and the number of completely different, i.e. non-overlapping gene clusters (# gene cluster classes). Note: For the parameter combination $s = 4$ and $\delta = 10$ the computation was stopped unfinished after 48 hours, and only results for the generation of the δ-loc tables are given.

	$\delta = 8$	$\delta = 12$	$\delta = 18$	$\delta = 24$	$\delta = 32$
$s = 10$					
running time	9s	40s	6h 20m	-	-
time for generation of δ-loc	99%	25%	$\ll 1\%$	(8m 23s)	(88m 16s)
# cluster filters	155	684	9010	$2.3 \cdot 10^4$	$4.2 \cdot 10^4$
# possible k-tuples	$6.3 \cdot 10^8$	$4.8 \cdot 10^{10}$	$9.1 \cdot 10^{12}$	$2.3 \cdot 10^{15}$	$8.8 \cdot 10^{16}$
# generated k-tuples	8002	$2.1 \cdot 10^6$	$3.4 \cdot 10^9$	-	-
# optimal k-tuples	67	166	830	-	-
# reference based k-tuples	31	80	195	-	-
# non nested k-tuples	14	43	155	-	-
# gene cluster classes	2	3	4	-	-
$s = 15$					
running time	9s	10s	118s	3h 23m	-
time for generation of δ-loc	100%	99%	10%	$\ll 1\%$	(18m 25s)
# cluster filters	53	81	338	2838	$2.4 \cdot 10^4$
# possible k-tuples	$4.1 \cdot 10^5$	$1.0 \cdot 10^8$	$1.3 \cdot 10^{11}$	$1.0 \cdot 10^{13}$	$9.7 \cdot 10^{15}$
# generated k-tuples	1658	$1.9 \cdot 10^4$	$7.1 \cdot 10^6$	$7.1 \cdot 10^8$	-
# optimal k-tuples	15	21	51	80	-
# reference based k-tuples	9	15	24	29	-
# non nested k-tuples	5	5	5	5	-
# gene cluster classes	1	1	1	1	-
$s = 20$					
running time	9s	10s	14s	1m 24s	16h 35m
time for generation of δ-loc	100%	99%	87%	17%	$\ll 1\%$
# cluster filters	48	64	118	278	3472
# possible k-tuples	$4.0 \cdot 10^5$	$1.5 \cdot 10^7$	$6.5 \cdot 10^8$	$1.6 \cdot 10^{11}$	$6.5 \cdot 10^{13}$
# generated k-tuples	1614	$1.6 \cdot 10^4$	$2.5 \cdot 10^5$	$7.9 \cdot 10^6$	$3.6 \cdot 10^9$
# optimal k-tuples	12	15	39	50	72
# reference based k-tuples	6	9	18	23	27
# non nested k-tuples	3	3	3	3	3
# gene cluster classes	1	1	1	1	1

Table 9.3: Experimental results on median gene cluster computation in five bacterial genomes for larger size and distance thresholds. Description of the given data is equivalent to Table 9.2. For parameter combinations for which computation was unfinished after 24 hours, only results for the generation of δ-loc tables are shown.

	$\delta = 0$	$\delta = 1$	$\delta = 2$	$\delta = 3$	$\delta = 4$
$s = 4$					
running time	2s	4s	59m 16s	-	-
# cluster filters	11	219	2656	5522	-
# possible k-tuples	11	$2.3 \cdot 10^6$	$7.4 \cdot 10^9$	$2.1 \cdot 10^{11}$	-
# generated k-tuples	11	$5.5 \cdot 10^5$	$1.7 \cdot 10^9$	-	-
# distinct median k-tuples	11	1245	$1.2 \cdot 10^5$	-	-
# distinct center k-tuples	11	1180	$1.2 \cdot 10^5$	-	-
# non nested k-tuples	6	162	$2.6 \cdot 10^4$	-	-
# gene cluster classes	6	17	29	-	-
$s = 6$					
running time	2s	4s	32s	6h 0m	-
# cluster filters	4	48	326	2766	5536
# possible k-tuples	4	$5.3 \cdot 10^4$	$1.4 \cdot 10^8$	$5.9 \cdot 10^{10}$	$1.1 \cdot 10^{12}$
# generated k-tuples	4	486	$2.6 \cdot 10^7$	$1.4 \cdot 10^{10}$	-
# distinct median k-tuples	4	213	$1.8 \cdot 10^4$	$7.2 \cdot 10^5$	-
# distinct center k-tuples	4	201	$1.8 \cdot 10^4$	$6.6 \cdot 10^4$	-
# non nested k-tuples	3	61	856	$1.6 \cdot 10^4$	-
# gene cluster classes	3	4	9	13	-
$s = 8$					
running time	2s	3s	6s	12m 51s	-
# cluster filters	2	22	70	382	2857
# possible k-tuples	2	5051	$6.4 \cdot 10^6$	$1.6 \cdot 10^9$	$3.3 \cdot 10^{11}$
# generated k-tuples	2	96	$3.3 \cdot 10^4$	$3.2 \cdot 10^8$	-
# distinct median k-tuples	2	17	2582	$1.4 \cdot 10^5$	-
# distinct center k-tuples	2	17	2510	$1.3 \cdot 10^5$	-
# non nested k-tuples	2	16	181	2560	-
# gene cluster classes	2	2	3	3	-
$s = 10$					
running time	2s	3s	5s	14s	3h 21m
# cluster filters	1	10	35	84	427
# possible k-tuples	1	722	$8.5 \cdot 10^5$	$1.5 \cdot 10^8$	$1.2 \cdot 10^{10}$
# generated k-tuples	1	11	4482	$1.3 \cdot 10^6$	$2.3 \cdot 10^9$
# distinct median k-tuples	1	9	298	$1.5 \cdot 10^4$	$5.2 \cdot 10^5$
# distinct center k-tuples	1	9	296	$1.4 \cdot 10^4$	$4.5 \cdot 10^5$
# non nested k-tuples	1	9	71	1120	$1.2 \cdot 10^4$
# gene cluster classes	1	1	2	2	3

Table 9.4: Experimental results on center gene cluster computation in five bacterial genomes for different combinations of size and distance thresholds. Description of the given data is equivalent to Table 9.2. The additional distinction between median and center k-tuples is due to the fact that candidate interval combinations for center gene clusters are checked for fulfilling the corresponding median constraints. For parameter combinations for which computation was unfinished after 24 hours, only results for the generation of δ-loc tables are shown (except for the uninformative setting $s = 4$ and $\delta = 4$).

$s = 5$, $\delta = 8$	$q = 5$	$q = 4$	$q = 3$	$q = 2$
running time	1m 53s	19m 8s	21m 10s	21m 15s
# cluster filters	4371	5102	5102	5102
# possible k-tuples	$3.1 \cdot 10^{10}$	$9.5 \cdot 10^{11}$	$9.5 \cdot 10^{11}$	$9.5 \cdot 10^{11}$
# generated k-tuples	$2.3 \cdot 10^{7}$	$1.5 \cdot 10^{8}$	$1.7 \cdot 10^{8}$	$1.7 \cdot 10^{8}$
# distinct median k-tuples	3636	8532	9139	9139
# distinct center k-tuples	2720	5733	6165	6165
# non nested k-tuples	748	592	478	478
# gene cluster classes	25	33	36	36

Table 9.5: Experimental results for q-covering median gene cluster computation in five bacterial genomes for $s = 5$ and $\delta = 8$. Description of the given data is equivalent to Table 9.2.

with $s = 5$ and $\delta = 8$ are shown in Table 9.5. Employing a quorum parameter increases runtimes considerably. However it seems that the specific value of the quorum is of minor importance compared to the initial decision for or a against a quorum parameter. It should also be mentioned that a part of the increase in runtime is also caused by the incompatibility of some of our optimization techniques with the use of a quorum parameter which were switched off for the respective computations. Concerning the gene cluster predictions in presence of a quorum parameter, we observe that a substantial number of gene clusters seems to be conserved only in a subset of the input genomes.

9.2.2 Evaluation of optimization techniques

For a better understanding of the performance of the basic algorithms and their optimizations, we studied the runtime effects of the different optimization techniques. In Table 9.6 all optimizations and their applicability to the different algorithm variants are summarized.

At first, we study the performance on median gene cluster computation.

optimization techniques	median	center	quorum
minimum cluster filter size	+	+	+
infrequent characters in cluster filter	+		
suboptimal interval borders	+	(+)	+
lower bounds on sum distances	+	(+)	
unfragmented entities	+	+	+
pairwise distances in table δ-loc		+	+

Table 9.6: Combinability of the optimization techniques with the different algorithm variants. + indicates full compatibility, (+) indicates partial compatibility. See Sections 6.6 and 7.7 for details.

	runtime	speed-up
minimum cluster filter size	36 m 14 s	-
infrequent characters in cluster filter	35 m 12 s	2.8%
suboptimal interval borders	2 m 26 s	93.2%
lower bounds on sum distances	35 m 25 s	2.2%
unfragmented entities	7 m 21 s	79.7%
all but unfragmented entities	1 m 35 s	95.6%
all optimizations	28 s	98.7%

Table 9.7: Impact of the different optimization techniques on the runtime of our approach to median gene cluster computation for the parameter combination $s = 6$ and $\delta = 9$.

