GS BG Universität Bielefeld

International
NRW Graduate School
in Bioinformatics and
Genome Research

Efficient Algorithms for Gene Cluster Detection in Prokaryotic Genomes

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften an der Technischen Fakultät
der Universität Bielefeld vorgelegte Dissertation

von

Thomas Schmidt

Mai 2005

Betreuer: Prof. Dr. Jens Stoye
Dr. Jörn Kalinowski

Acknowledgments / Danksagung

Diese Doktorarbeit entstand in der Zeit von Juni 2002 bis Mai 2005 am Centrum für Bioinformatik und Genomforschung (CeBiTec) der Universität Bielefeld. Die finanzielle Unterstützung dieser Arbeit wurde von der International NRW Graduate School in Bioinformatics and Genome Research bereitgestellt. Hierzu gilt mein erster Dank Prof. Dr. Enno Ohlebusch, der wesentlich zu meinem Interesse und meiner Aufnahme an der Graduate School beigetragen hat.

Die praktische Durchführung meiner Forschung, sowie die Erstellung dieser Arbeit, wäre ohne die Unterstützung einer Vielzahl von Menschen, denen ich an dieser Stelle meinen persönlichen Dank aussprechen möchte, nicht möglich gewesen.

Insbesondere danke ich Prof. Dr. Jens Stoye sowohl für meine kurzfristige und herzliche Aufnahme in die Arbeitsgruppe Genominformatik, als auch für die ausgezeichnete Betreuung während meiner Forschungszeit und bei der Erstellung dieser Arbeit. Ein weiterer Dank richtet sich auch an die Mitarbeiter des Instituts für Genomforschung der Universität Bielefeld. Hier waren es insbesondere Dr. Jörn Kalinowski, als mein zweier Betreuer, und Christian Rückert, die mir stets in allen Fragen, die die praktische Seite dieser Arbeit betrafen, mit Rat und Tat zur Seite standen. Ebenso möchte ich mich an dieser Stelle auch noch mal bei Ed Hein und Carolin Andreas bedanken, die als studentische Hilfskraft, bzw. während der Anfertigung ihrer Bachelorarbeit, ihren Teil zum Gelingen dieser Arbeit beigetragen haben. Ein herzlicher Dank meinerseits geht auch an Jan Krüger für die Bereitstellung von Gecko auf dem Bielefelder Bioinformatik Server (BiBiServ) und die unzähligen kreativen Kaffeepausen.

Ganz besonders herzlich möchte ich mich an dieser Stelle aber auch bei meinen Eltern Hildegard und Friedhelm für die vielfältige Unterstützung während meines Studiums und der Promotion bedanken. Nicht zuletzt gilt mein ganz persönlicher Dank an dieser Stelle meiner Partnerin Andrea für ihr Verständnis, ihre Geduld und Unterstützung während der Erstellung dieser Arbeit.

Gedruckt auf alterungsbeständigem Papier nach ISO 9706.

Contents

Chapter 1

Introduction

The research in genomics science rapidly emerged in the last few years, and the availability of completely sequenced genomes continuously increases due to the use of semi-automatic sequencing machines. Also these sequences, mostly prokaryotic ones, are well annotated, which means that the positions of their genes and parts of their regulatory or metabolic pathways are known. The traditional way in sequence comparison, e.g. to predict the function of unknown genes, is to establish orthologous relations to well-characterized genes in other organisms on nucleic acid or protein level. A new task in the field of comparative genomics now is to gain gene or protein information from the comparison of genomes on a higher level.

In the field of high-level genome comparison the attention is directed to the gene order and content in related genomes. During the course of evolution speciation results in offspring genomes that initially have the same gene order and content. In the same way the duplication of a whole genome creates a new genome which has two identical copies of the ancestral genome embedded in it. In both cases the offspring genomes underlie a process of divergence over time. This means that events like gene duplication, gene loss, horizontal gene transfer, and many more, result in a changed gene composition, while rearrangements like translocation, transposition, inversion and chromosome fission and fusion disrupt the gene order. If there is no selective pressure on the gene order, the successive occurrence of rearrangements over time will lead to a randomized gene order. Therefore, the presence of a region of conserved gene order is a source of evidence for the functional role of a gene group.

A more detailed examination of the genomic sequences reveals that there is a remarkable difference between the evolution of higher eukaryotes and more primitive organisms including prokaryotes and yeast. The close evolutionary relationship among animals or plants, which have been the subject of comparative mapping studies, is manifested by relatively long *conserved segments*, which are regions of the chromosome with identical gene content and a linear order in both. In contrast, two closely related prokaryotes often have a quite diverged gene order in general, but share many *gene clusters*, which are sets of genes in close proximity to each other, but not necessarily contiguous nor in the same order in both genomes. The existence of such gene clusters has been explained in different ways, e.g. by functional selection, operon formation, and the physical interaction of the gene products. Therefore, the conservation of gene order is a valuable information for many fields in genome research.

From an algorithmic and combinatorial point of view, the first descriptions of the concept of "closely placed genes" in the literature were only fragmentary, and sometimes confusing. The definitions of gene clusters differed as the case arises, and models for gene clusters were based on heuristic algorithms which depended on very specific parameters like the size of gaps between genes. These algorithms also lacked the necessary grounds to prove their correctness, or assess their complexity.

In the first formal descriptions of conserved genomic neighborhoods, genomes are often represented as permutations of their genes, and common intervals, i.e. intervals containing the same set of genes, are interpreted as gene clusters. But here the major disadvantage of representing genomes as permutations is the fact that paralogous copies of the same gene inside one genome can not be modelled. Together with the observation that with an increasing number of genes the relative frequency of paralogous genes inside a genome also significantly increases, it becomes clear that the modelling of genomes as permutations is insufficient for the use on real genomic data.

In this thesis, we developed a new model for gene clusters that allows the presence of paralogous genes in the genomes. Therefore, we generalized the representation of the genomes from permutations to strings and defined a gene cluster as a common character set, which is similar to a common interval but allows disregarding the frequency at which a particular gene occurs in an observed interval. For the established model, we present efficient algorithms to detect the gene clusters in any given number of genomes. Furthermore, we present the results from the application of our algorithms to real genomic data using our developed GECKO software. This software serves as a basic tool for the data preparation, the visualization of the computed clusters, and the interpretation of the results.

Parts of this dissertation thesis have been published in advance [64], and two further papers are submitted [18, 63]. The stand-alone version of GECKO was written in JAVA and can be obtained free of charge for non-commercial use from http://bibiserv.techfak.uni-bielefeld.de/gecko/ together with the data files used for the evaluation of GECKO and further documentation.

1.1 Structure of the Thesis

This thesis consists of seven chapters. In Chapter 2, an introduction into the basic terms and fundamental mechanisms in molecular biology and comparative genomics is given. We introduce the cell as the basic unit of all living organisms, and the DNA with its central role in growth, mutation, and evolution of the cell.

Chapter 3 is devoted to the biological background and the presentation of models and algorithms for gene cluster detection. Based on the formal model of gene clusters as common intervals of permutations, we present a more general and biologically meaningful model, by allowing the presence of paralogous genes. With a new algorithm (Algorithm CI) that we also present in this chapter, we show that gene clusters based on this model extension are still feasible to calculate.

In Chapter 4, we describe different approaches for the grouping of genes from several genomes to gene families, if they show similar amino-acid sequences in each of the genomes. Pretending that genes from one family carry out a similar function in the cell, the classification of genes to their families of homologous genes is the basic input for the following gene cluster detection algorithms.

In Chapter 5, we present the computer program Gecko with the implementation of our algorithms for gene cluster detection. The optimal visualization of the located gene clusters and the detection of patterns regarding the content of a cluster are the essential parts of the program allowing an efficient interpretation of the given results.

The applicability of our programs is shown in Chapter 6. In the first part of this chapter we verify the claimed time complexities on an artificially created data set. Subsequently, we prove the computed results from a set of 20 bacterial genomes on a well-known gene cluster from the literature (the tryptophan operon) as well as on recent studies on a set of genes regarding the sulfur utilization in the family of Corynebacteria.

Chapter 7 concludes the thesis by recalling the main results and presenting a foresight on further applications of gene cluster analysis.

Chapter 2

Biological Background

Since the motivation of this work as well as its applications and results have their origin in the field of genetics, this chapter will give an overview of the central terms and principles of molecular genetics and its recent branch, comparative genomics. This chapter cannot be a self-contained introduction neither to molecular genetics nor to comparative genomics, therefore the interested reader might use any of the relevant textbooks (e.g. [2, 48, 46, 36, 40]). The notation in the first part concerning the molecular genetics is according to [48], whereas the comparative genomics part conforms with [40].

2.1 Terms and Principles in Molecular Genetics

The *cell* is the basic unit of all living organisms. Today we distinguish between two different types of cells, the *prokaryotic cells* (literally, cells without a defined nucleus) and *eukaryotic cells* (cells with a true nucleus). Although they differ radically in their organization, both cell types contain the same groups of complex chemical components: proteins, nucleic acids, lipids, and polysaccharides. This observation leads to the assumption that all cells have evolved from the same type of cell, a universal ancestor whose nature is so far unknown. From a more general point of view, one can look at a cell as a chemical machinery converting one form of energy into another by breaking large molecules into smaller ones, building up larger molecules from smaller units, and performing many other kinds of chemical transformations. Here, the term *metabolism* refers to the collective series of chemical processes that occur in living organisms and the components of the cell's chemical machinery are called *enzymes*. In the cell, the enzymes are protein molecules that are capable of catalyzing specific chemical reactions and the fact, whether a cell is able to carry out a certain chemical reaction or not, depends on the presence of the particular enzyme in the cell that catalyzes this reaction. Generally, the specificity of enzymes is very high, such that even very similar chemical reactions are catalyzed by different enzymes. This specificity is determined primarily by the three-dimensional structure of the enzyme (see

Figure 2.1), which again is determined by the sequence of amino acids that the enzyme consists of.

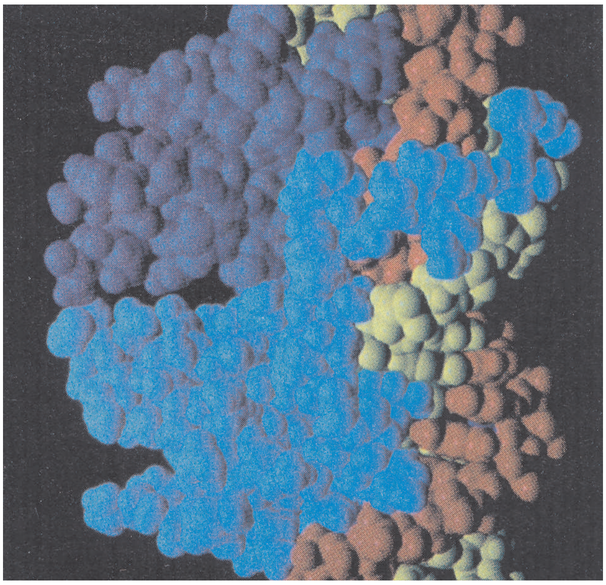


Figure 2.1: The three-dimensional structure of a protein (computer generated model [48]). Different regions of the protein (domains) are displayed in different color.

To answer the question how a cell is able to build precise amino acid sequences, representing the different proteins, one can look at a cell as a *coding device* that contains a storage for coded protein sequence data and a machinery able to access this information. This code is called *genetic code* and it is stored in the sequence of nucleotides in the hereditary molecule *deoxyribonucleic acid (DNA)*. The DNA in the cell is present as two long molecules, which are intertwined forming the DNA double helix (see Figure 2.2).

The DNA is the essential molecule for two major processes of the cell. First, before a cell division occurs, the DNA is being *replicated*, this means that each new cell gets an exact copy of the complete set of genetic instructions. Second, a piece of the genetic code, the *gene*, must be *transcribed* and *translated* to build a new protein. During the transcription, *messenger ribonucleic acid (mRNA)* molecules are formed that contain a complementary copy of the genetic information. In the translation process, the mRNA, now containing the genetic instructions to make the protein, combines with *ribosomes* and through the action of many other factors yields a protein that folds to perform a specific function in the cell (see also Section 2.1.2).

2.1.1 Mutation and Evolution

In the normal growth process of a cell, resulting in the division into two new cells, the replication of the DNA during this process ensures that both new cells finally contain

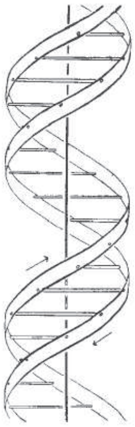


Figure 2.2: The double helix structure of the DNA molecule, detected by James D. Watson and Francis H. C. Crick [76] in 1953.

precisely the same genetic code. Even though the fidelity of the replication is very high, occasionally mistakes in copying the DNA occur. Such mistakes, called *mutations*, may have no detectable effect to the cell, but in other cases may cause the production of a malfunctioning protein, resulting in a defective cell, probably leading to its death. Such cells usually disappear from the population and therefore mutations are generally harmless. In very rare cases a mutation causes the formation of a protein with an improved function. This *selective advantage* over time might result in the replacement of the parent cell type from the population by the mutated type. This process of natural variation by mutation, together with its effects on the population of a species, is called *natural selection* and is the fundamental process of *evolution*.

The course of evolution has lead to the development of the two different cell types of prokaryotes and eukaryotes. The major structural differences between prokaryotic and eukaryotic cells are once the size, prokaryotic cells are approximately 25 times smaller than eukaryotic cells, and also the arrangement of DNA inside the cell. Eukaryotic cells contain a closed compartment, the membrane surrounded nucleus, hosting several DNA molecules. On the other hand, prokaryotes only contain a nuclear region, the *nucleoid*, which is not bounded by any membrane, and contains only a single DNA molecule. Prokaryotes are all bacterial species and can be divided into the two different lineages of *Bacteria* and *Archaea*. In contrast, there are several groups of eukaryotic microorganisms like *algae, fungi*, and *protozoa* and of course all multicellular life forms (plants and animals) are made of eukaryotic cells (see Figure 2.3).