For simplicity, we use a fixed parameter setting for our tests. We chose the combination $s = 6$ and $\delta = 9$, as it involves a substantial amount of computation, while being still fast enough to run the program efficiently with almost no optimizations. An exception is the test for minimum cluster filter size. If this is omitted, almost every small interval in the given string is processed as cluster filter which inflates runtimes drastically. Therefore, we use an extended version of the algorithm that contains already this test as reference for the following evaluations. We added each of the remaining optimizations separately and computed the relative speed-up in percent. Additionally, we combined all optimizations except for the conservation of unfragmented entities, as it is the only speed-up that affects the gene cluster predictions — even though only in reducing redundancies. The results of the tests are summarized in Table 9.7.

For center gene cluster computation, it was difficult to find a parameter setting where computation times are feasible when our program is run without optimizations. Only for $\delta = 1$, we obtained results in less than 6 hours. Results for the parameter setting $s = 6$ and $\delta = 1$ are shown in Table 9.8. As in median gene cluster computation, the removal of intervals with suboptimal interval borders and the consideration of unfragmented entities yield the biggest reduction of practical runtimes. Also the center specific optimization that rules out interval combinations based on pairwise distances seems to be useful for runtime reduction.

9.2.3 Comparison to a related approach

The gene cluster detection approach that is most similar to ours are the weighted median gene clusters introduced by Rahmann and Klau [71]. For a comparison, we reproduced the study described in their work. For that purpose, we obtained the annotated genomes of *C. glutamicum* and *M. tuberculosis* from `http://gi.cebitec.uni-bielefeld.de/comet`. Gene families

	runtime	speed-up
minimum cluster filter size	23m 8s	-
suboptimal interval borders	4s	99.7%
lower bounds on sum distances	22m 25s	3.1 %
unfragmented entities	6s	99.5%
pairwise distances in table δ-loc	1m 19s	94.3%
all but unfragmented entities	4s	99.7%
all optimizations	3s	99.8%

Table 9.8: Impact of the different optimization techniques on the runtime of our approach to center gene cluster computation for the parameter combination $s = 6$ and $\delta = 1$.

were already established in this dataset. The tests described in this section were performed on a 1.66 GHz Intel Core Duo T2300 processor with 520 Mb of main memory running under the Suse Linux operating system.

Our program detects the gene cluster reported in [71] in 17 seconds using parameters $\delta = 27$ and $s = 60$ while the ILP using CPLEX 9.03 runs for more than one hour on a superior processor to compute this cluster, using size parameter $D = 51$.

We also conducted a similar series of experiments as reported in [71] to find optimal gene clusters for each size between 5 and 150. Since our method finds gene clusters based on a distance threshold and not for a certain size, we had to run our algorithm several times for different minimal cluster sizes and distance thresholds. Despite this overhead our method was able to find all optimal gene clusters in this size range within 3 hours and 4 minutes.

9.3 Selected results

To evaluate our predictions from a biological perspective, we inspected the gene annotations of the clusters detected in the dataset of Section 9.1. Most of our predictions refer to well known conserved structures, like the operons for ATP biosynthesis and histidine biosynthesis, the operon for cell wall synthesis and cell division, as well as a large ribosomal cluster consisting of 28 genes. In the following, we exemplarily show some of our predictions.

9.3.1 Gene cluster for ATP biosynthesis

The structure of the gene cluster for ATP biosynthesis as predicted by our approach is shown in Figure 9.1. We observe that the main part of this cluster is perfectly conserved in respect to both, gene content and gene order. This part of the cluster corresponds to the genes: atpC (377), atpD (29), atpG (378), atpA (29), atpH (379), atpF (380), atpE (576), atpB (381).

Figure 9.1: The structure of the gene cluster for ATP biosynthesis as predicted by our approach.

The duplication of gene 29 turns out to be an artifact of the sequence based homology assignment. According to the gene annotation, these are two different genes, namely atpD and atpA. The fuzzy border on the right consists of two genes coding for glucose inhibited division proteins (702, 382) and a predicted coding region (1540).

9.3.2 The cell division and cell wall biosynthesis gene cluster

Another well-known cluster that we detected with our approach is the cell division and cell wall biosynthesis gene cluster. Its structure as predicted by our approach is shown in Figure 9.2. The most remarkable feature of this prediction are the multiple occurrences of gene 9 in the cluster. However, an examination of gene annotations reveals that the four occurrences of gene 9 are in fact four different genes namely, murE, murF, murD and murC. Apparently, high sequence similarity caused their grouping into a single gene family.

A further observation in this gene cluster is that gene ftsL (1225) appears to be missing in the cluster occurrences of *B. aphidicola* and *X. fastidiosa*, while singleton genes are inserted at the respective positions in the two genomes. However, the gene annotations reveal that one of them is in fact a copy the ftsL gene, and the other is classified as general cell division protein. Therefore, the cluster seems to be better conserved in reality than the homology assignment suggests.

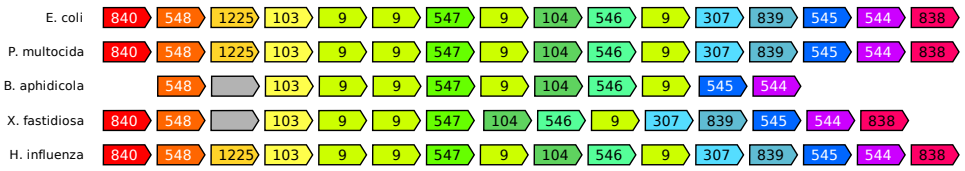


Figure 9.2: The structure of the cell division and cell wall biosynthesis gene cluster as predicted by our approach.

9.3.3 Gene cluster with changing reading direction

Another interesting cluster predicted by our approach is shown in Figure 9.3. It consists of two blocks whose order and orientation is conserved in three out of five genomes. In the other two, one block is either completely missing or reversed. This change in the reading direction indicates that the two blocks are not transcribed together and thus may, in fact, be two separate clusters. For the non-expert, the gene annotations give no hint at the cluster function: The contained genes code for miscellaneous products: a peptidyl-prolyl cis-trans isomerase, sulfur oxidation proteins, ribosomal proteins, elongation factors as well as some genes of unknown function.

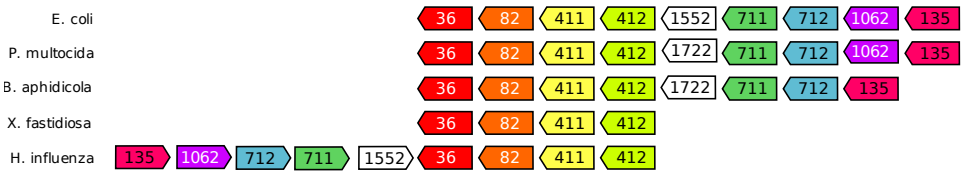


Figure 9.3: Predicted gene cluster with changing reading direction.

9.3.4 Gene cluster with changes in gene order

Except for the block interchange in the previous cluster, the gene order of all clusters studied so far is perfectly conserved in the compared genomes. Differences in the occurrences seem to be restricted to intermitting or missing gene occurrences. The picture is the same in most of the other gene cluster predictions in this dataset. However, there are a few exceptions. One is depicted in Figure 9.4, where we observe several rearrangement operations.

The gene annotations in this cluster suggest no obvious cluster function, as the gene functions seem to be rather diverse. They are summarized in Table 9.9. We leave as an open question, whether this gene cluster prediction constitutes a gene cluster in the biological sense, and if so, what functional role it may play in the studied organims.