Although given their structure, cells can be classified as either prokaryotic or eukaryotic, the similar structure alone does not necessarily imply an evolutionary relationship. Today's methods for estimating the evolutionary distance between two species (the difference in

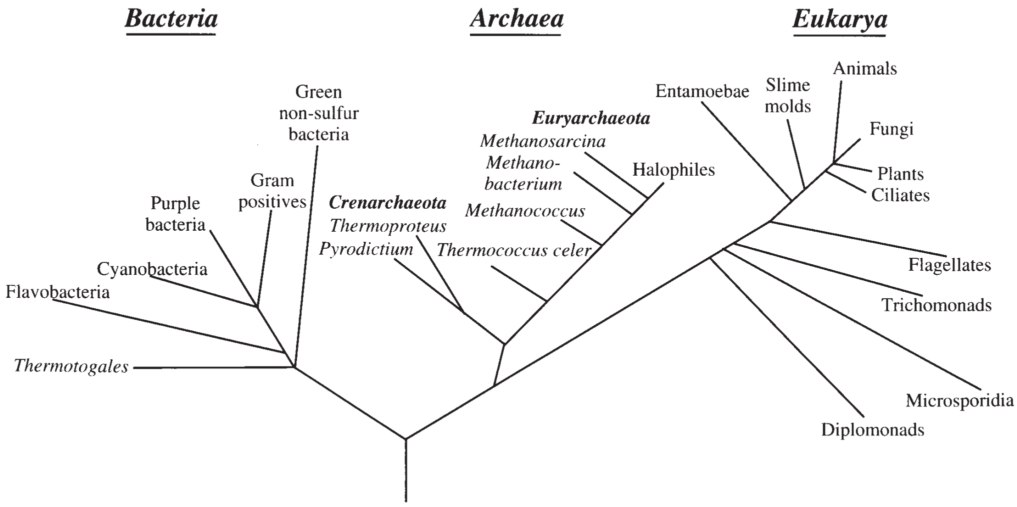


Figure 2.3: The phylogenetic tree created from the comparative sequencing of ribosomal RNA [48]. It shows the evolution of the three groups of living organisms.

time from their divergence up to now) rely on the comparison of their DNA's amino acid sequence. Here it turned out that the part of DNA encoding for the ribosomal RNA (structural RNA of the ribosome, the key cell structure involved in translation) is a well suited criterion for reconstructing phylogenetic (evolutionary) relationships.

2.1.2 DNA - The Code of Life

In the majority of prokaryotes studied so far, most or all of the cellular DNA is found as a single circular molecule. The cell is said to have a single *chromosome*, although the arrangement of DNA within this chromosome differs greatly from that within the chromosomes of eukaryotic cells. Since the length of a single unfolded DNA molecule (approx. 1 mm) exceeds the length of the whole cell by a factor of 1000 (e.g. in *E. coli*), it is folded back on itself many times, this folding is called *supercoiling*. Additionally, many prokaryotic species also contain much smaller but also circular DNA molecules called *plasmids*. Plasmids generally encode proteins that are not essential for cell growth, but provide proteins coding for additional features, e.g. to resist antibiotics or other toxic materials. Generally, prokaryotic protein coding regions can be found compactly attached to each other, such that a large part of prokaryotic DNA can be considered as protein coding.

In contrast, eukaryotic DNA is mostly found divided between two and more chromosomes with variable length, where each chromosome usually contains one linear double-stranded DNA molecule. In addition, protein coding genes in eukaryotes are frequently split into two or more coding regions (*exons*), which are intervened by non-coding regions called *introns*. Compared to prokaryotes, eukaryotic genes are not organized in such a compact consecutive order, there are also larger regions on eukaryotic chromosomes that contain only a few protein coding parts.

The different structures of prokaryotic and eukaryotic genes are accompanied by different mechanisms in their protein synthesis (see Figure 2.4).

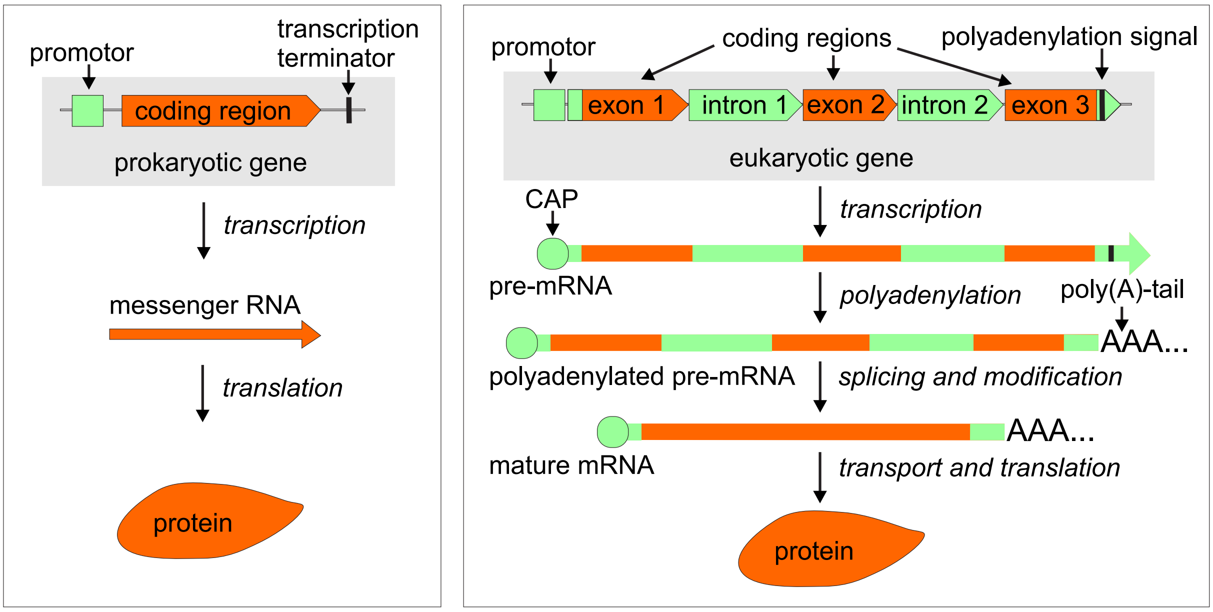


Figure 2.4: Information flow in prokaryotic and eukaryotic cells. The left part shows the DNA processing in a prokaryotic cell. The coding region of a gene is transcribed to its messenger RNA, and during the translation the protein is built. In eukaryotic cells (right) the situation is more complex. During the transcription the pre-mRNA is constructed from the DNA with its introns and exons. Then the polyadenylation adds the poly(A)-tail to the end of the pre-mRNA, and in the following splicing the intron are cut out yielding the mature mRNA. This is transported out of the nucleus into the cytoplasm, where the translation results in the final protein.

In prokaryotes, the synthesis of a protein starts with the docking of the enzyme *RNA polymerase*, responsible for the transcription, in the promotor region of the gene. While reading the DNA sequence starting from the promoter region, the RNA polymerase synthesizes the mRNA molecule until it reaches the transcription terminator. Then the protein is built by translating the mRNA into a sequence of amino acids. This is done by the ribosomes that read the sequence of nucleotides of the mRNA and for each triplet of nucleotides or *codon* add the encoded amino acid to the so far constructed amino acid chain. When the ribosome reaches a stop codon, indicating the end of the mRNA, it releases the protein that then folds to its final structure. Sometimes, we can find two or more protein coding regions in-between a promotor and stop codon. These genes are said to form an *operon*, i.e. a set of genes being always transcribed and translated together.

The protein synthesis in eukaryotes starts, similarly to prokaryotes, with the binding of the so called *RNA polymerase II* at the transcription start site in the promoter region of the eukaryotic gene. Shortly after the transcription begins, a typical feature of eukaryotic mRNA, the *cap*, is added to the start of the mRNA. After the last exon is transcribed, the *polyadenylation signal* in the gene allows the construction of the *poly(A)-tail*, and during the following *splicing*, the introns are cut out of the pre-mRNA. Many pre-mRNAs are spliced following a fixed scheme in which each intron is simply removed from the sequence. In contrast, the *alternative splicing* allows also to cut coding regions (exons) from the pre-mRNA. To this end, one eukaryotic gene can be spliced into many different mature mRNAs, coding for different proteins. Finally, the mature mRNA is transported out of the nucleus into the cytoplasm of the cell, and similar to prokaryotes, the ribosomes translate the mRNA constructing the encoded protein.

Finally, we can summarize that the DNA, or more precisely its whole nucleotide sequence that we call *genome*, defines the structure and behavior of each living cell. Therefore, the next section addresses the question what information we can infer from an organism when we have access to its nucleotide sequence.

2.2 Introduction to Comparative Genomics

More than 25 years ago, the development of an automated DNA sequencing process by Fred Sanger *et al.* [61] provided the opportunity to quickly reveal the genomic sequence of an organism. Up to now (May 2005) there are more than 261 completely sequenced genomes published (21 Archaea, 207 Bacteria, 33 Eukarya), and the Genomes Online Database GOLD[1] contains over 1100 sequencing projects that are currently in progress.

The following introduction mainly concerns comparative genomics on prokaryotes. Here a central topic is the detection of proteins and the explanation of their function in the cell. In comparative genomics on eukaryotes, the attention is mostly directed on other topics like conserved splicing and detection of transcription factor binding sites. These topics will not be part of this thesis and therefore we will skip this area in the introduction. Before we focus on the problem of locating the protein coding sequences in the DNA and the prediction of the function of these proteins, mainly based on amino acid/DNA sequence comparison, it is necessary to clarify the meaning of some basic and often inconsistently used terms.

[1] http://www.genomesonline.org

2.2.1 Similarity and Homology

A central concept in comparative genomics is the search for significant DNA sequence similarities to infer homology. Unfortunately, the term “homology”, even in many scientific publications, is often misused as synonym for similarity [55], while its correct meaning is simply “of common origin”. The distinction between homology and similarity arose for research in zoology and botanics, e.g. our hand and the wing of a bat are homologous organs, but the bat’s wing is ’only’ analogous (similar) to the wing of the butterfly. Homology does not necessarily imply similarity, and similarity does not guarantee homology, although in most cases it is a good indication. Considering the above definition, commonly used phrases like “sequence homology” or “significant homology” should be seen only as substitutes for “(significant) sequence similarity”.

If nowadays we conclude that two genes or proteins are homologous, this is always only a conjecture. The true evidence for homology can only be given by the exploration of their common ancestor and all intermediate forms. Since we do not have fossil records of these extinct forms, we can only predict homology based on the similarity of their DNA sequences. This sequence similarity can be expressed numerically and correlated with probability. Generally, we can state that the higher the similarity between two given sequences, the higher is the probability that they have not originated independently from each other and their similarity is merely by chance. And indeed, we find many proteins, conserved in a large number of species, showing such a high similarity that there seems to be no doubt that they are homologs.

On the other hand, the situation gets more difficult when the observed sequence similarity gets lower. The difficult question to answer is where to draw the line between similarity indicating homology and similarity due to other reasons or just by chance. In fact it seems that there is no fixed criterion to answer this question. A lower level of similarity might also indicate homology if additional other properties (e.g. similar protein structure) apply. One way to answer the question where to draw this line is discussed in Chapter 4.

One should be aware that common ancestry alone is not the only possible explanation for an observed sequence similarity. There exists the second logically consistent alternative that similarity is the result of convergence from unrelated species. In this case the similar sequence of genes is strictly required to perform their similar function. For example, convergence was detected in nature on a set of lysozyme molecules [42, 66, 67] in different species. But, in most cases the region of convergent sequence positions comprises only a few amino acid residues, which are only a small part of the protein. So, convergence alone is not able to explain the observable sequence similarity of whole proteins.

2.2.2 Orthologs and Paralogs

For a correct protein function inference, it is important to distinguish between two principal types of homology. These types, called *orthology* and *paralogy*, differ in their evolutionary relationship and may lead to different functional implications [22, 23]. The term orthology describes genes in different species, being derived from a single gene in their common ancestor. On the other hand, two genes are called paralogous if they have evolved through a gene duplication within the same ancestral genome (see Figure 2.5).

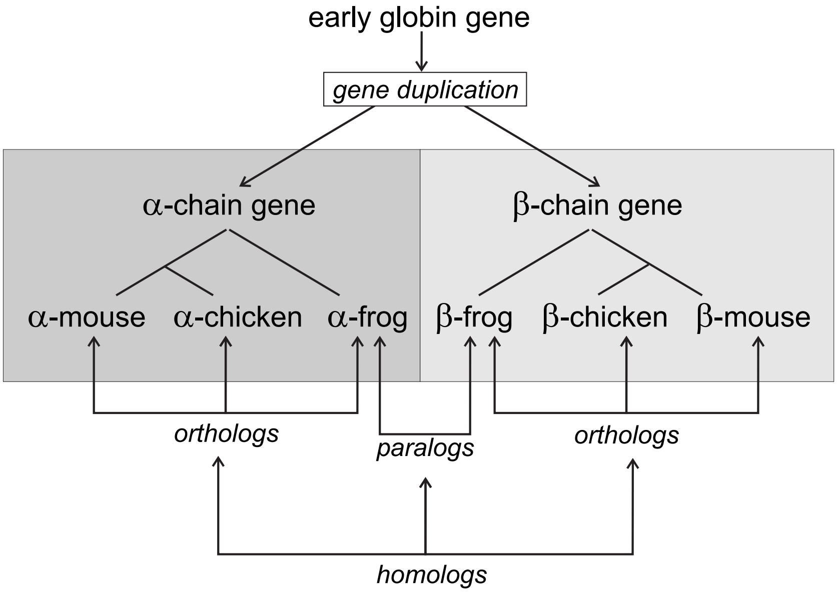


Figure 2.5: The development of a globin gene in the chicken, mouse, and frog genomes.

The identification of orthologous genes and proteins is an essential source of information for the functional genome analysis in comparative genomics today. A set of orthologous genes usually performs the same function as their common ancestor from which they are derived. Therefore, in most cases it is sufficient to explore the function of a single gene from such a set and transfer its function to all other genes of that set. This works well at least for those organisms that carry only a single copy of this gene and do not contain further genes capable of providing the same function. If the organism relies on the function of the encoded protein, this puts a strong selective pressure on the conservation of this gene.

In contrast, paralogous gene copies are free to evolve new functions. After emerging from a gene duplication event, they can distribute the selective pressure among each other. Today, sequenced genomes show a rate of 25% to 80% of genes being member of a family of paralogs [34], reflecting the functional diversification that occurred at different periods in evolution.