Figure 9.4: Predicted gene cluster with rearranged gene order.

genome	gene id	annotation
E. coli	324	structural component; Ribosomal proteins - modification
	323	factor; Proteins - translation and modification (umk)
	322	enzyme; Central intermediary metabolism: interconversions (rrf)
	321	factor; Proteins - translation and modification (yaeM)
	320	1-deoxy-D-xylulose 5-phosphate reductoisomerase (yaeS)
	319	undecaprenyl pyrophosphate synthase
	861	enzyme; Fatty acid and phosphatidic acid biosynthesis (yaeL)
	657	zinc metallopeptidase (yzzY)
	101	conserved protein (ompH)
	1273	factor; Basic proteins - synthesis, modification
	62	UDP-3-O-(3-hydroxymyristoyl)-glucosamine N-acyltransferase
P. multocida	324	RpS2
	323	Tsf
	322	PyrH
	321	Errf
	320	Dxr
	319	unknown
	861	CdsA
	657	unknown
	101	unknown
	1273	Skp
	62	FirA
B. aphidicola	324	30S ribosomal protein S2 (tsf)
	323	elongation factor Ts (pyrH)
	322	uridylate kinase (frr)
	321	ribosome recycling factor (dxr)
	320	1-deoxy-D-xylulose 5-phosphate reductoisomerase (uppS)
	319	undecaprenyl pyrophosphate synthetase (yaeT)
	101	hypothetical protein (dnaE)
X. fastidiosa	321	ribosome recycling factor
	319	undecaprenyl pyrophosphate synthetase
	861	phosphatidate cytidylyltransferase
	320	1-deoxy-D-xylulose 5-phosphate reductoisomerase
	657	conserved hypothetical protein
	101	outer membrane antigen
	62	UDP-3-O-(R-3-hydroxymyristoyl)-glucosamine N-acyltransferase
H. influenzae	324	ribosomal protein S2 (rpS2)
	323	elongation factor Ts (tsf)
	62	UDP-3-O-(3-hydroxymyristoyl)-glucosamine (lpxD)
	1273	outer membrane protein, putative
	101	protective surface antigen D15
	657	conserved hypothetical transmembrane protein
	861	CDP-diglyceride synthetase (cdsA)
	319	conserved hypothetical protein

Table 9.9: Functional annotations of the genes occurring in the predicted gene cluster that is illustrated in Figure 9.4.

Chapter 10

Applications in phylogeny reconstruction

In the previous chapter, we have shown the potential of our method in detecting functionally related gene complexes. While this is the most obvious field of application, one can easily think of other problem settings in comparative genomics where approximate gene clusters may be useful, like the determination of positional homologs.

We explore a more distant field of application in this chapter, namely whole genome based phylogeny reconstruction. In the literature, two major lines of this field can be found: approaches that compute edit distances between genomes based on a set of rearrangement operations, e.g `GRAPPA` [56] and `MGR` [84], and feature based approaches that derive genomic distances from counting the number of structural features of a certain type, like gene content [80, 45, 37], gene adjacencies [45, 4, 10], or common/conserved intervals [10], that are shared between genomes. We focus on the latter approach and try to derive whole genome based distance measures based on the rate of conservation of approximate gene clusters between genomes.

The approach presented in this chapter can be viewed as a generalization of other feature-based methods: Phylogeny reconstruction based on gene content is a special case where only trivial gene clusters of size one are considered. The same is true for approaches based on (unsigned) adjacencies which are essentially gene clusters of size two. Clearly, common intervals are also a subset of approximate common intervals. (Only conserved intervals with their additional constraints can not be represented by our model.) Most of the above mentioned approaches are not applied to sequences of genes but on derived permutations, although a restriction to equal gene content is not forced by the underlying models. Hence, the novelty of the presented approach is two-fold: We use a more permissive gene cluster definition and omit the preprocessing step to transform sequences of genes into permutations. The intention behind the first generalization is to make our distance

measures more robust against gene losses and gene insertions in conserved segments, as well as against errors in the genome data, while the idea behind using sequences is simply to use the complete information contained in gene order.

10.1 Distance Measures

Based on the assumption that two species are the closer related the more structural features are conserved between their genomes, we use the following basic distance measure for two strings S and T:

$$\text{dist}(S,T) = 1 - \frac{1}{2}\left(\frac{\text{cons}(S,T)}{\text{cons}(S,S)} + \frac{\text{cons}(T,S)}{\text{cons}(T,T)}\right), \tag{10.1}$$

where the term $\text{cons}(S,T)$ stands for the number of features of a certain type present in S that can also be found in T. In normalizing this value with $\text{cons}(S,S)$, the number of features of the given kind in S, we ensure that $0 \leq \frac{\text{cons}(S,T)}{\text{cons}(S,S)} \leq 1$. Since the conservation ratio is in general not symmetric, we build the average over both ratios so that the overall distance lies between 0 and 1. A similar measure can be found in [10].

However, when using features like common intervals that grow quadratically in the string length, the value of $\text{cons}(S,T)$ can become disproportionally small compared to a self-comparison $\text{cons}(S,S)$. As a consequence, the distances between non-identical genomes will cluster at the top-end of the distance range. To account for this imbalance, the general distance formula can be modified as follows:

$$\text{dist}(S,T) = 1 - \frac{1}{2}\left(\sqrt{\frac{\text{cons}(S,T)}{\text{cons}(S,S)}} + \sqrt{\frac{\text{cons}(T,S)}{\text{cons}(T,T)}}\right) \tag{10.2}$$

The above definitions allow for a wide range of structural features whose conservation can be represented by $\text{cons}(S,T)$. The most basic ones among them are *gene content* and *unsigned adjacencies*:

- $\text{GC}(S,T) := |\{i \mid \text{there is a } j \text{ with } \mathcal{CS}(S[i,i]) = \mathcal{CS}(T[j,j])\}|$,
- $\text{Adj}(S,T) := |\{i \mid \text{there is a } j \text{ with } \mathcal{CS}(S[i,i+1]) = \mathcal{CS}(T[j,j+1])\}|$.

Based on these definitions, it becomes evident that both features are special cases of common intervals. A conserved gene is a common interval of size one, and a conserved adjacency is a common interval of size two. Note that the number of both structural elements is limited by the length of S. There are at most $|S|$ conserved genes and at most $|S|-1$ conserved adjacencies ($|S|$ in circular chromosomes).

A natural extension is to consider larger segments of genes conserved between species, i.e. *common intervals*, and even segments that are only partially conserved, i.e. *approximate common intervals*:

- $\mathrm{CI}(S,T) := |\,\{[i,j]_S \mid \mathcal{CS}(S[i,j]) \text{ has a location in } T\}\,|$,
- $\mathrm{CI}(S,T,\delta) := |\,\{[i,j]_S \mid \mathcal{CS}(S[i,j]) \text{ has a } \delta\text{-location in } T\}\,|$.

Apparently, $\mathrm{CI}(S,T)$ equals $\mathrm{CI}(S,T,\delta)$ for $\delta = 0$. Since there are $\binom{n}{2}$ non-empty substrings of a sequence of length n, the number of substrings of S with a δ-location in T is also restricted by $\binom{n}{2}$. To represent the degree of conservation of a genome segment, not only the gene content but also the conservation of the gene order should be considered. This is done implicitly in the above definitions, as for every conserved interval the following relation holds: the better its gene order is conserved, the more subintervals are conserved and thus contribute to $\mathrm{cons}(S,T)$.

Apart from this, all intervals count equally in the previous definitions independent of their size and their degree of conservation. To account for such differences, we can weight the interval counts by length, by the degree of conservation or by a combination of both attributes. This is realized in the following definitions where the term $\mathrm{CI}(S,T,d,\ell)$ denotes the number of intervals in S with length ℓ that have a d-location in T but no $(d–1)$-location:

- $\mathrm{CI}_{\mathrm{size}}(S,T,\delta) := \sum_{d=0}^{\delta} \sum_{\ell=d+1}^{|S|} \ell \cdot \mathrm{CI}(S,T,d,\ell)$
- $\mathrm{CI}_{\mathrm{deg}}(S,T,\delta) := \sum_{d=0}^{\delta} \sum_{\ell=d+1}^{|S|} \frac{\ell-d}{\ell} \cdot \mathrm{CI}(S,T,d,\ell)$
- $\mathrm{CI}_{\mathrm{size,deg}}(S,T,\delta) := \sum_{d=0}^{\delta} \sum_{\ell=d+1}^{|S|} (\ell-d) \cdot \mathrm{CI}(S,T,d,\ell)$.

As the conserved intervals are weighted by their lengths, the maximum values of the terms $\mathrm{CI}_{\mathrm{size}}(S,T,\delta)$ and $\mathrm{CI}_{\mathrm{size,deg}}(S,T,\delta)$ are bounded by the sum $\sum_{\ell=1}^{n} \ell \cdot ((n+1)-\ell) = \frac{1}{6} \cdot n \cdot (n+1) \cdot (n+2)$ while the value of $\mathrm{CI}_{\mathrm{deg}}(S,T,\delta)$ is bounded by $\binom{n}{2}$.

10.2 Experimental Results

To evaluate the distance measures proposed in this study, we applied them to a benchmark dataset for whole genome based phylogeny reconstruction and compared our results to the trees obtained by Blin et al. on the basis of common intervals in permutations [10]. The dataset comprises the genomes of 12 γ-proteobacteria, summarized in Table 10.1.

abbreviation	species name	RefSeq	#genes
BAPHI	*Buchnera aphidicola APS*	NC_002528	564
ECOLI	*Escherichia coli K12*	NC_000913	4183
HAEIN	*Haemophilus influenzae Rd*	NC_000907	1709
PAERU	*Pseudomonas aeruginosa PA01*	NC_002516	5540
PMULT	*Pasteurella multocida Pm70*	NC_002663	2015
SALTY	*Salmonella typhimurium LT2*	NC_003197	4203
WGLOS	*Wigglesworthia glossinidia brevipalpis*	NC_004344	653
XAXON	*Xanthomonas axonopodis pv. citri 306*	NC_003919	4192
XCAMP	*Xanthomonas campestris*	NC_003902	4029
XFAST	*Xylella fastidiosa 9a5c*	NC_002488	2680
YPEST-CO92	*Yersinia pestis CO_92*	NC_003143	3599
YPEST-KIM	*Yersinia pestis KIM5 P12*	NC_004088	3879

Table 10.1: Overview of the benchmark dataset of twelve γ-proteobacteria used for phylogeny reconstruction.