Unfortunately, the interaction of speciation events, the origin of orthologous genes, and gene duplications, responsible for the paralogous families, result in very complex evolutionary scenarios, making it quite impossible to infer their precise way of evolution. This situation can get even more tricky when considering the fact that during evolution in certain lineages some copies of a gene may have disappeared (*lineage-specific gene loss*) [6, 14] or after speciation duplicated orthologs evolved independently (*lineage-specific expansion*) [35, 45].

Summarizing, we can state that the observation of sequence similarity between genes is a hint to their common origin. The knowledge about homology can be used as a basis to transfer information concerning function or evolution from an experimentally verified protein to an uncharacterized homolog. However, those inferences should always be treated with the appropriate caution, since it is not always simple to elucidate their true way of evolution up to now.

2.2.3 Gene Prediction

A first important issue after finishing the DNA sequencing is to locate the genes, i.e. the regions of the DNA sequence that potentially encode proteins. Prokaryotic genes are usually made up of uninterrupted stretches of DNA between a start codon (typically ATG, but sometimes also GTG, TTG, or CTG) and a stop codon (typically TAA, TGA, or TAG). This region is called an *open reading frame (ORF)*, and its average length is approx. 1000 base pairs (bp), while the usual minimum length of an ORF is at least 100 bp. For most common purposes a suitable strategy to define a gene is to take the longest ORF for a given region of DNA. Further evidence for a predicted ORF can be obtained using the following methods, e.g.:

- search a database for homologous proteins encoded by a similar sequence;
- calculate the codon bias or other statistical features of the sequence and compare it to that of known proteins;
- search for a typical ribosomal binding site preceding the predicted coding region;
- check if the coding region has a typical promotor region in front.

Generally one can distinguish between two different types of gene identification approaches. The *intrinsic* methods rely only on the evaluation of the statistical properties of the DNA sequence (length, codon usage, presence or absence of a promotor region or ribosomal-binding site). These methods are provided with a training set of known protein sequences, from which they extract the information how a protein coding sequence should look like. Popular examples for these intrinsic methods are GeneMark [11], Glimmer [17], and the more recent program ZCURVE [28].

The *extrinsic* approach to predict a gene includes a comparison of the putative protein coding sequences with verified homologous protein sequences from a database, and a search for functional motifs. If the putative ORF product has a significant similarity to at least one protein from the database, one can be quite sure that the ORF under observation is a true gene. Obviously, the combination of statistical and similarity based approaches has the potential to improve the reliability of the results obtained by a single method. Two programs that implement such combination are ORPHEUS [24] and CRITICA [7].

A still unsolved problem for all described methods is the objective evaluation of their predictive accuracy, since only a small fraction of the predicted genes are experimentally verified. The only way out is the use of a sample set where the location of the genes is known. This set is divided into a training set and a test set. The accuracy of the prediction then can be expressed in terms of sensitivity and specificity for each applied method (see also [12]).

2.2.4 Function Prediction and Annotation

The next step after locating the protein coding regions on the DNA is to explore their functional role and biological meaning for the organism. Unfortunately, the discovery and verification of protein functions in wet-lab experiments is extremely time consuming and very expensive. Therefore, bioinformatic tools have been developed that provide an automated prediction of the possible function of a gene. Again, the most important information that is used in different approaches, is the sequence and structural similarity to proteins from a database whose function is already experimentally verified. On average we can observe that for approx. 30-35% of the genes no reliable prediction can be made with such methods, and for many of the rest the predicted functions are very general and need to be refined [27]. A further complication arises from the fact that orthology (inferred from similarity) in general is not a one-to-one relationship (see Section 2.2.2). In those cases the transfer of functional assignments has to be handled with a special caution, because paralogous gene copies might have acquired new functions.

Another source of evidence for a functional prediction can be derived from conserved *phylogenetic patterns* [53, 26, 80]. A phylogenetic pattern is defined by a set of genomes in which a group of orthologous genes appears. The general idea is that if a group of genes together performs a required function in an organism, e.g. they participate in one important metabolic pathway, they all should be conserved during evolution. In that sense, from the detection of groups of orthologs with a similar phylogenetic pattern, one can infer that these genes are functionally associated. In contrast, the detection of a complementary phylogenetic pattern is a strong indication of an evolutionary event called *non-orthologous gene displacement*. Here, the function of a particular gene was taken over by another gene that has no common ancestor, and therefore no similarity, with the original gene [27]. For those genes, the inference of function only by similarity is convicted to fail.

The evaluation of genomic neighborhoods is an additional approach towards the discovery of protein function. Here, the central idea is that functionally related proteins tend to conserve their genomic neighborhood, and in many cases form an operon [21]. How to use these *gene clusters* for functional and evolutionary predictions is the main topic of Chapter 3.

Chapter 3

Gene Clusters

In this chapter, we present the concept of gene clusters and its manyfold applications in comparative genomics. In the literature, the concept of gene clusters is frequently used, often with varying meanings, lacking a clear definition of what it stands for, and how it can be used to gain information from. Therefore, in the beginning of this chapter, we focus on the biological relevance of gene clusters and their application in genome annotation and evolutionary analysis. Afterwards, we present different formal models of gene clusters, accompanied by efficient algorithms to detect them, and discuss their applicability to real genomic data.

3.1 Biological Relevance of Gene Clusters

The first emergence of the term gene cluster in the literature dates back to the year 1966 where R. H. Bauerle and P. Margolin [8] described the functional organization of five genes forming the “tryptophan gene cluster” in *S. typhimurium*, compared to their organization in *E. coli.* In the following years, the detection of the tryptophan gene cluster was reported also in other species [54], and numerous other clusters were reported. The discovery of more and more gene clusters raised the question what the reasons for the conservation of clusters over time might be and what information regarding function and evolution these clusters contain [13, 58].

The first systematic approaches discovering the mechanisms behind gene cluster conservation were reported in the mid-nineties when complete prokaryotic genome sequences became available. Shortly after (almost) finishing the sequencing of the first bacterial genomes in 1996, Tatusov *et al.* [73] observed that the gene order between the two closely related genomes of *E. coli* and *H. influenzae* surprisingly showed almost no conservation for regions larger than a few genes. In [49], Mushegian *et al.* described a similar observation on a larger set of also more distantly related genomes, leading to the conclusion that generally, the gene order in bacterial genomes is under no selective pressure and therefore extensively shuffled over time [39]. On the other hand, those local regions of gene order conservation were found even in distantly related species, where here the order inside the conserved region again may vary. This trend in evolution to conserve the formation of locally conserved clusters over time allows the inference of many different types of information about the functions of genes in those clusters as well as their evolution.

3.1.1 Using Gene Clusters for Function Prediction

In 1997, Tamames *et al.* [69] developed a combined approach of sequence comparison and functional gene product classification on the genomes of *E. coli* and *H. influenzae*, studying their gene order relationship and genome organization. They addressed the question whether and to what degree functionally related genes are adjacent within and between the two bacteria. For this purpose, they classified all genes from the two genomes into eight general functional classes (e.g. transport, metabolism) and compared the number of neighboring gene pairs assigned to the same functional class with those belonging to different functional classes inside each genome and afterwards between the two genomes. In both cases, they observed a statistically highly significant tendency for genes of the same functional group to occur as a consecutive gene pair and proposed that this clustering represents some functional constraints on the genomes that are maintained by natural selection. Additional support for the hypothesis that evolutionary pressure leads to the conservation of gene clusters comes from the observation that 90 % of the conserved gene pairs found between *E. coli* and *H. influenzae* maintain their relative orientation, i.e. they are transcribed in the same direction.

One year later, Dandekar *et al.* [15] took advantage of the availability of more published genome sequences and developed an approach based on the comparison of nine genomes. They divided them into three groups of three genomes each, such that in each group at least one more distantly related genome is present, since the conservation of gene order in closely related species is more likely to be a result of a lack of time for genome rearrangements after their divergence from a common ancestor. In each of the three genome sets, they were able to locate approximately 100 genes conserved in groups of at least two genes, from which at the time of study at least 75 % of the genes are experimentally verified to encode functionally interacting proteins.

Both articles clearly reveal that gene order conservation is a hint at functional interaction of gene products. Therefore, it could routinely be used as a tool for predicting protein function in several different ways [15]:

1. If the products of two genes have only been tentatively assigned functions, their location in a conserved cluster can be used to predict the functional interaction as well as their function.

2. If a gene with an unknown function is found in a conserved cluster among several genomes, the functional interaction of the genes may give valuable information delineating the search space for their unknown function.

3. Even if the function of all individual genes in a cluster is known, the fact that they are not randomly distributed in the genome might reveal novel functional or evolutionary aspects (see following Section).

3.1.2 Using Gene Clusters for Evolutionary Analysis

Similar to the function prediction for gene products based on sequence similarity, also the first described studies of molecular evolution used the comparison of single genomic sequences. Here again, the availability of completely sequenced genomes allows to direct the analysis to a higher, more comprehensive level of whole genomes.

Addressing the question, how genomic information can be used to obtain information about genome evolution, Huynen and co-workers [33, 34] described how genome content can create baseline expectations for measures of genome distances. In their approach working on the first nine available genome sequences, they used information from the fraction of known orthologous sequences between the genomes and combined it with the conservation of gene order and the spatial clustering of genes for which an ortholog is present in another genome. After correlating these measures with the divergence time between the compared genomes they were able to derive calibration curves that show how their selected indicators for genome evolution can be used to extract information about the evolution of new microbial genomes.

A more detailed analysis of the relation between the amount of gene cluster conservation and evolutionary distance was performed by Tamames in [68], observing that the distribution of gene clusters over the evolutionary distance fits to a sigmoid curve (see Figure 3.1). This curve can be subdivided into three parts, where the first part (dashed line) shows that the gene order conservation for closely related species is very high since there was not enough time for an extensive genomic rearrangement. The second part (solid line) shows a region of decreasing gene order for phylogenetically more distant species. In this area, gene order conservation could be used as one criterion to estimate the phylogenetical distance between the analyzed genomes. For a larger evolutionary distance, the curve shows a saturation (dotted line), where for an increasing genomic distance, the amount of cluster conservation is constant at a low level. The gene cluster analysis of genomes from this region with a larger phylogenetic distance is especially relevant for the inference of functional association between the gene products of a conserved cluster, since here the conserved order cannot be due to a recent descent from a common ancestor.

All described approaches show that the conservation of gene order is a source of information for many fields in genomic research. Especially the combination of information

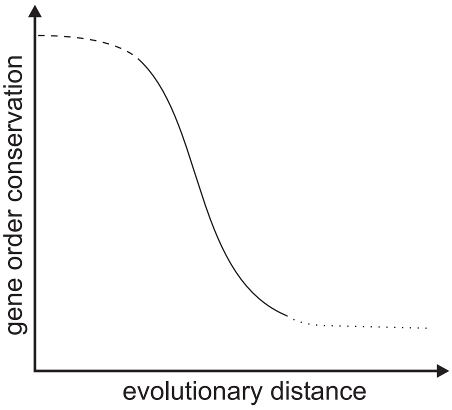


Figure 3.1: Sigmoid distribution of conserved gene order with increasing evolutionary distance. The distribution is divided into three parts. The first part (dashed line) shows the very high level of gene order conservation for phylogenetically closely related species. In the second part (solid line) the conservation of gene order decreases constantly with an increase in evolutionary distance, and in the end (dotted line) the cluster conservation reaches a saturation level for distantly related species.

from conserved gene clusters with different approaches in comparative genomics, like detection of gene fusion and phylogenetic profiling, provides a powerful strategy to unravel their way of evolution or the hidden function of certain gene products [79]. Therefore, the detecting of conserved gene clusters is an important task on the way of discovering of so far missing pieces in the puzzle of bacterial life.

3.2 Existing Approaches for Gene Cluster Detection

After discovering the value of gene cluster conservation, in recent years many approaches have been developed to detect regions of conserved genomic neighborhood. These regions are used to infer evolutionary information or functional associations of the proteins encoded by genes from those gene clusters.

3.2.1 Gene Cluster Detection using Conserved Gene Pairs

Overbeek *et al.* [52] developed an automatic procedure to detect gene pairs of *close* orthologs, where close in this case means belonging to one *run* of genes (i.e. a set of genes separated by at most 300 bp on a genome), and *orthologs* are operationally defined as bidirectional best hits (BBH) between two genomes. Additionally, they scored the hits according to their phylogenetic distance inferred from the rRNA-based phylogenetic tree, since it is expected that pairs of close orthologs are more likely to occur by chance in phylogenetically closely related species than in distantly related genomes. They also suggested some relaxation to the definition of close gene pairs by not insisting on BBHs and instead using a threshold defined by a certain expectation value for the observed sequence similarity. With their approach they were able to reconstruct several previously known metabolic pathways [51].

The general concept of conserved gene pairs of Overbeek *et al.* was extended by Lathe and co-workers [44] with some modifications in the parameter settings for a conserved pair (a shorter distance of 250 bp for close pairs, and same transcriptional orientation). Starting from the detection of a conserved gene pair they search for more conserved neighbors in further genomes. If such neighbors are found, these are added to the set of conserved genes which they call *über-operon*, and the procedure starts again on the newly added genes. Applying the method to 15 prokaryotic genomes, they observed that although the exact neighborhood of each gene is not necessarily conserved, each gene can be found in a context of transcriptionally associated genes. Therefore, this method is especially useful to locate gene clusters that do not form an operon-like structure although their genes are transcriptionally related [41].

Web-based Tools

The implementation of Overbeek's approach was the first genome context-based operon comparison tool which was available in the WIT database[1]. Unfortunately, this database has been closed for the public in 2001.