We took the preprocessed genome sequences from [10] and computed pairwise distances for all genome combinations using both distance formulas, the four approximate common intervals based measures and a broad range of distance thresholds: $\delta = 0, 1, 5, 10, 20$. In total, we derived 40 distance matrices for this dataset which we processed by the tree reconstruction program `fitch` of the PHYLIP package [25], which is based on the *Fitch-Margoliash* approach [27]. For all runs of the program, we used options `J`, causing a randomization of the input order, and `G`, allowing for global branch-swapping. For assessing the predicted trees, we compared them to the putative reference tree for this dataset (see Figure 10.1(b)). This tree was generated from concatenated protein data using a neighbor-joining approach [51]. To measure the distance between the trees, we used the `treedist` command of the PHYLIP package which computes a tree distance based on the Robinson-Foulds metric [73].

Table 10.2 shows the Robinson-Foulds distances between the predicted trees and the reference tree. In many cases, the predicted phylogeny is very close to the reference tree or even identical with it. In cases where the Robinson-Foulds distance equals 2 or 4, the distance is due to a misplacement of the two endosymbionts *Buchnera aphidicola* and *Wigglesworthia glossinidia brevipalpis* as shown in Figure 10.2, whose placement in the reference tree is discussed controversially in the literature [33, 51].

Furthermore, we can conclude from Table 10.2 that the second distance formula employing the square root clearly outperforms the plain distance formula on this dataset. Apparently, taking the square root of the ratio $\frac{\text{cons}(S,T)}{\text{cons}(S,S)}$ partly compensates for the relatively low conservation rate. Sur-

	Distance Formula (10.1)					Distance Formula (10.2)				
$\delta =$	0	1	5	10	20	0	1	5	10	20
CI	4	4	2	0	0	2	2	2	2	0
CI_{size}	12	8	6	6	6	8	6	0	0	0
CI_{deg}	4	4	4	2	0	2	2	2	2	2
$CI_{size,deg}$	12	8	6	6	6	8	6	0	0	0

Table 10.2: Robinson-Foulds distances between predicted trees and the putative reference tree for all four approximate common intervals based distance measures and a collection of distance thresholds δ.

prisingly, it seems to be counterproductive to weight the conserved clusters by their length. However, this could be an artifact of our method as the weighting increases the discrepancy between $\text{cons}(S, T)$ and $\text{cons}(S, S)$ further. Taking also approximate gene clusters into account appears to be beneficial as the distance to the reference tree decreases in all cases with growing δ.

An example for the dependency of the reconstruction accuracy and δ is depicted in Figure 10.2. The results were achieved with the Distance Formula (10.1) using the term $\text{CI}(S, T, \delta)$ for counting approximate common intervals. One can see that the topology of the tree conforms better and better with the reference tree for increasing values of δ until identity is reached for $\delta \geq 10$.

This example of our results is also the one suited best for a comparison with the method presented in [10], as both were obtained with comparable distance formulas for phylogeny reconstruction based on common intervals. Recall that there are still two essential differences between these two approaches: our distance formula is defined on sequences while the one used by Blin et al. is defined on permutations, and we count not only common intervals but also approximate common intervals. When comparing the two approaches on the test dataset, we observe the following: While *Pseudomonas aeruginosa* is placed into a wrong clade in the trees predicted by Blin et al. [10] (see Figure 10.1(a)), its location in the trees constructed by our method for $\text{CI}(S, T, \delta)$ is congruent with the reference tree even for $\delta = 0$. This indicates that working with sequences instead of permutations alone can already be beneficial for accurate tree reconstruction. However, for small δ, our method has the same difficulties in placing *Buchnera aphidicola* and *Wigglesworthia glossinidia brevipalpis* into the right clade, as was also reported by Blin et al. Interestingly, this problem vanishes for larger values of δ which supports our conjecture that using approximate common intervals should increase the robustness of common intervals based approaches to phylogeny reconstruction.

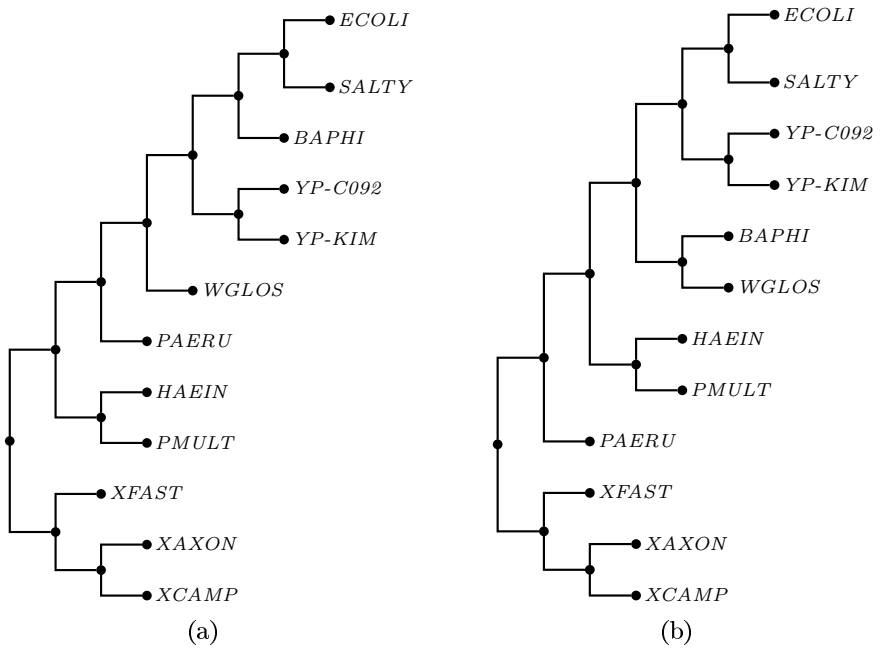


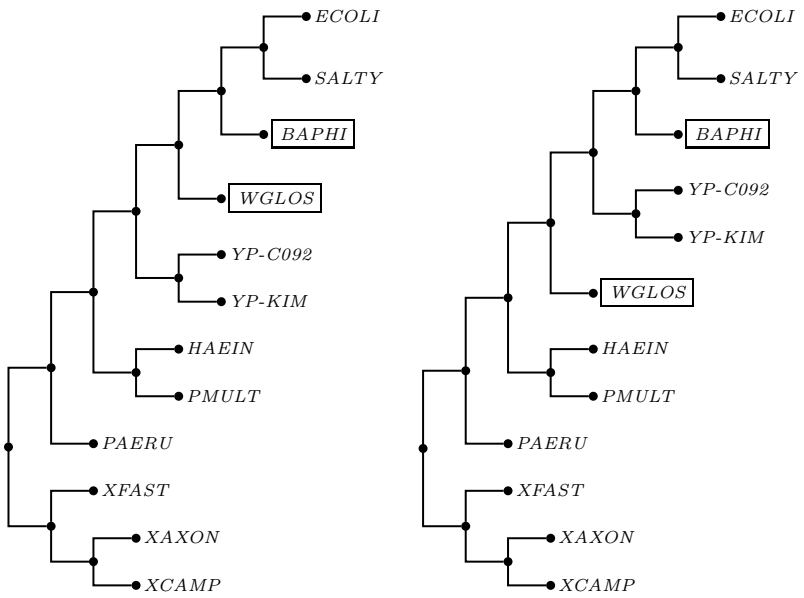
Figure 10.1: Tree topologies used for comparison with our predictions: (a) tree predicted by Blin et al. based on common intervals in permutations [10], (b) putative reference tree obtained by Lerat et al. [51] (taken from [10]).

10.3 Concluding remarks

In this chapter, we introduced a family of new whole-genome distance measures that are based on approximate gene cluster conservation. Our method is directly applicable to gene sequences, such that the artificial reduction of the gene sequences to permutations often used in phylogeny reconstruction is not necessary. The second novelty of our approach is that not only perfectly conserved gene clusters are considered but also clusters occurring with a slightly different gene content in the compared species. Initial experimental results suggest that both model extensions are beneficial for phylogenetic tree reconstruction. A comparison with a previous approach [10] supports this conjecture as well.