Another well known web-based tool for gene cluster detection is the Search Tool for Recurring Instances of Neighboring Genes STRING[2], developed by Bork and colleagues [65]. This tool is based on a similar approach as WIT. The STRING search starts from a single protein sequence that can be entered as a FASTA[3] file or just by its gene name from a complete genome. If the sequence is entered in FASTA format it is compared against the database of all proteins encoded in complete genomes so that the user can choose one of the best hits as query gene. Then STRING performs an iterative analysis of gene neighborhoods around the query gene in all genomes in the database. After the nearest neighbors of a gene in question are identified, the next iteration of STRING would look for their neighbors and record if any of these were found previously. If no new neighbors are found, STRING reports that the search has "converged". If this does not happen even after five consequent search cycles, the program would just tabulate how many times each particular gene was found in the output. The good graphical representation of genes and their neighborhood makes STRING a fast and convenient tool to search for consistent gene

[1] http://wit.mcs.anl.gov

[2] http://www.bork.embl-heidelberg.de/STRING

[3] http://bioweb.pasteur.fr/seqanal/interfaces/fasta.html

associations in complete genomes, given a single query gene.

The similarity neighborhood approach web tool SNAPper at MIPS[4] [38] is similar to STRING, but instead of using pre-computed pairs of orthologs, it looks for sequence pairs with a user-defined significant similarity. In addition, SNAPper does not require the related genes to form conserved gene strings, they only need to be in the proximity of each other. SNAPper looks for the homologs of a given protein, then takes neighbors of the corresponding genes and again looks for their homologs, and so on. The program then builds a similarity-neighborhood graph (SN-graph), which consists of the chains of orthologous genes in different genomes and adjacent genes in the same genome. The hits that form a closed SN-graph, i.e. recognize the original set of homologs, are predicted to be functionally related. An advanced version of SNAPper offers the choice to select several parameters, allowing a fine-tuning of the performance of the tool depending on the particular query protein.

3.2.2 Gene Cluster Detection using Graph Comparison

Besides the approach of locating gene clusters by identifying conserved gene pairs, another successful way to detect gene clusters is based on the comparison of different types of graphs.

In 2000, Ogata *et al.* published a method for identifying correlated gene clusters based on a graph comparison algorithm [50]. The general idea of this method is that one can represent the sequence of genes in a genome as a one-dimensional connected graph whose vertices correspond to the genes and two adjacent genes are connected by a single edge disregarding the direction of transcription. A second graph is derived from the Kyoto Encyclopedia of Genes and Genomes (KEGG[5]) database, where a metabolic pathway from the database is treated as a graph with gene products as vertices connected by edges representing the chemical compounds used. Thus, two adjacent vertices represent successive reaction steps in the used pathway. Given these two graphs, the essential procedure then is to extract a set of gene products catalyzing successive reactions in a pathway that are encoded by genes in close locations on the genome. Therefore, the correspondence between the two graphs is given by EC[6] numbers, which assign to each gene its enzymatic function (if known). Then the detection of functionally related enzyme clusters (FREC) becomes a problem of detecting groups of EC numbers that are formed by clusters of corresponding vertices in both graphs.

With this approach Ogata *et al.* developed a method to locate functionally associated genes in a single genome when their functions are known. On the other hand, this method

[4] http://pedant.gsf.de/snapper/

[5] http://www.genome.ad.jp/kegg/

[6] http://www.chem.qmul.ac.uk/iubmb/enzyme/

can also be used to find clusters of conserved genes between two different genomes, by replacing the graph from the KEGG database with the graph representation of a second genome. In this case, the connection between the two graphs (EC numbers in the case of using KEGG) could be represented by links based on sequence similarity. With this graph representation, Fujibuchi *et al.* [25] developed a three step procedure to find conserved clusters in multiple genomes. In the first step of this procedure the graph comparison is used to identify correlated clusters in a pairwise genome comparison. Afterwards, all corresponding pairwise clusters are combined using P-quasi complete linkage clustering, which is an intermediate between single and complete linkage clustering and requires that each element of the cluster has a connection to at least $P\,\%$ of the other elements. In the final step, the located gene clusters are analyzed in more detail trying to identify gene fusion events or paralogs. Applying their method to a set of 16 prokaryotes and one uni-cellular eukaryote, it turned out that the gained results significantly depend on the determination of the parameter P. An additional disadvantage of this graph based approach is the expected running time for 17 genomes, which they expected to be about one week on ten CPUs in parallel.

Finally, a combination of the pairwise and graph based approaches can be found in [59], where Rogozin and co-workers delineated a more general type of connected gene neighborhood, called *gene arrays*, which is similar to the über-operon concept of Lathe *et al.* [44] since it combines partially overlapping regions of conserved gene order.

Summary

The preceding sections showed that the conservation of gene order contains valuable information for many fields in genome research, especially for gene function prediction and the inference of evolutionary relationships. Unfortunately, the definition of gene clusters differs significantly from method to method, and the underlying models are often based on heuristic algorithms, which depend on very specific parameters like the size of gaps between genes, the completeness ratio of clusters, or the scoring scheme for phylogenetic distances. This leads to the problem that generally the results of these methods are not comparable, and the fact that so far only a few gene clusters are experimentally verified makes it even harder to rate the quality of the clusters reported by a single method.

Another drawback of all methods described above is that due to the missing formal model, the development of sound and efficient algorithms is very difficult, but indispensable due to an increasing number of available sequences. Finally, a statistical analysis to test the significance, if an observed gene cluster occurs just by chance, is also missing in many of the described approaches. Such an analysis was performed by Durand and Sankoff [20], who present probabilistic models to determine the significance of gene clusters, but leave open the question how to detect these gene clusters in two or more given genomes.

Therefore, the following sections of this chapter address the establishment of formal models for gene clusters together with efficient algorithms for their detection in any given number of genomes.

3.3 A First Formal Model for Gene Clusters

The first rigorous formulation of the concept of a gene cluster was given by Uno and Yagiura [75]. Representing the genomes as permutations of numbers, they introduced the notion of *common intervals* as contiguous regions in each of two permutations containing the same elements. This simplified representation was the origin for many studies on modelling gene clusters, therefore we will discuss this model with some of its extensions in more detail in the following.

3.3.1 Gene Clusters and Common Intervals

The following high-level genome representation was motivated by studies in the field of global genome rearrangement. The representation of genomes as permutations is used in the evolutionary analysis of genomes [29, 62] as well as in the functional analysis of gene order [31, 32, 75].

Genes are described by numbers from the set $N := \{1, 2, ..., n\}$, so that a genome is given by the sequence of its genes, which is a permutation π of the numbers from N. By $\pi(i) = j$ is denoted that the ith element of π is j. A set of indices $\{x, x+1, ..., y\}$ is denoted by $[x, y]$ for $x, y \in N$ and $x \leq y$. The corresponding set of elements $\{\pi(x), \pi(x+1), ..., \pi(y)\}$ from π is given by $\pi([x, y])$ and called an *interval* of π. Let $\Pi = (\pi_1, \pi_2, ..., \pi_k)$ be a family of k permutations of N. W.l.o.g. in the following is always assumed that $\pi_1 = id_n := (1, 2, ..., n)$.

With the representation of genomes as permutations, we define a simple and restricted model for gene clusters based on the definition of common intervals.

Definition (common interval). A *common interval* of the family Π is a k-tuple $c = ([l_1, u_1], ..., [l_k, u_k])$ with $1 \leq l_j < u_j \leq n$ for all $1 \leq j \leq k$ if and only if:

$$\pi_1([l_1, u_1]) = \pi_2([l_2, u_2]) = ... = \pi_k([l_k, u_k]).$$

This definition allows to identify a common interval c by the contained elements, i.e. $c \equiv \pi_j([l_j, u_j])$ for $1 \leq j \leq k$. Since $\pi_1 = id_n$, the above set equals the index set $[l_1, u_1]$, and this will be the *standard notation* for a common interval. The set of all common intervals of Π is denoted C_Π and because of $l_j < u_j$ it contains no common interval of size one.

Example. Let $N = \{1, ..., 5\}$ and $\Pi = (\pi_1, \pi_2)$ with $\pi_1 = id_5$, and $\pi_2 = (1, 4, 5, 2, 3)$. We have

$$C_\Pi = \{([1, 5], [1, 5]),\ ([2, 3], [4, 5]),\ ([2, 5], [2, 5]),\ ([4, 5], [2, 3])\},$$

and in standard notation:

$$C_\Pi = \{[1, 5], [2, 3], [2, 5], [4, 5]\}.$$

3.3.2 Algorithms for finding Common Intervals

After defining the set of all common intervals of Π, now the challenge is to develop an efficient algorithm that finds all elements of C_Π. If Π consists of only two permutations ($k = 2$), an easy test if an interval $\pi_2([x, y])$, $1 \leq x < y \leq n$, is a common interval of $\Pi = (\pi_1, \pi_2)$ is based on the following functions:

$$\begin{aligned} l(x, y) &:= \min \pi_2([x, y]), \\ u(x, y) &:= \max \pi_2([x, y]), \\ f(x, y) &:= u(x, y) - l(x, y) - (y - x). \end{aligned} \tag{3.1}$$

Since $f(x, y)$ counts the number of elements in $[l(x, y), u(x, y)] \setminus \pi_2([x, y])$, an interval $\pi_2([x, y])$ is a common interval of Π if and only if $f(x, y) = 0$.

An easy way to find C_Π is to test for each pair of indices (x, y), $1 \leq x < y \leq n$, if $f(x, y) = 0$, yielding a naive $O(n^3)$ time algorithm. The usage of running minima and maxima reduces the time complexity to $O(n^2)$. Uno and Yagiura [75] described an algorithm where the main idea was to reduce the number of pairs to be tested by introducing *wasteful candidates*. This resulted in an $O(n+K)$ time algorithm for finding all K common intervals of two permutations of n elements.

Heber and Stoye [32] extended this solution to find all common intervals of k permutations. Therefore, they introduced the concept of *irreducible intervals* and presented an algorithm that runs in optimal $O(kn + K)$ time, where again K is the output size. They also showed how to deal with circular permutations, multi-chromosomal permutations and signed permutations, where the sign is a representation for the coding direction of the gene [31].

To apply the gene cluster model on real data, the genomes have to be transformed into permutations. This transformation is usually performed using the following preprocessing procedure:

1. Detect the genes in the genomes (using a gene prediction algorithm, see Section 2.2.3).

2. Find for each gene its orthologs by searching for bidirectional best hits (BBHs) in the other genomes (using a similarity based approach, see Section 2.2.2).

3. Delete all genes from the genome sequence having no corresponding ortholog in each of the other genomes.

4. Number the genes such that:

   - the first genome sequence becomes the identity permutation $\pi_1 = id_n$,
   - the genes in the following sequences are numbered so, that each gene gets the same number as its ortholog in π_1.

3.3.3 Limitations of the Model

In the application to real data, the simplicity of this model for gene clusters, based on the definition of common intervals, comes at the cost of quality. There are many assumptions and simplifications made, which make this model of gene clusters insufficient to the use on real biological data. The problems causing these insufficiencies can be divided into two groups, where the first group is the preprocessing of the data, and the second group of problems results from the strictness of the model.

Preprocessing errors: During the preprocessing, each orthologous gene is assigned a unique number from the permutation over N. This causes problems in cases, where paralogous genes inside a genome are present (Figure 3.2a). A second reason for an insufficient cluster detection is the usage of BBHs to identify orthologous genes. This may lead to a wrong classification of orthologous genes, especially when there are two or more hits with almost the same significance of similarity (Figure 3.2b). Further sources of errors resulting in a poor quality of detected clusters are the disregarding of the coding direction of the genes and the neglecting of the length of the non-coding regions between the single genes.

Errors from strictness: The stringency in deciding whether an interval of two genomes is a common interval is given by the function f in Equation (3.1) and simply extends for multiple genomes. Exactly the same (number of) orthologous genes have to be part of the region under observation, and this for each of the handled genomes. But over time there might occur evolutionary events, causing that a gene product is no longer required for an organism, or its function is taken over by another gene. So, the gene originally coding for that product does no longer underlie the evolutionary pressure of staying in the cluster. It can be expected, that over time this gene will move out of the cluster. After such an event the described algorithms are no longer able to detect the involved gene cluster, just

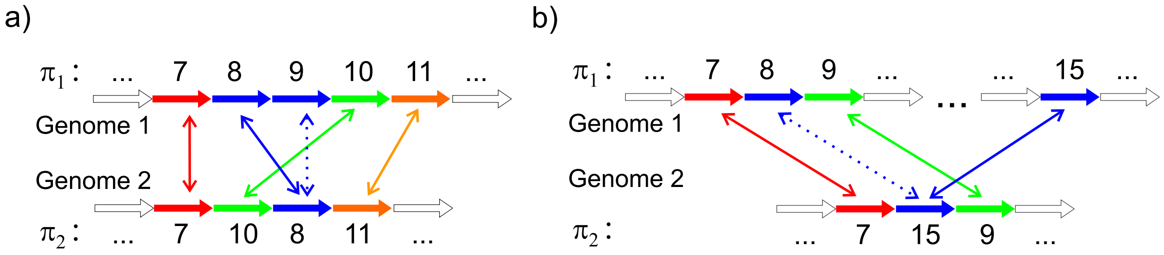

a)
π_1: ... 7 8 9 10 11 ...
Genome 1
Genome 2
π_2: ... 7 10 8 11 ...
b)
π_1: ... 7 8 9 ... ... 15 ...
Genome 1
Genome 2
π_2: ... 7 15 9 ...

Figure 3.2: Errors from preprocessing. The genomes are drawn as a sequence of arrows, where arrows with the same color indicate homologous genes. In (a), a duplication of the blue gene in Genome 1 raises the problem to assign the "correct" ortholog to the blue gene in Genome 2. The choice using the '8' (solid blue line) results in a scenario where no gene cluster could be detected, while using the '9' (dotted blue line) will result in a partially conserved region of the elements 9,10,11. In (b), the blue gene 15, duplicated from gene 8 outside the cluster in Genome 1, is more similar to the blue gene 15 in Genome 2. With this assignment the whole gene cluster could not be detected, while assigning the 8 to the blue gene in Genome 2 (dotted line) is the preferable choice to detect the cluster.

because of a single missing gene in one genome. Therefore, it seems to be necessary to develop models allowing gene clusters with a certain amount error tolerance.