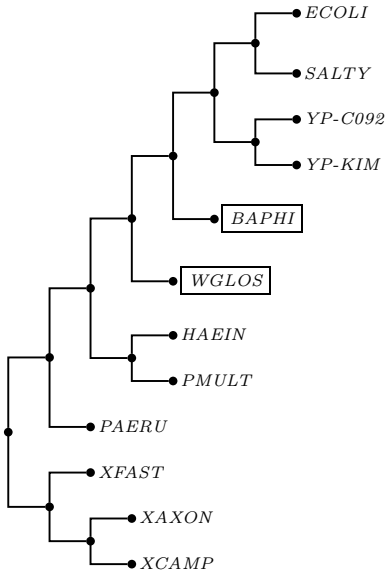
While the suggested distance measures yield promising results, some questions remain open: For example, an explanation for the rather counter-intuitive observation that it seems to be unfavorable to consider the length of the conserved blocks is missing. This could either be an artifact of our distance measures, that would not occur for methods using a more sophisticated way to deal with low interval conservation rates, or it reflects the situation that the length of an interval is already sufficiently represented by the counts of its subintervals.

Another point of further research is to study in more detail the effect of

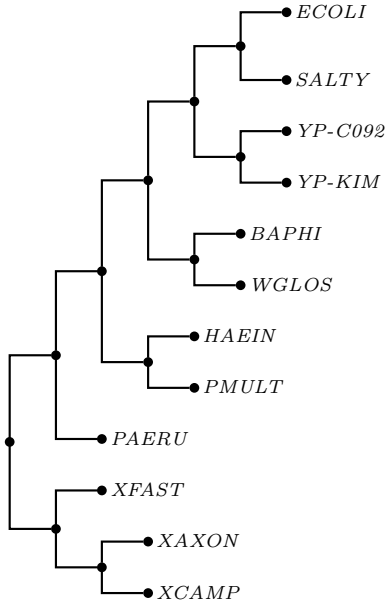


(a) $\delta = 0$

(b) $\delta = 1$



(c) $\delta = 5$



(d) $\delta = 10$

Figure 10.2: Phylogenetic trees predicted with Distance Formula (10.1) using $\mathrm{CI}(S, T, \delta)$ to measure conservedness of approximate common intervals for different values of δ. The boxed species are misplaced with respect to the reference tree. Tree (d) is congruent with the reference tree.

the distance threshold on the prediction quality. There should be an optimal value such that prediction accuracy worsens when increasing the threshold beyond this point.

A more general issue is the question how approximate common intervals conservation correlates with the divergence times of the compared species. Most likely, a saturation level is reached after some time such that further genome rearrangement operations, including changes of the gene content, have only a marginal effect on the conservation rates. As a first step to answer these questions, one could study the impact of the different rearrangement operations, including changes of the gene content, on the conservation rates of approximate common intervals. For a more thorough analysis, a complete model of whole genome evolution would be needed. However, this process is not yet completely understood.

Chapter 11

Conclusion and perspectives

In this thesis, we studied novel approaches to the detection of approximate gene clusters in multiple genomes. We formalized the concept of approximate common intervals and defined thereon four gene cluster models that are based on the symmetric set distance to measure the degree of conservation between gene clusters and their approximate occurrences. The presented models differ in two ways: the distance constraining mode and the gene cluster concept. Distances are either constrained individually for each approximate occurrence or via their sum, while the two cluster concepts either allow any combination of genes to form a gene cluster, or only those sets having a perfect reference occurrence in at least one of the studied genomes.

We have shown that the latter type of gene clusters can be computed efficiently using an extension of the Connecting Intervals Algorithm for common intervals computation. During our study of the original version of this algorithm, we discovered a simple modification that reduces its space complexity from quadratic to linear dependence on the genome lengths. We believe that the underlying trick is also applicable to the computation of approximate gene cluster occurrences. However, due to the higher intricacy of the extended algorithm, the realization will be technically more involved. To improve the scalability of our approaches to the comparison of a larger number of genomes, it will be useful to elaborate this improvement in the future.

For our general approximate gene cluster discovery problems, we had to face a search space that grows exponentially with the number of compared genomes. Here, we managed to design a filter scheme that sufficiently narrows down the search space to compute approximate gene clusters in multiple genomes. Additional optimization techniques substantially reduced the practical runtimes and extended the range of feasible parameter settings. Further work in this direction may become essential as typical gene cluster discovery scenarios develop towards the comparison of ever larger numbers of genomes.

This development makes only sense if also complete losses of gene clusters are allowed in a subset of the compared genomes. Otherwise, the detections will be limited to a small number of core gene clusters. In this thesis, we described extensions of our gene cluster models that cover missing gene cluster occurrences and showed how the respective algorithms can be adapted to these models. The asymptotic time complexities were not affected by these extensions. However, we have seen that the practical runtimes can increase substantially for the general approximate gene cluster discovery problems. As this is partly due to the fact that some of the optimization techniques are in their current form not combinable with missing gene cluster occurrences, the situation can be probably improved by designing optimization techniques that are customized to these model variants.

Our experimental results showed that our approaches are capable of detecting biologically meaningful gene clusters that are only approximately conserved in the studied genomes. Manual inspection of gene annotations suggests that most of the predictions correspond to well-known gene clusters, but for some no obvious functional classification can be derived from the gene annotations. The natural next step in this direction is a deeper analysis of the predictions in cooperation with biologists and a large-scale application to genomic datasets. Here, the selection of newly sequenced genomes may increase the chances to detect novel gene clusters and/or novel gene functions.

The experimental results gained up to now allow for a first assessment of the proposed gene cluster models. We frequently observed both missing and intermitting genes in approximate gene cluster occurrences. These findings support the basic idea of our set distance based gene cluster models, that allowing only gaps and no deletions in approximate cluster occurrences as seen in max-gap clusters systematically underestimates the degree of gene cluster conservation.

Moreover, we observed that for larger distance thresholds the optimal consensus gene set of a combination of gene cluster occurrences has often no perfect occurrence in any of the studied genomes. This seems to suggest that the additional effort in solving the general approximate gene cluster discovery problem pays off in terms of prediction completeness. However, one has to be aware that the same approximate gene cluster occurrences can be found under the reference based gene cluster model simply by relaxing the distance thresholds as necessary to account for the increased distances to a close gene set instead of an optimal consensus. Therefore, if the target is only to detect genome segments with similar gene content, it may be completely sufficient to use a reference based gene cluster model.

Concerning the use of a sum or pairwise distance constraint, our current results are inconclusive. Both approaches have their advantages and disadvantages: While the sum distance constraint is a suitable way to cope with different degrees of gene cluster conservation, it leads to undesired ef-

fects when comparing large numbers of genomes. To allow for the same degree of overall gene cluster conservation, the distance threshold needs to grow with the number of compared genomes. Already for a small amount of genomes, we observed the effect that most cluster occurrences are smaller than the distance threshold. This can increase computation times as single gene occurrences need to be considered as candidate cluster occurrences and a substantial number of them is even reported as approximate cluster occurrence if the degree of conservation is sufficient in the other genomes. Such single gene cluster occurrences are not informative especially if the gene in question has several occurrences in the genome. A simple remedy against this undesired effect could be an additional constraint that defines the minimum size of a cluster occurrence. For the pairwise distance constraint, we observe the completely opposite situation. It automatically rules out badly conserved gene cluster occurrences, but has in turn trouble to detect gene clusters with different degrees of conservation in the studied genomes. For practical purposes it might be interesting to define a hybrid version between the two distance constraining modes that limits both the pairwise and the sum distance.

Concerning the general approximate gene cluster discovery problem another interesting question is whether the median or the center is a better representation of an approximate gene cluster. Intuitively, one might assume that consensus gene sets should have an evolutionary interpretation as the putative gene cluster composition in the most recent common ancestor. However, one can easily see that this is unlikely in general. Assuming that the degree of gene cluster conservation between genomes corresponds to their evolutionary divergence time, we have to see that the median depends strongly on the evolutionary distances of the chosen genomes. If these are not well distributed over the phylogenetic tree, the state of the median is dominated by the best represented clade and represents — if at all — a gene cluster occurrence of an ancestor of this clade. For a center based consensus set, a phylogenetic interpretation seems to be more convincing at first glance, as the center has approximately the same distance to each of the studied genomes. However, also the composition of the center can be distorted by certain conservation patterns. For example, a single badly conserved gene cluster occurrence with intermitting genes can cause a center based consensus to include some of these intermitting genes to minimize the maximum pairwise distance. So, evolutionary interpretability is not a quality of our consensus gene sets and is therefore no useful criterion for their comparison.

While we have shown that our set distance based gene cluster models improve over existing approaches, we believe it is currently not possible to choose "the best one" among these models. In practice, it may be useful to run gene cluster predictions under several models and compare the results. Alternatively, one can simply pick the model that best matches the intention

of the current gene cluster search and the computational resources available.

However, our findings give rise to a number of possible refinements that are interesting for all four approximate common-intervals based gene cluster models. To provide a better reflection of the biological reality in our genome representations, we should extend our models and computational approaches towards circular and multi-chromosomal genomes. Such an extension is straight-forward: For gene cluster detection in circular chromosomes, we simply need to allow for gene clusters that overrun the current string boundaries and continue at the other end of the chromosome. For multi-chromosomal genomes, we need to adapt the search strategy such that all chromosomes of a genome are considered for containing cluster occurrences. This can be simply realized by iterating over all chromosomes of a genome whenever it is processed in the course of the algorithm. The latter extension is already included in our implementations.