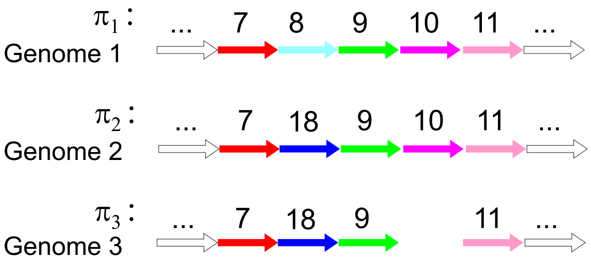

π_1: ... 7 8 9 10 11 ...
Genome 1
π_2: ... 7 18 9 10 11 ...
Genome 2
π_3: ... 7 18 9 11 ...
Genome 3

Figure 3.3: Errors from strictness. In Genome 2 and 3, gene 8 (light blue) is replaced by gene 18 and additionally in Genome 3 gene 10 is missing. In this scenario, no cluster will be reported, even though the conservation of the other genes is a highly valuable information.

3.4 Modelling Paralogous Genes

In the previous section, we have seen that the development of a formal model for gene clusters allows the creation of efficient algorithms detecting them. However, we have also seen that if a model is too abstract, then it is not able to produce results of sufficient quality. A possible approach to build a more accurate cluster model could contain the incorporation of more genomic information (e.g. coding direction, paralogs, length of a gene, ...) into the model. But, these model extensions quickly increase the computational complexity of the algorithms to detect those clusters.

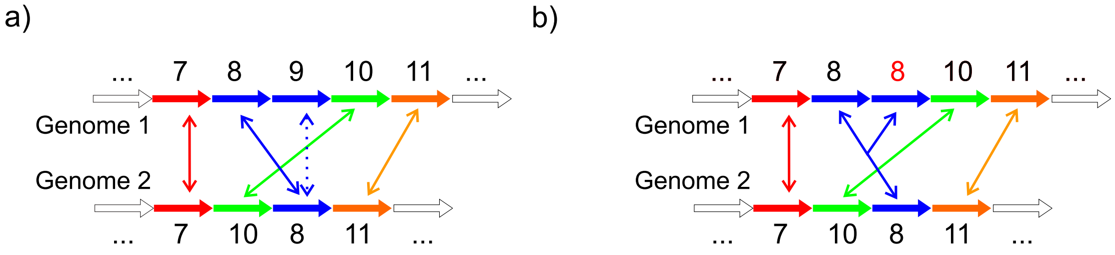


Figure 3.4: Representing paralogs. If a gene duplication (a) is detected (genes 8, 9), we represent both duplications (b) by the same number (8).

With respect to both speed and accuracy, in this section we present our model for gene clusters, which tries to keep a balance between computational complexity and biological relevance of the developed model. Incorporating the notion of paralogs, our extended model is able to solve the problems arising from the preprocessing of the genomes (see Section 3.3.3). Whenever a gene duplication is observed, it is no longer the goal to identify which of the duplicated genes corresponds to a certain homolog in another genome, instead each paralog is denoted by the same identifier (number), see Figure 3.4. Allowing for possible multiplicities, obviously the genomes are no longer permutations but sequences (strings) of their genes. To this end, the modelling of gene clusters as common intervals over permutations has to be generalized to strings.

3.4.1 Definitions

Given a string S over the finite alphabet of integers $\Sigma := \{1, ..., m\}$, $|S| \leq n$ is the length of S, $S[i]$ refers to the ith character of S, and $S[i, j]$ is the substring of S that starts with the ith and ends with the jth character of S. For convenience it will always be assumed for a string S that $S[0]$ and $S[|S| + 1]$ are characters not occurring elsewhere in S, so that border effects can be ignored when speaking of the left or right neighbor of a character in S. In our application of comparative genomics, the characters from Σ represent the genes. We will refer to S as a genome or a string interchangeably.

Definition (character set). Given a string S, the *character set* of a substring $S[i, j]$ is defined by

$$\mathcal{CS}(S[i, j]) := \{S[k] \mid i \leq k \leq j\} \subseteq \Sigma.$$

A character set represents the set of all genes occurring in a given substring of a genome, where the order and the number of occurrences of paralogous copies of a gene is irrelevant. Observe that if S is a permutation $\pi = S$ then $\mathcal{CS}(S[i,j]) = \pi([i,j])$.

Definition ($\mathcal{CS}$-location, maximal). Given a string S over an alphabet Σ and a subset $C \subseteq \Sigma$, the interval $[i,j]$ is a *$\mathcal{CS}$-location* of C in S if and only if $\mathcal{CS}(S[i,j]) = C$. A $\mathcal{CS}$-location $[i,j]$ of C is *left-maximal* if $S[i-1] \notin C$, it is *right-maximal* if $S[j+1] \notin C$, and it is *maximal* if it is both left- and right-maximal.

A $\mathcal{CS}$-location of a subset C of Σ represents a contiguous region in a genome that contains exactly the genes contained in C, allowing for possible multiplicities. Note that C has a *maximal* $\mathcal{CS}$-location in S if and only if C has a $\mathcal{CS}$-location in S.

Definition (common $\mathcal{CS}$-factor of k strings). Given a collection of k strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$ over an alphabet Σ, a subset $C \subseteq \Sigma$ is a *common $\mathcal{CS}$-factor* of $\mathcal{S}$ if and only if C has a $\mathcal{CS}$-location in each S_l, $1 \leq \ l \leq k$.

A common $\mathcal{CS}$-factor of k genomes represents a gene cluster that occurs in each of the k genomes. This concept is similar to a common interval of k permutations, but it allows the presence of paralogous genes in the genomes and particularly within a gene cluster.

Example. Let $S_1 = (3,5,7,9,7,4,6,8,1,2)$ with $\Sigma = \{1,2,\cdots,9\}$. The character set $\mathcal{CS}(S_1[3,5])$ of the substring $S_1[3,5]$ is the set $\{7,9\}$. The set $C = \{7,9\}$ has three different $\mathcal{CS}$-locations in S_1: $[3,4]$, $[4,5]$, and $[3,5]$, where only $[3,5]$ is maximal. With $S_2 = (5,2,7,4,9,7,2)$, C is a common $\mathcal{CS}$-factor of the two strings S_1 and S_2, because C has also a $\mathcal{CS}$-location in S_2: $[5,6]$.

These definitions motivate the following two problems, where the solution of Problem 2 implies a solution of Problem 1:

Problem 1. Given a collection of k strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$, find all its common $\mathcal{CS}$-factors.

Problem 2. For each common $\mathcal{CS}$-factor of $\mathcal{S}$, find all its maximal $\mathcal{CS}$-locations in each of the S_l, $1 \leq \ l \leq k$.

Unfortunately, the presented algorithms for detecting gene clusters, defined as common intervals on permutations (see Section 3.3.2), are not straightforwardly extendable to detect common $\mathcal{CS}$-factors. The reason is the counting function of Equation 3.1, which requires that each number in the represented genome occurs exactly once.

Therefore, in the following sections, we present several different approaches to solve Problem 1 and Problem 2 as efficiently as possible for two strings, finally presenting a worst case optimal quadratic time and linear space algorithm. Afterwards, we show how to generalize the algorithms to any given number of strings. For all algorithms the input is always given by two strings S_1 and S_2 of numbers from the alphabet Σ. The output for each common $\mathcal{CS}$-factors is then given as a pair of its $\mathcal{CS}$-locations $([i,j]_{S_1},[i',j']_{S_2})$, where $[i,j]_{S_l}$ denotes a $\mathcal{CS}$-location $[i,j]$ of sequence S_l.

3.4.2 First Simple Algorithms

A first non-trivial solution to find all common $\mathcal{CS}$-factors of two strings was the Two Stack Algorithm, shown in Algorithm 1.

The general idea of this algorithm is, that for fixed starting positions i in S_1 and i' in S_2 one alternately extends the substring $S_1[i,j]$ and $S_2[i',j']$ to the right, while keeping track of the characters that are missing in the substring of the other sequence. Therefore, the stack ST_1 is used to store all characters occurring in $S_2[i',j']$ and not in $S_1[i,j]$, and ST_2 visa versa. To store the information that a character $c \in \Sigma$ occurs in the substring $S_1[i,j]$ (resp. $S_2[i',j']$) read so far, a bitvector OCC_1 (OCC_2) of length Σ is used, such that $OCC_1[c] = 1$ ($OCC_2[c] = 1$) if c occurs in $S_1[i,j]$ ($S_2[i,j]$). The boolean *phase* indicates whether S_1 (*phase*=`false`) or S_2 (*phase*=`true`) is being extended. Since the stacks contain only the characters that are present in exactly one of the substrings read so far, a common $\mathcal{CS}$-factor of the two strings is found only if both stacks are empty (see example in Figure 3.5).

For the analysis of this algorithm observe that there are $|S_1| \cdot |S_2|$ different pairs of start indices i and i' for the outer `while`-loop starting in line 5 of Algorithm 1. The body of this `while`-loop is executed at most $|S_1| + |S_2|$ times, since in each iteration, j or j' is incremented by one or more if the `while`-loops in lines 15 or 18 are executed. With exception of the `while`-loop starting in line 10, all other operations can be performed in constant time. To show that for a fixed pair of start indices i and i' the body of the `while`-loop starting in line 10 is executed at most $|S_1|$ times, observe that this loop is executed at most once for each element pushed on the stack ST_1. This holds since each character in S_1 can be pushed on the stack only once. The same argumentation holds for the `while`-loop not stated explicitly if $phase = false$.

Summarizing we can state that with n denoting the length of the longer sequence, the Two Stack Algorithm outputs all common $\mathcal{CS}$-factors of S_1 and S_2 in $O(n^3)$ time.

Algorithm 1 Two Stack Algorithm

```
for $i = 1, \dots, |S_1|$ do
  for $i' = 1, \dots, |S_2|$ do
    $OCC_1[c] \leftarrow 0$, $OCC_2[c] \leftarrow 0 \; \forall \; c \in \Sigma$,    clear stacks $ST_1$, $ST_2$
    initialize: $j \leftarrow i$,    $j' \leftarrow i'$,    $phase \leftarrow$ true,    $OCC_1[S_1[j]] \leftarrow 1$,    $push(ST_1, S_1[j])$
    while $j \leq |S_1|$ and $j' \leq |S_2|$ and $[i, j]_{S_1}$, $[i', j']_{S_2}$ are left-maximal do
      if $phase$ then
        {walking in $S_2$:}
        if $S_2[j'] = peek(ST_1)$ then
          $OCC_2[S_2[j']] \leftarrow 1$
          while $notempty(ST_1)$ and $OCC_2[peek(ST_1)] = 1$ do
            $pop(ST_1)$
          end while
          if $empty(ST_1)$ then
            if $empty(ST_2)$ then
              while $OCC_1[S_1[j + 1]] = 1$ do
                $j \leftarrow j + 1$
              end while
              while $OCC_2[S_2[j' + 1]] = 1$ do
                $j' \leftarrow j' + 1$
              end while
              output the pair $([i, j]_{S_1}, [i', j']_{S_2})$
              $j' \leftarrow j' + 1$
              if $S_2[j'] \leq |S_2|$ then
                $OCC_2[S_2[j']] \leftarrow 1$
                $push(ST_2, S_2[j'])$
                $j \leftarrow j + 1$
                $phase \leftarrow not(phase)$
              end if
            else
              $phase \leftarrow not(phase)$
              $j \leftarrow j + 1$
            end if
          else
            $j' \leftarrow j' + 1$
          end if
        else
          if $OCC_1[S_2[j']] = 0$ and $OCC_2[S_2[j']] = 0$ then
            $push(ST_2, S_2[j'])$
          end if
          $OCC_2[S_2[j']] \leftarrow 1$, $j' \leftarrow j' + 1$
        end if
      else
        {walking in $S_1$ (symmetric to previous block):}
        substitute: $j \leftrightarrow j', i \leftrightarrow i', ST_1 \leftrightarrow ST_2, C_1 \leftrightarrow C_2, S_1 \leftrightarrow S_2$
      end if
    end while
  end for
end for
```

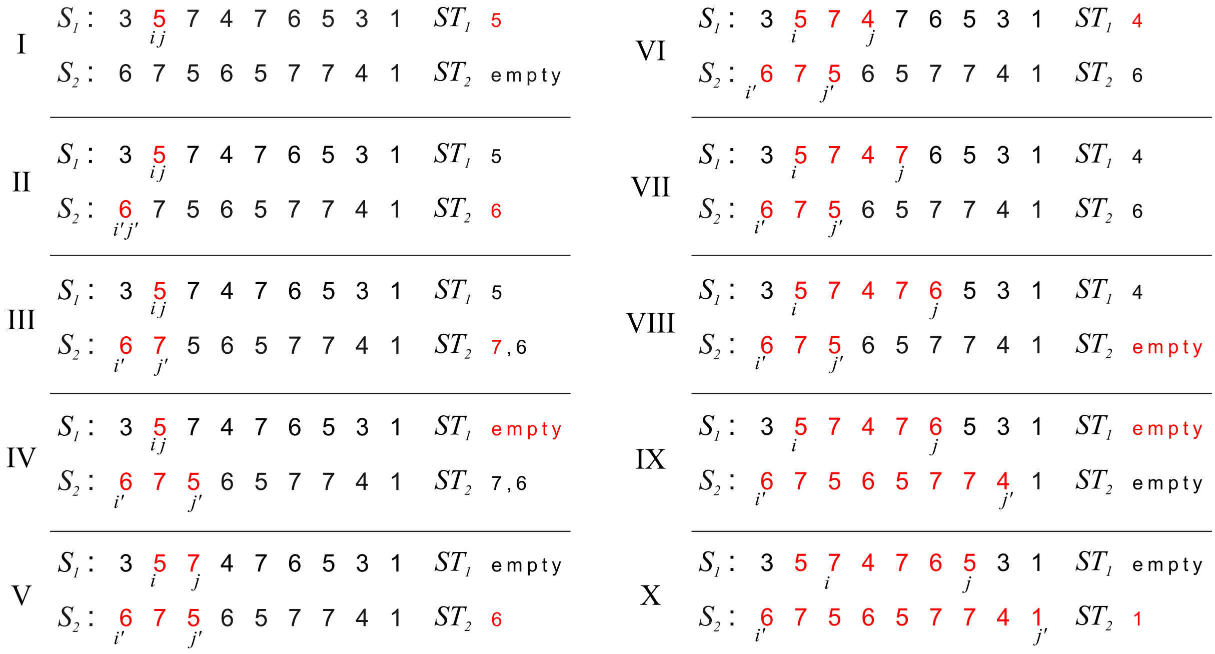

I
S1: 3 5 7 4 7 6 5 3 1 ST1 5
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 empty
II
S1: 3 5 7 4 7 6 5 3 1 ST1 5
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 6
i' j'
III
S1: 3 5 7 4 7 6 5 3 1 ST1 5
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 7,6
i' j'
IV
S1: 3 5 7 4 7 6 5 3 1 ST1 empty
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 7,6
i' j'
V
S1: 3 5 7 4 7 6 5 3 1 ST1 empty
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 6
i' j'
VI
S1: 3 5 7 4 7 6 5 3 1 ST1 4
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 6
i' j'
VII
S1: 3 5 7 4 7 6 5 3 1 ST1 4
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 6
i' j'
VIII
S1: 3 5 7 4 7 6 5 3 1 ST1 4
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 empty
i' j'
IX
S1: 3 5 7 4 7 6 5 3 1 ST1 empty
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 empty
i' j'
X
S1: 3 5 7 4 7 6 5 3 1 ST1 empty
i j
S2: 6 7 5 6 5 7 7 4 1 ST2 1
i' j'