Another enhancement might be the use of a more sophisticated cost function for missing and intermitting genes in cluster occurrences. So far, we use the same cost for both types of changes, gene insertions and gene losses. One could argue that intermitting genes, especially if they are on the opposite strand, have less severe effects on the functionality of a gene cluster than relocations or losses of genes from the consensus gene set. Therefore, it might be useful to decrease the cost of such events. It is, however, unclear how such a weighting should look like in detail.

Besides the weighting, another modification of our set distance measure should be considered. The biological idea behind the symmetric set distance is that the content of gene cluster occurrences changes by gene losses and the insertion of genes. However, differences in the gene content can as well be explained by gene transformations, i.e. the nucleotide sequences of homologous genes may diverge to the point that their evolutionary relationship is no longer detected when gene families are constructed. This separation may be wanted, as also the function of the genes may have diverged sufficiently, or unwanted, in case of erroneous homology assignment. In any case, such a gene transformation is counted as two events, the loss and the gain of a gene. To count these transformations as single events, the symmetric set distance could be replaced by the set transformation distance:

$$D(C, C') \;=\; \max\left\{|C \setminus C'|, |C' \setminus C|\right\}.$$

Also the concept of homology assignment in general could be reconsidered. As we have seen, current approaches based on sequence similarity have difficulties to restore the evolutionary relationships between genes, which causes gene clusters to appear less conserved than they actually are. Improvements in the homology assignment may reduce this problem, but perfect homology assignment is unlikely to be achieved in the near future. Therefore, we propose a rather radical approach that completely avoids homology

assignment by using gene similarities instead of gene equality. This means that we assign every pair of gene occurrences a similarity score. In doing so, we get a continuous instead of a binary relation between gene occurrences. We could then measure the degree of gene cluster conservation based on the overall gene similarity in the cluster occurrences.

Another radical change of our gene cluster models would be the integration of phylogenetic information into our concept of consensus gene clusters. In doing so, it may be possible to overcome the shortcomings of center and median gene cluster representatives as described above. One approach in this direction would be the reconstruction of ancestral gene cluster compositions based on a parsimony approach. The genes assigned to each inner node in the tree could then be interpreted as a consensus gene cluster of the respective subtree.

Apart from these model refinements, future work on approximate common intervals based gene cluster models could also go in the direction of exploring other fields of application than gene cluster prediction. Our models and algorithmic approaches are not limited to strings of gene family ids. Any kind of sequences can be processed to detect approximate common intervals based conservation patterns. A promising candidate for such studies are sequences of transcription factor binding sites, as conservation patterns of such elements are used to predict regulatory regions in genomes [43]. Also applications in non-biological sequences are imaginable, like the detection of text passages with similar word content

Based on the results achieved in this thesis, we belief that further studies on approximate common intervals based gene cluster models and the numerous extensions described above are a worthwhile endeavor. With the increasing availability of completely sequenced genomes, the demand for efficient and flexible approaches to gene cluster prediction is likely to grow in the near future and new algorithmic challenges will arise.

Bibliography

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, 1990.

[2] A. Amir, L. Gasieniec, and R. Shalom. Improved approximate common interval. *Inf. Process. Lett.*, 103(4):142–149, 2007.

[3] A.J. Enright and. S.Van Dongen and C.A. Ouzounis. An efficient algorithm for large-scale detection of protein families. *Nucleic Acids Res.*, 30(7):1575–1584, 2002.

[4] E. Belda, A. Moya, and F.J. Silva. Genome rearrangement distances and gene order phylogeny in γ-proteobacteria. *Molecular biology and evolution*, 22(6):1456–1467, 2005.

[5] A. Bergeron, C. Chauve, F. de Mongolfier, and M. Raffinot. Computing common intervals of k permutations, with applications to modular decomposition of graphs. In *Proceedings of ESA 2005*, volume 3669 of *LNCS*, pages 779–790, 2005.

[6] A. Bergeron, C. Chauve, and Y. Gingras. *Bioinformatics Algorithms: Techniques and Applications*, chapter Formal models of gene clusters, page 177. Wiley Book Series on Bioinformatics. Wiley-Interscience, 2008.

[7] A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. In *Proceedings of WABI 2002*, volume 2452 of *LNCS*, pages 464–476, 2002.

[8] A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. In *Proceedings of COCOON 2003*, volume 2697 of *LNCS*, pages 68–79, 2003.

[9] J. Besemer and M. Borodovsky. GeneMark: web software for gene finding in prokaryotes, eukaryotes and viruses. *Nucleic Acids Res.*, 33(Web Server Issue):W451, 2005.

[10] G. Blin, C. Chauve, and G. Fertin. Genes order and phylogenetic reconstruction: Application to γ-proteobacteria. In *Proceedings of RECOMB-CG 2005*, volume 3678 of *LNBI*, pages 11–20, 2005.

[11] G. Blin and J. Stoye. Finding nested common intervals efficiently. In *Proceedings of RECOMB-CG 2009*, volume 5817 of *LNCS*, pages 59–69, 2009.

[12] S. Böcker, K. Jahn, J. Mixtacki, and J. Stoye. Computation of median gene clusters. *J. Comp. Biol.*, 16(8):1085–1099, 2009.

[13] G. Bourque and L. Zhang. Models and methods in comparative genomics. *Advances in Computers: Computational Biology and Bioinformatics*, 68:59, 2006.

[14] C. Burge and S. Karlin. Prediction of complete gene structures in human genomic DNA. *J. Mol. Biol.*, 268:78–94, 1997.

[15] C. Chauve, Y. Diekmann, S. Heber, J. Mixtacki, S. Rahmann, and J. Stoye. On common intervals with errors. Report 2006-02, Technische Fakultät der Universität Bielefeld, Abteilung Informationstechnik, 2006.

[16] F. Chen, A.J. Mackey, C.J. Stoeckert Jr, and D.S. Roos. OrthoMCL-DB: querying a comprehensive multi-species collection of ortholog groups. *Nucleic Acids Res.*, 34(Database Issue):D363, 2006.

[17] X. Chen, Z. Su, P. Dam, B. Palenik, Y. Xu, and T. Jiang. Operon prediction by comparative genomics: an application to the synechococcus sp. wh8102 genome. *Nucleic Acids Res.*, 32(7):2147–2157, 2004.

[18] J.C. Chiu, E.K. Lee, M.G. Egan, I.N. Sarkar, G.M. Coruzzi, and R. DeSalle. OrthologID: automation of genome-scale ortholog identification within a parsimony framework. *Bioinformatics*, 22(6):699, 2006.

[19] A. L. Delcher, D. Harmon, S. Kasif, O. White, and S. L. Salzberg. Improved microbial gene identification with Glimmer. *Nucleic Acids Res.*, 27(23):4636–4641, 1999.

[20] G. Didier. Common intervals of two sequences. In *Proceedings WABI 2003*, volume 2812 of *LNBI*, pages 17–24, 2003.

[21] G. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. *J. Discr. Alg.*, 5:330–340, 2007.

[22] T. Dobzhansky and A. H. Sturtevant. Inversions in the chromosomes of *Drosophila Pseudoobscura*. *Genetics*, 23:28–64, 1938.

[23] D. Durand and D. Sankoff. Tests for gene clustering. In *Proceedings of RECOMB 2002*, pages 144–154. ACM Press, 2002.

[24] M. D. Ermolaeva, O. White, and S. L. Salzberg. Prediction of operons in microbial genomes. *Nucleic Acids Res.*, 29(5):1216–1221, 2001.

[25] J. Felsenstein. PHYLIP - phylogeny inference package (version 3.2). *Cladistics*, 5:164–166, 1989.

[26] W.M. Fitch. Homology a personal view on some of the problems. *Trends in genetics*, 16(5):227–231, 2000.

[27] W.M. Fitch and E. Margoliash. Construction of phylogenetic trees. *Science*, 155:279–284, 1967.

[28] M. Frances and A. Litman. On covering problems of codes. *Theor. Comput. Sci.*, 30:133–119, 1997.

[29] J. Gramm, R. Niedermeier, and P. Rossmanith. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, 37(1):25–42, 2003.

[30] X. He and M. H. Goldwasser. Identifying conserved gene clusters in the presence of homology families. *J. Comp. Biol.*, 12:638–656, 2005.

[31] S. Heber and J. Stoye. Algorithms for finding gene clusters. In O. Gascuel and B. Moret, editors, *Proceedings WABI 2001*, volume 2149 of *LNCS*, pages 252–263. Springer Verlag, 2001.

[32] S. Heber and J. Stoye. Finding all common intervals of k permutations. In A. Amir and G. Landau, editors, *Proceedings of CPM 2001*, volume 2089 of *LNCS*, pages 207–218. Springer Verlag, 2001.