Figure 3.5: Starting with $i = 2$ in S_1, line I shows the initial configuration of the stacks ST_1 and ST_2. In lines II and III, during the increase of j', the characters not occurring in $S_1[i, j]$ are pushed onto the stack ST_2. In line IV, the character on the top of ST_1 is read in $S_2[i', j']$. Then ST_1 is updated, such that all characters on top of ST_1 are removed from ST_1 if they occur in $S_2[i', j']$ (here only '5'). Since, ST_1 is now empty, the substring to be increased changes from S_2 to S_1. Now the characters read during the increase of j and not occurring in $S_2[i', j']$ are pushed onto ST_1 (lines V-VII). In line VIII, the character on top of ST_2 is read in S_1, ST_2 is updated and since it is now empty the increasing substring changes to S_2. Finally, in line IX the substring $S_2[i', j']$ contains all characters from $S_1[i, j]$ and both stacks are empty. Then both substrings are tested for right-maximality ($S_1[i, j]$ is not right-maximal) and the common $\mathcal{CS}$-factor $([2, 7]_{S_1}, [1, 8]_{S_2})$ is reported. Afterwards (in line X), the procedure continues with the search for further common $\mathcal{CS}$-factors until the end of S_1 or S_2 is reached.

Shifting Algorithm

A second less complex algorithm solving our problems in cubic time, is the Shifting Algorithm (see Algorithm 2). Here, the basic idea is that for a fixed maximal $\mathcal{CS}$-location in S_1, one shifts through S_2, searching for maximal $\mathcal{CS}$-locations of the same character set as the $\mathcal{CS}$-location in S_1.

Algorithm 2 Shifting Algorithm

```
 1: $OCC_1[c] \leftarrow 0$ for each character $c$ of $S_1$ in $\Sigma$
 2: $OCC_2[c] \leftarrow 0$ for each character $c$ of $S_2$ in $\Sigma$
 3: for $i = 1, \ldots, |S_1|$ do
 4:   $j \leftarrow i$
 5:   $OCC_1[S_1[i]] \leftarrow 1$
 6:   while $j \leq |S_1|$ and $[i, j]_{S_1}$ is left-maximal do
 7:     $OCC_1[S_1[j]] \leftarrow 1$
 8:     if $[i, j]_{S_1}$ is right-maximal then
 9:       $i' \leftarrow 1$, $j' \leftarrow 1$
10:       while $j' \leq (|S_2| + 1)$ do
11:         if $OCC_1[S_2[j']] = 1$ then
12:           $OCC_2[S_2[j']] \leftarrow 1$
13:           $j' \leftarrow j' + 1$
14:         else
15:           if $|OCC_1| = |OCC_2|$ then
16:             output the pair $([i, j]_{S_1}, [i', j' - 1]_{S_2})$
17:           end if
18:           while $i' \leq j'$ do
19:             $OCC_2[S_2[i']] \leftarrow 0$
20:             $i' \leftarrow i' + 1$
21:           end while
22:           $j' \leftarrow j' + 1$
23:         end if
24:       end while
25:     end if
26:     $j \leftarrow j + 1$
27:   end while
28: end for
```

For a fixed position i, the substring $S_1[i, j]$ is extended to the right as long as it is left-maximal. Again, $OCC_1[c] = 1$ indicates that character c occurs in the substring of S_1 read the so far, and additionally $|OCC_1|$ counts the different characters (number of ones in OCC_1). For each left-maximal substring of S_1 that is also right-maximal (which is explicitly tested in line 8), an iterated shift of i' and j' through S_2 is performed, where $[i', j']_{S_2}$ is a $\mathcal{CS}$-location candidate for a common $\mathcal{CS}$-factor.

The shifting is an alternating application of two steps (see Figure 3.6). The first step

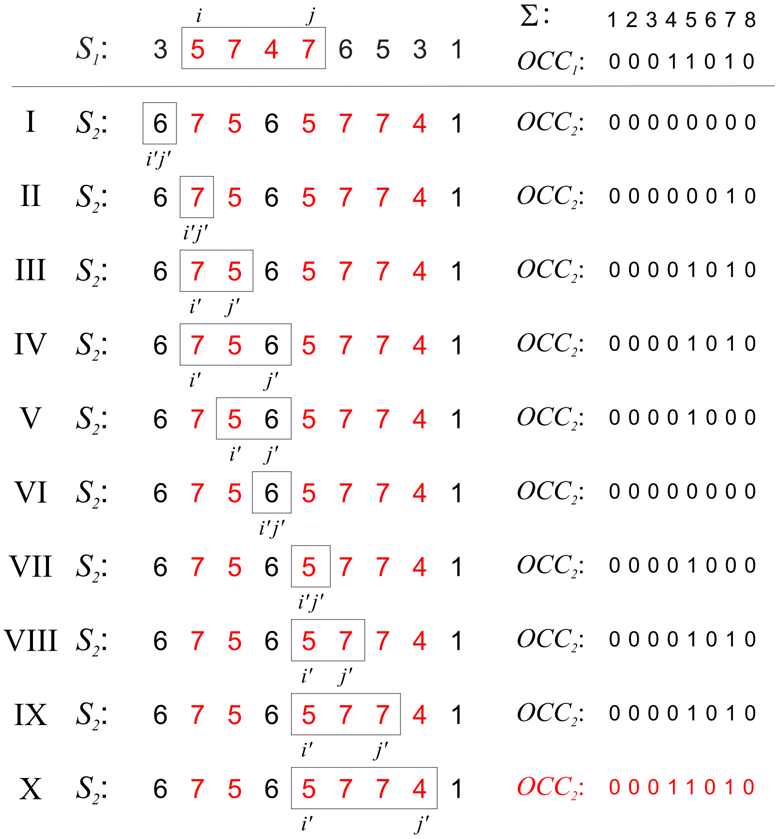

i j
S1: 3 5 7 4 7 6 5 3 1
Σ: 1 2 3 4 5 6 7 8
OCC1: 0 0 0 1 1 0 1 0
I S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 0 0 0 0
i'j'
II S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 0 0 1 0
i'j'
III S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 1 0 1 0
i' j'
IV S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 1 0 1 0
i' j'
V S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 1 0 0 0
i' j'
VI S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 0 0 0 0
i'j'
VII S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 1 0 0 0
i'j'
VIII S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 1 0 1 0
i' j'
IX S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 0 1 0 1 0
i' j'
X S2: 6 7 5 6 5 7 7 4 1 OCC2: 0 0 0 1 1 0 1 0
i' j'

Figure 3.6: Shifting through S_2. The extension step starts, once a character from $S_1[i,j]$ is found in S_2 (line II). The extension continues until the first character not in $S_1[i,j]$ is detected (line IV). If now the character set of $S_1[i,j]$ and $S_2[i,j-1]$ are equal, a common $\mathcal{CS}$-factor is found (line X). In lines V-VII the interval boundaries are moved to the new start position behind the last index of the substring processed before.

(lines 12 and 13) extends the substring $S_2[i', j']$ to the right, as long as its character set contains only characters also occurring in $S_1[i, j]$. The characters read are marked in OCC_2. In the second step, once a character not in $S_1[i, j]$ is read, we know that $S_2[i', j' - 1]$ was right-maximal and if $S_2[i', j' - 1]$ contains all characters from $S_1[i, j]$ (test in line 15), we have detected a common $\mathcal{CS}$-factor. Since $S_2[i', j' - 1]$ was right-maximal, we cannot find another common $\mathcal{CS}$-factor by extending $S_2[i', j' - 1]$ to the right. So, the left interval boundary i' is increased to the next possible starting position for a new $\mathcal{CS}$-location $(j' + 1)$ establishing again the left-maximality. During the increase of i', all characters read are removed from OCC_2, such that $|OCC_2| = 0$, when $i' = j' + 1$.

Obviously, this algorithm runs in $O(n^3)$ time, since there are $\binom{n}{2}$ substrings in S_1, for which a linear time shift for i' and j' through S_2 has to be performed. This yields the total time complexity of $O(n^3)$.

3.4.3 An Algorithm Using the Naming Technique

In 2003, Amir *et al.* [4] developed an algorithm for efficient text fingerprinting by sets of symbols, originally addressing a problem in natural language processing. It turned out that these sets of symbols are exactly the character sets from the definition of common $\mathcal{CS}$-factors, and therefore this algorithm can be applied to solve our problems as well.

On a high level, this algorithm can be described as follows: Enumerate for the strings S_1 and S_2 all occurring character sets and assign them a name representing these sets. Common $\mathcal{CS}$-factors are all character sets whose names are found at least once in each of the two strings.

For given a string S_1 of length n over alphabet Σ (for simplicity, it is assumed that $|\Sigma|$ is a power of 2), the algorithm performs $|\Sigma|$ iterations, where in the k-th iteration all the substrings of S_1 are enumerated, whose character sets are of size k (see Figure 3.7). Therefore, a maximal substring $S_1[a, b]$ is maintained, and similar to OCC in the previous algorithm, a binary array LIFE$[1..|\Sigma|]$ is used, where LIFE$[i]$ is 1 if character i is present in the character set of $S_1[a, b]$, and 0 otherwise.

Initially, the substring contains the longest prefix of S whose character set has size k. Then, the substring is modified in the following way: First, b is increased until the character set of $S_1[a, b]$ has size $k + 1$, then the left boundary a is increased until the character set of $S_1[a, b]$ has size k, and finally b is increased again, as long as the character set remains with size k. This substring movement procedure is repeated until the end of S_1 is reached. By this procedure, easily the character sets of all substrings of S_1 can be found in $\Theta(n|\Sigma|)$ time, but the same set may be found several times. The rest of the algorithm deals with the identification of multiple occurrences of the same character set. Therefore, some additional data structure is used.

A subarray LIFE$[i2^l + 1..(i + 1)2^l]$ of LIFE $(0 \leq l \leq \log|\Sigma|,\ 0 \leq i \leq |\Sigma|/2^l - 1)$ will

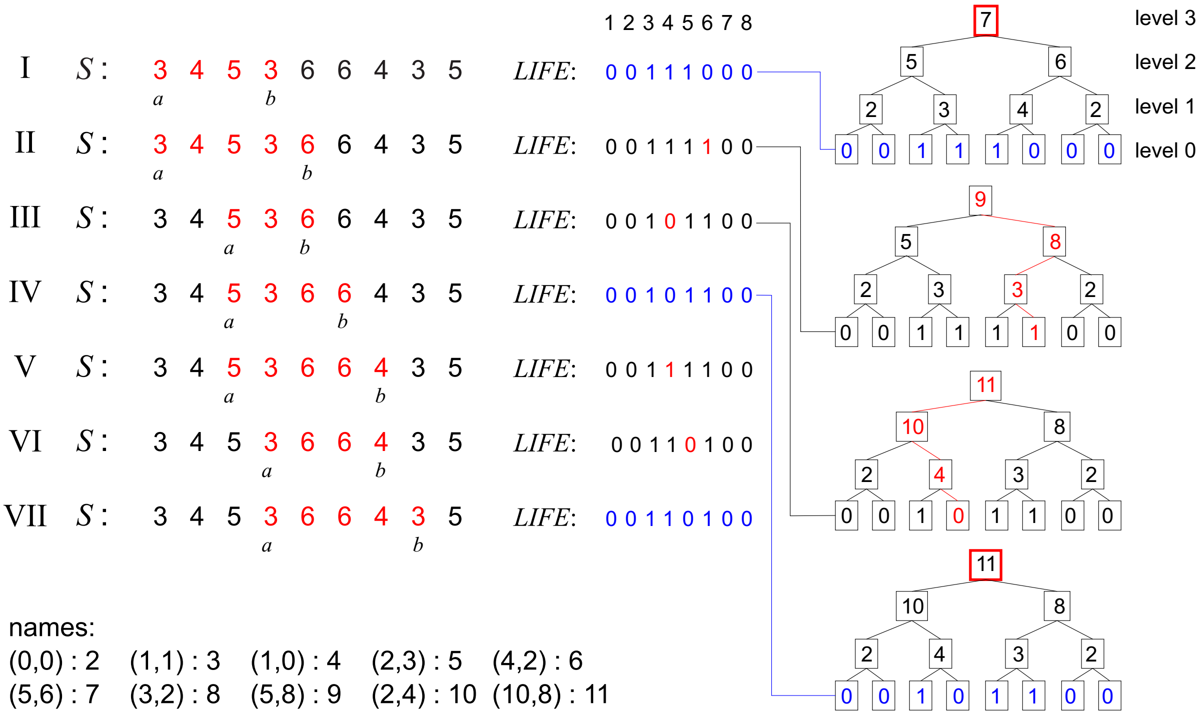

1 2 3 4 5 6 7 8
I S: 3 4 5 3 6 6 4 3 5 LIFE: 0 0 1 1 1 0 0 0
a b
II S: 3 4 5 3 6 6 4 3 5 LIFE: 0 0 1 1 1 1 0 0
a b
III S: 3 4 5 3 6 6 4 3 5 LIFE: 0 0 1 0 1 1 0 0
a b
IV S: 3 4 5 3 6 6 4 3 5 LIFE: 0 0 1 0 1 1 0 0
a b
V S: 3 4 5 3 6 6 4 3 5 LIFE: 0 0 1 1 1 1 0 0
a b
VI S: 3 4 5 3 6 6 4 3 5 LIFE: 0 0 1 1 0 1 0 0
a b
VII S: 3 4 5 3 6 6 4 3 5 LIFE: 0 0 1 1 0 1 0 0
a b
names:
(0,0) : 2 (1,1) : 3 (1,0) : 4 (2,3) : 5 (4,2) : 6
(5,6) : 7 (3,2) : 8 (5,8) : 9 (2,4) : 10 (10,8) : 11
level 3
level 2
level 1
level 0
7
5 6
2 3 4 2
0 0 1 1 1 0 0 0
9
5 8
2 3 3 2
0 0 1 1 1 1 0 0
11
10 8
2 4 3 2
0 0 1 0 1 1 0 0
11
10 8
2 4 3 2
0 0 1 0 1 1 0 0

Figure 3.7: Enumerating maximal substrings of three characters. Starting from the first maximal substring of three characters $[1, 4]$ in line I, b is extended to 5, reading the first character not in $[1, 4]$ (line II). Then a is incremented (to position 3) until the character set of the remaining substring is of size 3 again. To ensure right-maximality, in a third step (line IV) b is incremented as long as no new character is read, resulting in the new maximal substring $[3, 6]$ containing three different characters. These three steps are repeated until the end of S is reached. In the center, for each substring the corresponding array LIFE is shown. The red numbers indicate the changed positions, compared to the previous line. The blue numbers represent the maximal substrings of three different characters, which have to be collected and named. The naming of each LIFE array is shown on the right. The name on the upper level is the representation for each character set of the corresponding substring. Here, the numbers 7 and 11 represent (name) the character sets to be enumerated in lines I and IV, respectively.

be called a *block of level l*. The main idea of the algorithm is assigning a *name* for each block in LIFE in all the configurations of LIFE. The naming is done in a way that ensures that two blocks of the same level with the same content get assigned the same name. In particular, in all the maximal locations of some character set C, the names assigned to the entire array (i.e., to the block of level $\log|\Sigma|$) are equal.