[33] J.T. Herbeck, P.H. Degnan, and J.J. Wernegreen. Nonhomogeneous model of sequence evolution indicates independent origins of primary endosymbionts within the enterobacteriales (γ-proteobacteria). *Mol. Biol. Evol.*, 22(3):520–32, 2005.

[34] R. Hoberman and D. Durand. The incompatible desiderata of gene cluster properties. In *Proceedings of the RECOMB-CG 2005*, volume 3678 of *LNBI*. Springer Verlag, 2005.

[35] K. Homma, S. Fukuchi, Y. Nakamura, T. Gojobori, and K. Nishikawa. Gene Cluster Analysis Method Identifies Horizontally Transferred Genes with High Reliability and Indicates that They Provide the Main Mechanism of Operon Gain in 8 Species of γ-Proteobacteria. *Molecular biology and evolution*, 24(3):805, 2007.

[36] Franziska Hufsky, Léon Kuchenbecker, Katharina Jahn, Jens Stoye, and Sebastian Böcker. Swiftly computing center strings. *BMC Bioinformatics*, 12:106, 2011.

[37] D.H. Huson and M. Steel. Phylogenetic trees based on gene content. *Bioinformatics*, 20(13):2044, 2004.

[38] M.A. Huynen and P. Bork. Measuring genome evolution. *Proceedings of the National Academy of Sciences*, 95(11):5849, 1998.

[39] M.A. Huynen and B. Snel. Gene and context: integrative approaches to genome analysis. *Advances in Protein Chemistry*, 54:345–379, 2000.

[40] K. Jahn. Efficient computation of approximate gene clusters based on reference occurrences. In *RECOMB-CG*, volume 6398, pages 264–277. 2011.

[41] K. Jahn and J. Stoye. Approximative Gencluster und ihre Anwendung in der komparativen Genomik. *Informatik Spektrum*, 32(4):288–300, 2009.

[42] L.J. Jensen, P. Julien, M. Kuhn, C. von Mering, J. Muller, T. Doerks, and P. Bork. eggNOG: automated construction and annotation of orthologous groups of genes. *Nucleic Acids Res.*, 36(Database issue):D250, 2008.

[43] H. Klein and M. Vingron. Using transcription factor binding site cooccurrence to predict regulatory regions. *Genome Informatics*, 18:109–118, 2007.

[44] A.B. Kolsto. Dynamic bacterial genome organization. *Molecular Microbiology*, 24(2):241–248, 1997.

[45] J.O. Korbel, B. Snel, M.A. Huynen, and P. Bork. SHOT: a web server for the construction of genome phylogenies. *Trends in Genetics*, 18(3):158–162, 2002.

[46] I. Korf. Gene finding in novel genomes. *BMC Bioinformatics*, 5(59), 2004.

[47] A. Krause, J. Stoye, and M. Vingron. Large scale hierarchical clustering of protein sequences. *BMC Bioinformatics*, 6(15), 2005.

[48] L. Kuchenbecker. *Consensus Gene Cluster Computation based on* Closest String. Bachelor thesis, Bielefeld University, Bielefeld, 2008.

[49] J. K. Lanctot, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. *Inf. Comput.*, 185(1):41–55, 2003.

[50] J. Lawrence. Selfish operons: The evolutionary impact of gene clustering in prokaryotes and eukaryotes. *Curr. Opin. Genet. Dev.*, 9:642–648, 1999.

[51] E. Lerat, V. Daubin, and N. A. Moran. From gene trees to organismal phylogeny in prokaryotes: The case of the γ-proteobacteria. *PLoS Biology*, 1:101–109, 2003.

[52] Z. Li, L. Wang, and K. Zhang. Algorithmic approaches for genome rearrangement: a review. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(5):636–648, 2006.

[53] X. Ling, X. He, and D. Xin. Detecting gene clusters under evolutionary constraint in a large number of genomes. *Bioinformatics*, 25(5):571, 2009.

[54] K. Liolios, I.M.M. Chen, K. Mavromatis, N. Tavernarakis, P. Hugenholtz, V. M. Markowitz, and N. C. Kyrpides. The genomes on line database (gold) in 2009: status of genomic and metagenomic projects and their associated metadata. *Nucleic Acids Res.*, 38(Database issue):gkp848+, January 2010.

[55] B. Ma and X. Sun. More efficient algorithms for closest string and substring problems. In *Proceedings of RECOMB 2008*, pages 396–409, 2008.

[56] B.M.E. Moret, S. Wyman, D.A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proc. 6th Pacific Symp. Biocomputing PSB 2001*, pages 583–594, 2001.

[57] J. H. Nadeau and B. A. Taylor. Lengths of chromosomal segments conserved since divergence of man and mouse. *Proc. Natl. Acad. Sci. USA*, 81:814–818, 1984.

[58] R. Overbeek, M. Fonstein, M. D'Souza, G. D. Pusch, and N. Maltsev. The use of gene clusters to infer functional coupling. *Proc. Natl. Acad. Sci. USA*, 96(6):2896–2901, 1999.

[59] A. Paccanaro, J. A. Casbon, and M. A. S. Saqi. Spectral clustering of protein sequences. *Nucleic Acids Res.*, 34(5):1571–1580, 2006.

[60] L. Parida. Gapped permutation patterns for comparative genomics. In *Proceedings of WABI 2006*, volume 4175 of *LNCS*, pages 376–387, 2006.

[61] G. Parra, E. Blanco, and R. Guigó. `GeneID` in *Drosophila*. *Genome Res.*, 10:511–515, 2000.

[62] S. Pasek, A. Bergeron, J.L. Risler, A. Louis, E. Ollivier, and M. Raffinot. Identification of genomic features using microsyntenies of domains: domain teams. *Genome Res.*, 15(6):867, 2005.

[63] G. Pavesi, G. Mauri, F. Iannelli, C. Gissi, and G. Pesole. GeneSyn: a tool for detecting conserved gene order across genomes. *Bioinformatics*, 20(9):1472–1474, 2004.

[64] S. Penel, A.M. Arigon, J.F. Dufayard, A.S. Sertier, V. Daubin, L. Duret, M. Gouy, and G. Perrière. Databases of homologous gene families for comparative genomics. *BMC Bioinformatics*, 10(Suppl 6):S3, 2009.

[65] P. Pevzner and G. Tesler. Genome rearrangements in mammalian evolution: lessons from human and mouse genomes. *Genome Res.*, 13(1):37, 2003.

[66] M. Pignatelli, F. Serras, A. Moya, R. Guigo, and M. Corominas. CROC: finding chromosomal clusters in eukaryotic genomes. *Bioinformatics*, 25(12):1552, 2009.

[67] P. Pipenbacher, A. Schliep, S. Schneckener, A. Schonhuth, D. Schomburg, and R. Schrader. ProClust: improved clustering of protein sequences with an extended graph-based approach. *Bioinformatics*, 18:182–191, 2002.

[68] M.N. Price, K.H. Huang, E.J. Alm, and A.P. Arkin. A novel method for accurate operon predictions in all sequenced prokaryotes. *Nucleic Acids Res.*, 33(3):880, 2005.

[69] K.D. Pruitt, T. Tatusova, and D.R. Maglott. NCBI reference sequences (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic Acids Res.*, 35(Database issue).

[70] N. Raghupathy and D. Durand. Gene cluster statistics with gene families. *Molecular Biology and Evolution*, 26(5):957, 2009.

[71] S. Rahmann and G. W. Klau. Integer linear programs for discovering approximate gene clusters. In *Proceedings of WABI 2006*, volume 4175 of *LNBI*, pages 298–309, 2006.

[72] V. Ranwez, F. Delsuc, S. Ranwez, K. Belkhir, M.K. Tilak, and E.J.P. Douzery. OrthoMaM: A database of orthologous genomic markers for placental mammal phylogenetics. *BMC Evolutionary Biology*, 7(1):241, 2007.

[73] D.R. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53:131–147, 1982.

[74] I. B. Rogozin, K. S. Makarova, Y. I. Wolf, and E. V. Koonin. Computational approaches for the analysis of gene neighborhoods in prokaryotic genomes. *Briefings in Bioinformatics*, 5(2):131–149, 2004.

[75] I. B. Rogozin, K. S. Makarova, Y. I. Wolf, and E. V. Koonin. Computational approaches for the analysis of gene neighbourhoods in prokaryotic genomes. *Briefings in Bioinformatics*, 5(2):131–149, 2004.

[76] S. L. Salzberg, A. L. Delcher, S. Kasif, and O. White. Microbial gene identification using interpolated Markov models. *Nucleic Acids Res.*, 26(2):544–548, 1998.

[77] D. Sankoff. Edit distance for genome comparison based on non-local operations. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proceedings of CPM 1992*, volume 644 of *LNCS*, pages 121–135. Springer Verlag, 1992.

[78] T. Schmidt and J. Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. In *Proceedings of CPM 2004*, volume 3109 of *LNCS*, pages 347–358, 2004.