The naming is performed in the following way: Consider the initial configuration of LIFE (i.e., the one that corresponds to the first substring). The name of a block of level 0 is the corresponding value in LIFE. Now, suppose that names for all the blocks of level $l-1$ are assigned and the names to the blocks of level l are assigned from left to right. A block of level l is composed of two blocks of level $l-1$. Suppose that the names of these blocks are x and y. If the pair (x, y) appeared previously, then the name of the current block is the name that was assigned to the pair (x, y). Otherwise the minimum unused name is assigned to the pair (x, y), and also to the current block. See Figure 3.7 for an example.

After each movement of the interval, two positions in the array LIFE are changed: In one position, a 1 is changed into 0, and in a second position, a 0 is changed into 1. After each of these changes, the algorithm updates the names of the blocks. This can be done efficiently as only one block changes its name in each level.

Checking whether a pair (x, y) appeared previously can be done in $O(\log n)$ time using balanced binary search trees. Thus, creating the names for the first interval takes $O(|\Sigma|\log|\Sigma|\log n)$ time, and updating the names after each interval movement takes $O(\log|\Sigma|\log n)$ time. Therefore, the resulting overall time complexity of this algorithm is $O(n|\Sigma|\log|\Sigma|\log n)$.

To find common $\mathcal{CS}$-factors of two strings, the algorithm is applied to a second string using the same naming as in the previous one. Finally, names of character sets occurring in both strings are common $\mathcal{CS}$-factors. In contrast to applications in natural language processing where this algorithm was designed for, in our application for gene cluster detection the alphabet size is closely related to the length of the genomes. We always assume that $|\Sigma| \in \Theta(n)$, yielding a total time complexity of $O(n^2\log^2 n)$.

In [18], we showed how to further reduce the time complexity of this algorithm to $\Theta(n|\Sigma|(\log(|\Sigma|/\log n) + 1))$ by improving the technique of assigning the names to each level of name blocks. The general idea of this improvement is to use a preprocessing step, calculating all possible names for a fixed level of the name representation, allowing a constant time look-up for a required name at runtime. Additionally, by using an efficient representation of these names, the time complexity to create the name for a character set of a fixed size at a fixed starting position can be further reduced from $\Theta(\log|\Sigma|)$ to $\Theta(\log(|\Sigma|/\log n) + 1)$. In the simplest case, this efficient representation of a name for a character set is the number corresponding to its binary value in LIFE.

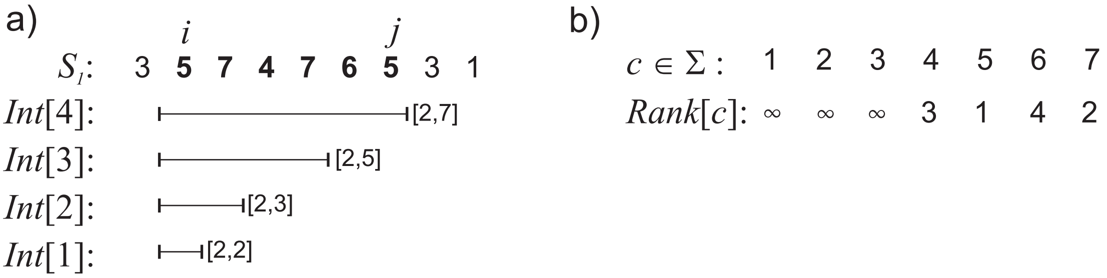


Figure 3.8: Preprocessing of $S_1 = (3, 5, 7, 4, 7, 6, 5, 3, 1)$. For a fixed position $i = 2$ of S_1, the longest maximal interval is $[2, 7]$ (a). For each character in this interval, its rank is assigned in the left-to-right order of their first occurrence (b). Afterwards, for each finite rank r the interval $Int[r]$ denotes the maximal substring in S_1 starting at position i and containing only characters of rank $\leq r$.

3.4.4 A First Quadratic Time Algorithm

The algorithm developed by Didier in 2003 [19], is related to the shifting algorithm (Algorithm 2) in Section 3.4.2, since it also starts with the detection of maximal substrings in S_1 and then, in a preprocessing of S_2, builds a data structure allowing the efficient detection of substrings in S_2 containing the same character set.

For a fixed left index i in S_1, Didier's algorithm scans the suffix of S_1 starting at position i for maximal intervals. During the scan, to each character $c \in \Sigma$ its *rank* $Rank[c]$ is associated, i.e. the position of c in the list of *different* characters as they occur in left-to-right order in $[i, j]_{S_1}$, and ∞ if c does not occur in the interval. The corresponding maximal interval $[i, j]_{S_1}$ for a finite rank r is stored in $Int[r]$, see Figure 3.8.

For each fixed position i in S_1 a preprocessing of S_2 has to be performed. In the preprocessing, for each position k of S_2 the following tables are filled (see Figure 3.9):

1. *LNeighbor*, containing for any position k of finite rank r the greatest position smaller than k with rank $r + 1$, if it exists. Otherwise $LNeighbor[k]$ is undefined.

2. *RNeighbor*, containing for any position k of finite rank r the smallest position greater than k with rank $r + 1$, if it exists. Otherwise $RNeighbor[k]$ is undefined.

3. *LDistance* containing for any position k that has a left neighbor k_l, the maximum rank of the characters occurring in $S_2[k_l, k]$ and ∞ otherwise.

4. *RDistance*, containing for any position k that has a right neighbor k_r, the maximum rank of the characters occurring in $S_2[k, k_r]$ and ∞ otherwise.

5. *Succ*, containing the successor of the k-th position, defined as $LNeighbor[k]$ if

a)

S_2:	6	7	5	6	5	7	7	4	1
Rank:	4	2	1	4	1	2	2	3	∞
LNeighbor:	-	-	2	-	2	-	-	4	-
LDistance:	∞	∞	2	∞	4	∞	∞	4	∞
RNeighbor:	-	8	6	-	6	8	8	-	∞
RDistance:	∞	4	4	∞	2	3	3	∞	∞
Succ:	-	8	2	-	6	8	8	4	∞
MaxInt:	[1,8]	[2,3]	[3,3]	[1,8]	[5,5]	[5,7]	[5,7]	[5,8]	-

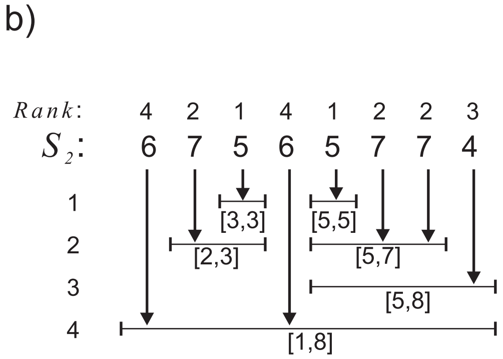


Figure 3.9: Preprocessing of $S_2 = (6, 7, 5, 6, 5, 7, 7, 4, 1)$ with S_1 and $i = 2$ from the previous example (Figure 3.8). For each position k in S_2, the displayed arrays are computed (a). In (b) the hierarchical organization of the intervals $MaxInt$ is shown.

$LDistance[k] \leq RDistance[k]$ and $RNeighbor[k]$ otherwise. If $LDistance[k] = RDistance[k] = \infty$, k has no successor.

6. $MaxInt$, containing for any position k the maximal interval $[x, y]_{S_2}$, such that $x \leq k \leq y$ and each character $c \in [x, y]_{S_2}$ has a rank $\leq Rank[S_2[k]]$.

Since a large part of the algorithmic work is done in the preprocessing, the final step of collecting the common $\mathcal{CS}$-factors is straight-forward (see Algorithm 3).

Algorithm 3 Reporting Intervals

```
for all positions $k$ with $Rank[S_2[k]] = 1$ do
  $LBound \leftarrow k$, $RBound \leftarrow k$
  $j \leftarrow k$
  repeat
    $LBound \leftarrow \min(j, LBound)$
    $RBound \leftarrow \max(j, RBound)$
    if $[LBound, RBound] \subseteq MaxInt[j]$ then
      output $([i, Int[Rank[S_2[j]]]]_{S_1}, MaxInt[j]_{S_2})$
    end if
    mark $j$ such that it is not traversed again
    $j \leftarrow Succ[j]$
  until $j$ is empty or $j$ is marked
end for
```

For each starting position k in S_2 with $Rank(S_2[k]) = 1$, an interval $[BoundL, BoundR]_{S_2}$ is created, tracking the region in S_2 covered by the characters being read along the path of positions j succeeding k. If $[BoundL, BoundR]_{S_2} \subseteq MaxInt[j]$, then this region is a sub-interval containing all characters from the maximal interval of the characters read so far, i.e. a common $\mathcal{CS}$-factor $([i, Int[Rank[S_2[j]]]]_{S_1}, MaxInt[j]_{S_2})$ is detected. Finally, the interval $MaxInt[j]$ is marked in order not to be reported again, and j becomes its successor $Succ[j]$. If the successor is empty or marked before, the algorithm continues with the next position k with $Rank(S_2[k]) = 1$.

The analysis of Algorithm 3 which reports the intervals is rather obvious. Since due to the marking, each position k in S_2 is read at most once, the algorithm runs in $O(n)$ time. Some more attention needs to be directed to the preprocessing of S_2. Here, observe that $LNeighbor$ and $RNeighbor$ can be created in linear time, as well as $Succ$ once the distance tables are filled. More difficult is the computation of the tables $RDistance$ and $LDistance$. To compute $RDistance$, Didier uses a stack algorithm to store each position of S_2 which qualifies as a candidate for a right neighbor of a newly read position of S_2. On the stack, the stored elements appear ordered according to their rank appearing in right-to-left order in S_2. Therefore, the value of $RDistance[k]$ can be found as the rank of greatest position in the stack that is smaller or equal to $RNeighbor[k]$. To find the greatest position in the stack, Didier uses a binary search ($LeftDistance$ is computed using the same strategy). Since the number of elements on the stack can be as large as the number of characters in S_2, and the binary search can be performed in $O(\log n)$ time, and the complexity for filling the distance tables is $O(n \log n)$.

Since the table $MaxInt$ is computed again in linear time, using a simple stack algorithm, the total time complexity for the preprocessing and therefore for the whole algorithm is $O(n \log n)$ for a fixed position i in S_1. This yields the total time complexity of $O(n^2 \log n)$ to compute all common $\mathcal{CS}$-factor of S_1 and S_2.

In [18], we presented a modified version of this algorithm reducing the time complexity to $O(n^2)$ while staying still linear in space consumption. The general idea of this improvement is the more efficient calculation of the tables $RDistance$ and $LDistance$ (see Figure 3.9). Recall that the entry $RDistance[i]$ ($LDistance[i]$) denotes the maximal occurring rank in the interval $[i, j]_{S_2}$ where j is the right (left) neighbor of i. To compute the tables $RDistance$ and $LDistance$, Didier uses a stack to store all possible candidates for right neighbors and performs a binary search on the elements on the stack to find the character of maximal rank in $[i, j]_{S_2}$, once the right neighbor j is determined. This binary search for the maximum rank takes $\log s$ time, where s is the number of elements on the stack, and in the worst-case this number can be as large as n. A more detailed analysis of this search revealed that it could also be formulated as a Range Maximum Query (RMQ) on the ranks of the interval $[i, j]_{S_2}$. In [9], Bender and Farach-Colton formulated an algorithm allowing to answer an RMQ in constant time after a linear time preprocessing of the input sequence. Thus, performing an RMQ preprocessing on S_2 and replacing the binary search by an RMQ on the interval $[i, j]_{S_2}$, the over all preprocessing time for S_2 (tables and RMQ) reduces to $O(n)$ and allows the detection of all common $\mathcal{CS}$-factors with all their $\mathcal{CS}$-locations in quadratic time using linear space.