[79] T. Schmidt and J. Stoye. Gecko and GhostFam – rigorous and efficient gene cluster detection in prokaryotic genomes. In N. Bergman, editor, *Comparative Genomics*, volume 2 of *Methods in Molecular Biology*, chapter 12, pages 165–182. Humana Press, 2007.

[80] B. Snel, P. Bork, and M.A. Huynen. Genome phylogeny based on gene content. *Nature genetics*, 21:108–110, 1999.

[81] J. Tamames, G. Casari, C. Ouzounis, and A. Valencia. Conserved clusters of functionally related genes in two bacterial genomes. *J. Mol. Evol.*, 44(1):66–73, 1997.

[82] J. Tamames et al. Evolution of gene order conservation in prokaryotes. *Genome Biol*, 2(6):1–0020, 2001.

[83] R.L. Tatusov, N.D. Fedorova, J.D. Jackson, A.R. Jacobs, B. Kiryutin, E.V. Koonin, D.M. Krylov, R. Mazumder, S.L. Mekhedov, A.N. Nikolskaya, et al. The COG database: an updated version includes eukaryotes. *BMC Bioinformatics*, 4(1):41, 2003.

[84] G. Tesler. GRIMM: Genome rearrangements web server. *Bioinformatics*, 18(3):492–493, 2002.

[85] T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26(2):290–309, 2000.

[86] Y. I. Wolf, I. B. Rogozin, A. S. Kondraskov, and E. V. Koonin. Genome alignment, evolution of procaryotic genome organization, and prediction of gene function using genomic context. *Genome Res.*, 11:356–372, 2001.

[87] I. Yanai, J.C. Mellor, and C. DeLisi. Identifying functional links between genes using conserved chromosomal proximity. *Trends in Genetics*, 18(4):176–179, 2002.

[88] Q. Yang and S.H. Sze. Large-scale analysis of gene clustering in bacteria. *Genome Res.*, 18(6):949, 2008.

[89] G. Yona, N. Linial, and M. Linial. ProtoMap: automatic classification of protein sequences and hierarchy of protein families. *Nucleic Acids Res.*, 28(1):49, 2000.

[90] Q. Zhu, Z. Adam, V. Choi, and D Sankoff. Generalized gene adjacencies, graph bandwidth and clusters in yeast evolution. *IEEE/ACM Trans. Comput. Biol. Bioinf.*, 6:213–220, 2009.

Appendix A

Incompleteness of the ACI Algorithm

As already mentioned in Chapter 4, an alternative approach to the detection of δ-locations can be found in the literature, namely the ACI Algorithm by Amir et al. [2]. However, one can show that this graph-based approach misses a certain subclass of δ-locations. In the following, we sketch this algorithm as far as necessary to understand the incompleteness problem, present an example for which the algorithm fails to report the complete solution set and categorize the subclass of δ-locations that are detected by the approach. Unfortunately, there is no apparent remedy for this problem in the ACI Algorithm, at least none that would not substantially increase the algorithmic complexity.

A.1 Search strategy of the ACI Algorithm

For simplicity, we review the ACI Algorithm using the terminology and notation of the main part of this thesis. The problem that Amir et al. intend to solve in [2] is as follows:

Problem 19 *Given a set of k strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ over an alphabet Σ and a distance threshold δ, report for each character set $C = \mathcal{CS}(S_\ell[i_\ell, j_\ell])$ all character sets $C' = \mathcal{CS}(S_{\ell'}[i_{\ell'}, j_{\ell'}])$ with $D(C, C') \leq \delta$ and all locations of C'.*

The suggested ACI Algorithm works in two steps: First, a graph $G = (V, E)$ is built such that

- the set of distinct character sets $C = \mathcal{CS}(S_\ell[i_\ell, j_\ell])$ in $\mathcal{S}$ maps to the vertex set $V = \{v_C \mid C = \mathcal{CS}(S_\ell[i_\ell, j_\ell])\}$, and
- edges are drawn between vertices v_C, $v_{C'}$ if and only if $D(C, C') = 1$, labeled with the unique character in $(C \setminus C') \cup (C' \setminus C)$.

By construction, this graph represents all pairs of character sets in $\mathcal{S}$ whose pairwise distance is exactly one. An example of such a graph is given in Figure A.1. For the second step of the algorithm, recall from graph theory that a path in a graph $G = (V, E)$ is a sequence of vertices $v_0, v_1, \ldots, v_m$ such that there is an edge (v_i, v_{i+1}), $0 \leq i < m$, between every pair of successive vertices, and the length of the path corresponds to the number of edges in the path. This concept is used in the ACI Algorithm to collect δ-locations from the graph: Beginning from every vertex $v_C \in V$, all paths of length δ are traversed in a depth-first manner as long as no edge label is read that occurred earlier in the path.

A.2 Incompleteness of the algorithm

Clearly, every vertex on the traversed paths corresponds to the character set of a δ-location of C. However, not all such character sets are detected by this method, as we can see in Figure A.1: The shortest path between the two gray-shaded vertices has length four, although the distance between the corresponding character sets is only two. Therefore, the algorithm will not report the locations of the respective character sets as δ-locations of each other.

We find the same type of problem for all pairs of character sets for which at least one intermediate character set is not represented in the graph. This is no exceptional event, as it can happen with all intervals that share some characters but are not nested. Moreover, we observe that every path between the two gray-shaded vertices contains redundant edge labels, a second reason why the algorithm would not detect the connection between the two character sets.

A simple remedy against the incompleteness of the algorithm is to allow for both, longer paths and redundant edge labels. However, these changes have a negative effect on the asymptotic time complexity. In the original algorithm, the complete cost for the depth-first traversal of paths can be hidden in the output size, as all vertices on these paths are reported as δ-locations of the character set associated with the start vertex. If we allowed for longer paths to cope with missing intermediate character sets, this is no longer possible.

We leave as an open question, whether a more sophisticated way of modifying the algorithm can be found that preserves its time complexity. For the ACI Algorithm in its current state, we can summarize our findings as follows:

Observation 13 *The depth-first search of the ACI Algorithm detects for a given character set $C \subseteq \Sigma$ only those character sets C' of δ-locations for which there is a sequence of character sets $C_0, C_1, \ldots, C_d$, with $C_0 = C$, $C_d = C'$, $d \leq \delta$, and $|C_i \setminus C_{i+j}| + |C_{i+j} \setminus C_i| = j$ for all $j \leq \delta - i$.*

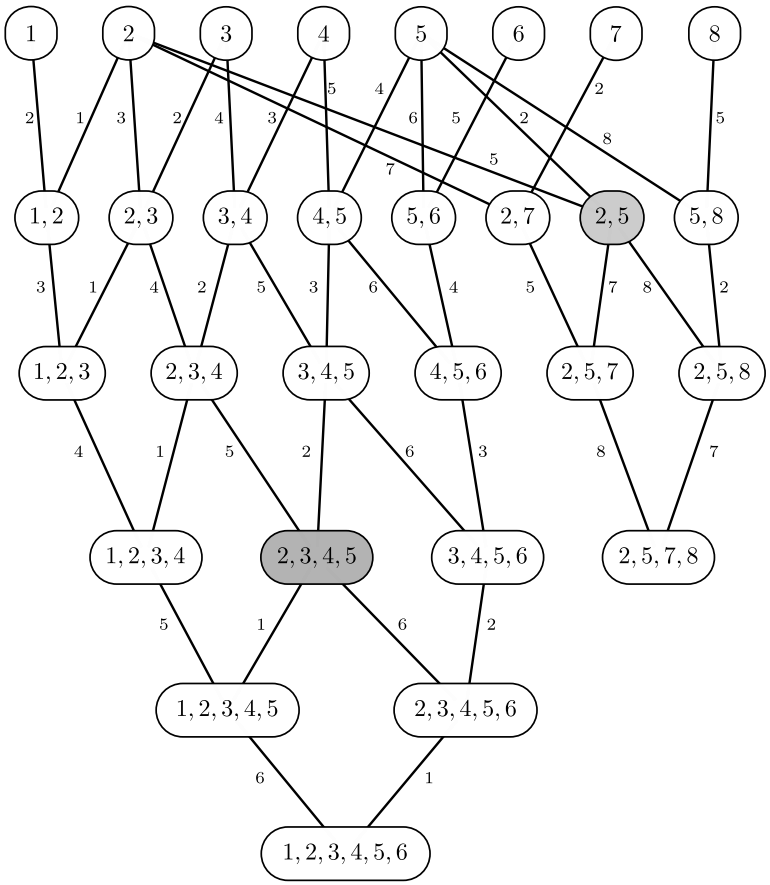


Figure A.1: Graph constructed by the ACI Algorithm for the two strings $S_1 = 1, 2, 3, 4, 5, 6$ and $S_2 = 7, 2, 5, 8$. The locations of the gray shaded character sets should be reported as δ-locations of each other for every $\delta \geq 2$. However, the shortest path between these vertices has length four and contains redundant edge labels. It is therefore not considered in the algorithm.