(a) $POS[1] = 9$
$POS[2] = empty$
$POS[3] = 1, 8$
$POS[4] = 4$
$POS[5] = 2, 7$
$POS[6] = 6$
$POS[7] = 3, 5$

(b) $NUM(i, j)$:

$_i\backslash^j$	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
1	1	2	3	4	4	5	5	5	6
2		1	2	3	3	4	4	5	6
3			1	2	2	3	4	5	6
4				1	2	3	4	5	6
5					1	2	3	4	5
6						1	2	3	4
7							1	2	3
8								1	2
9									1

Figure 3.10: Preprocessing of $S_1 = (3, 5, 7, 4, 7, 6, 5, 3, 1)$ with $\Sigma = \{1, \ldots, 7\}$. (a) for each character $c \in \Sigma$, $POS[c]$ holds the positions at which c occurs in S_1; (b) the table NUM holding the values $|\mathcal{CS}(S_1[i, j])|$.

3.4.5 A Simpler Quadratic Time Algorithm

With the algorithm "Connecting Intervals" (CI) [64], we developed a simple algorithm that solves our problems in $\Theta(n^2)$ time requiring $\Theta(n^2)$ space. An overview of the algorithm is given in Algorithm 4.

In a preprocessing step, the algorithm constructs two simple data structures, illustrated in Figure 3.10. The first data structure, POS, contains for each character $c \in \Sigma$ a list $POS[c]$ that holds the positions of occurrence of c in string S_1 in ascending order, see Figure 3.10 (a). The second data structure, NUM, is a $|S_1| \times |S_1|$ table where entry $NUM(i, j)$ contains the number $|\mathcal{CS}(S_1[i, j])|$ of *different* characters in the substring $S_1[i, j]$ for each $1 \leq i \leq j \leq |S_1|$, see Figure 3.10 (b). Clearly, POS requires linear space and can be computed in linear time by a simple scan over S_1, while NUM requires $\Theta(n^2)$ space and its computation takes $\Theta(n^2)$ time.

On a high level, Algorithm CI can be described as follows (see Figure 3.11): For a fixed position i in S_2, while reading the substring of S_2 starting at that position, the observed characters are marked in S_1, and simultaneously maximal intervals of marked characters are tracked. This is iterated for all start positions i of substrings in S_2.

The maximal intervals of marked characters in S_1 are candidates for common $\mathcal{CS}$-factors with the current substring $S_2[i, j]$. It only needs to be tested (i) if the character set of a candidate interval coincides with that of $S_2[i, j]$, and (ii) if the substring $S_2[i, j]$ is a maximal $\mathcal{CS}$-location of its character set.

In fact, to test (i) it suffices to compare the number of different characters in the two substrings. We know that the maximal marked intervals in S_1 contain a subset of the characters in $S_2[i, j]$, hence if the character sets have equal size, they must be equal. The number of different characters in $S_2[i, j]$ can be tracked while reading the substring of

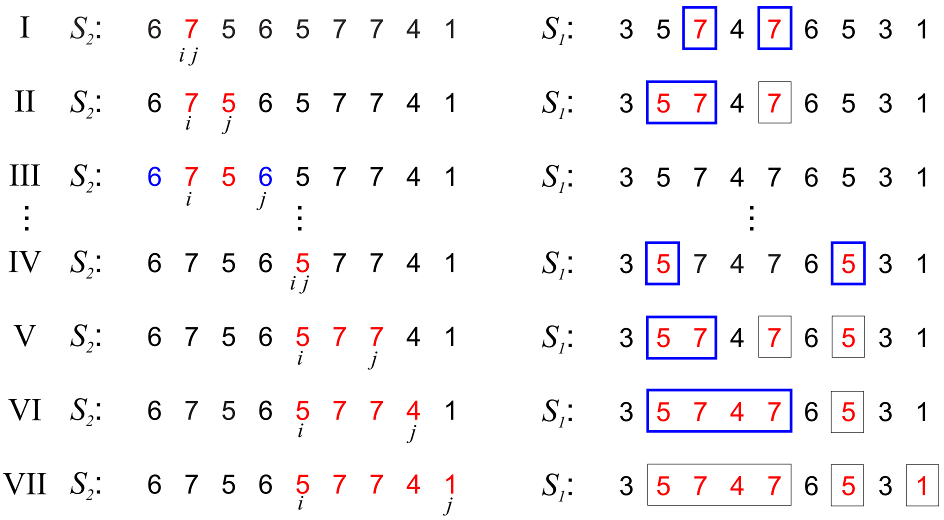

I S_2: 6 7 5 6 5 7 7 4 1 S_1: 3 5 7 4 7 6 5 3 1
i j
II S_2: 6 7 5 6 5 7 7 4 1 S_1: 3 5 7 4 7 6 5 3 1
i j
III S_2: 6 7 5 6 5 7 7 4 1 S_1: 3 5 7 4 7 6 5 3 1
i j
IV S_2: 6 7 5 6 5 7 7 4 1 S_1: 3 5 7 4 7 6 5 3 1
i j
V S_2: 6 7 5 6 5 7 7 4 1 S_1: 3 5 7 4 7 6 5 3 1
i j
VI S_2: 6 7 5 6 5 7 7 4 1 S_1: 3 5 7 4 7 6 5 3 1
i j
VII S_2: 6 7 5 6 5 7 7 4 1 S_1: 3 5 7 4 7 6 5 3 1
i j

Figure 3.11: Example for Algorithm CI. In lines I-III position $i = 2$ is fixed as the left end of the increasing interval $[i, j]$ of S_2. While moving j to the right, the observed characters are marked in S_1 (red color), and maximal intervals of marked characters are tracked (the boxes). If $[i, j]_{S_2}$ contains the same elements as a tracked interval, a common $\mathcal{CS}$-factor is detected (blue boxes). If, while moving j to the right, $[i, j]_{S_2}$ is no longer right-maximal (line III), the algorithm stops the movement of j and selects $i + 1$ as new fixed left end (skipped in the figure). In lines IV-VII, the procedure is illustrated for $i = 5$, where the two marked intervals $[2, 3]_{S_1}$ and $[5, 5]_{S_1}$ are merged in line VI.

Algorithm 4 Connecting Intervals (CI)

```
 1: preprocessing: build data structures $POS$ and $NUM$
 2: for $i = 1, \dots, |S_2|$ do
 3:    $OCC[c] \leftarrow 0$ for each character $c$ in $\Sigma$
 4:    $j \leftarrow i$
 5:    while $j \leq |S_2|$ and $[i, j]$ is left-maximal in $S_2$ do
 6:       $c \leftarrow S_2[j]$
 7:       $OCC[c] \leftarrow 1$
 8:       while $[i, j]$ is not right-maximal in $S_2$ do
 9:          $j \leftarrow j + 1$
10:       end while
11:       for each position $p$ in $POS[c]$ do
12:          mark position $p$ in $S_1$
13:          find the maximal interval $[start, end]$ of positions marked so far containing position $p$
14:          if $NUM(start, end) = |OCC|$ and $[start, end]$ is maximal in $S_1$ then
15:             output the pair $([i, j]_{S_1}, [start, end]_{S_2})$
16:          end if
17:       end for
18:       $j \leftarrow j + 1$
19:    end while
20: end for
```

S_2 starting at position i. (In Algorithm 4 we use a binary vector OCC plus a counter $|OCC|$ that counts the number of ones in OCC.) The number of different characters in a maximal marked interval in S_1 can be read from the table NUM that was computed in the preprocessing phase.

Test (ii) is performed implicitly by the way how the value of j is incremented and the `while`-loop starting in line 5 of Algorithm 4 is terminated. Clearly, during the process of increasing j, once the interval $[i, j]$ of S_2 is not left-maximal for some $j \geq i$ (i.e. $S_2[i-1] = S_2[j']$ for some $j' \in \{i, \dots, j\}$), it will never be left-maximal for any $j'' > j$. Hence it is a valid action to terminate the while loop as soon as $[i, j]$ of S_2 is not left-maximal, and left-maximality is guaranteed whenever the body of the `while`-loop is entered. Right-maximality is explicitly tested in line 8 of Algorithm 4. This can be done in constant time by testing if $OCC[S_2[j+1]] = false$.

For the analysis we have to show how the marking and tracking of maximal intervals in S_1 is performed. Obviously, marking the r occurrences of character $c = S_2[j]$ in S_1 is possible in $O(r)$ time using the list $POS[c]$. Further, if for each maximal interval of marked positions in S_1 the interval boundaries $[start, end]$ are stored at the left and right end of the interval, then it is easy to test, whenever a position p of S_1 is newly marked, if it connects to already existing intervals (ending at position $p-1$ or starting at position $p+1$ or both), and to increase these intervals by index p (if p connects to only one interval) or merge the two intervals (if p connects to two intervals). All this can be done in constant time for each newly marked position p of S_1.

The `for`-loop starting in line 2 of Algorithm 4 is executed $|S_2| \leq n$ times; and in the outer `while`-loop together with the `while`-loop in line 8, j is incremented at most $|S_1| \leq n$ times. More difficult is the analysis of the `for`-loop starting in line 11. Here, observe that due to the test for right-maximality in line 8, this `for`-loop is reached for each character $c = S_2[j]$ only once, and hence for each i the body of the loop is executed at most $\sum_{c \in \Sigma} |POS[c]| = |S_1| \leq n$ times, where $|POS[c]|$ is the number of occurrences of character c in S_1. Together with the preprocessing, this yields the overall $\Theta(n^2)$ time and space complexity. Due to the fact that the number of common $\mathcal{CS}$-factors can be as large as $n(n+1)/2$, e.g. assume $S_1 = S_2 = (1, 2, \ldots, n)$, this algorithm is time-optimal in the sense of worst case analysis.

3.4.6 Improving the Running Time in Practice

Regarding the application of the algorithms to biological data, there is an additional option of improving the runtime for Algorithm CI. If we bound the maximal size of the character set of a common $\mathcal{CS}$-factor (i.e. the maximal number of different genes in a gene cluster) to a fixed number s, Algorithm CI can be modified to report all common $\mathcal{CS}$-factors in $O(ns)$ time if the number of paralogs inside a $\mathcal{CS}$-location is small, compared to the number of different characters in the corresponding character set. This improvement can be achieved simply by terminating the extension of the interval $[i, j]_{S_2}$, once $|OCC|_{S_2} > s$ (see line 5 in Algorithm 4). The same technique is also applicable in the improved version of the fingerprinting algorithm described in Section 3.4.3. Here, the algorithm is applied only to all character sets of S_1 with at most s different characters, yielding an overall time complexity of $\Theta(ns(\log(|\Sigma|/\log n) + 1))$. For the quadratic time version of Didier's algorithm, the limitation of the maximal character set size has no effect on the running time, since for each fixed i in $[i, j]_{S_1}$, a linear time preprocessing for the RMQ has to be performed.

3.4.7 Generating Non-redundant Output

Independent from the selected algorithm, a common $\mathcal{CS}$-factors is always reported as a pair of its maximal $\mathcal{CS}$-locations $([i, j]_{S_1}, [i', j']_{S_2})$, leading to a large amount of redundancy for paralogous gene clusters. For example, given $S_1 = (1, 2, 3, 1, 2)$ and $S_2 = (1, 2, 4, 1, 2, 5, 1, 2)$, the algorithm outputs the $\mathcal{CS}$-locations for the common $\mathcal{CS}$-factor $\{1, 2\}$ in the following way:

$$([1,2],[1,2]), ([1,2],[4,5]), ([1,2],[7,8]), ([4,5],[1,2]), ([4,5],[4,5]), ([4,5],[7,8]).$$

A non-redundant output of the following form should be preferred, though:

$$S_1 : [1,2],[4,5] \quad - \quad S_2 : [1,2],[4,5],[7,8].$$

This output can be obtained by an efficient storage of common $\mathcal{CS}$-factors. Two additional tables LOC_1 and LOC_2, each of size $|S_1| \times |S_1|$, are used to store the lists of intervals for the common $\mathcal{CS}$-factors. Then, in a first step, the chosen algorithm is applied to S_1 as first *and* second input sequence, yielding the paralogous gene clusters within S_1. These are stored in LOC_1 such that if $[i',j']_{S_1}$ is contained in list $LOC_1(i,j)$, then $\mathcal{CS}(S_1[i',j']) = \mathcal{CS}(S_1[i,j])$, in the following way. Initially, all lists $LOC_1(i,j)$ are empty. Whenever a common $\mathcal{CS}$-factor with maximal $\mathcal{CS}$-locations $[i,j]_{S_1}$ and $[i',j']_{S_1}$, $i' \neq i$, of a paralogous cluster is detected, then the $\mathcal{CS}$-location $[i',j']_{S_1}$ is appended to the list in $LOC_1(i,j)$ and the interval $[i',j']_{S_1}$ is marked, so that it is not being tested again.

In the second step, the algorithm is applied to S_1 and S_2, detecting the orthologous gene clusters between these two genomes. Whenever a common $\mathcal{CS}$-factor with maximal $\mathcal{CS}$-locations $[i,j]_{S_1}$ and $[k,l]_{S_2}$ is found, the $\mathcal{CS}$-location $[k,l]_{S_2}$ is appended to $LOC_2(i,j)$. Finally, the output for each non-empty entry $LOC_2(i,j)$ is

$$S_1 : [i,j]_{S_1}, LOC_1(i,j) \quad - \quad S_2 : LOC_2(i,j).$$

3.4.8 Multiple Genomes

To solve Problems 1 and 2 for any given $k \geq 2$, the described algorithms can easily be extended to more than two strings. The general idea is that a set of characters $C \subseteq \Sigma$ is a common $\mathcal{CS}$-factor of k strings $\mathcal{S} = \{S_1, \ldots, S_k\}$ if and only if it is a (pairwise) common $\mathcal{CS}$-factor of a fixed string (w.l.o.g. S_1) and all other strings in $\mathcal{S}$. Therefore, the algorithm is applied to each pair of input strings (S_1, S_r) with $S_r \in \mathcal{S}$ and $1 \leq r \leq k$. Note that, in order to generate non-redundant output, S_1 is also compared to itself. Since the first input string is always S_1, the preprocessing step for the table NUM has to be performed only once. For each processed pair (S_1, S_r) of input strings, a table LOC_r is built.

The output is performed by testing for each non-empty field in $LOC_k(i,j)$ if the corresponding fields in $LOC_r(i,j)$ with $1 < r < k$ are also non-empty. If this is the case, the output is:

$$S_1 : [i,j]_{S_1}, LOC_1(i,j) \; - \; S_2 : LOC_2(i,j) \; - \; \ldots \; - \; S_k : LOC_k(i,j)$$

The k iterations of the algorithm lead to an overall worst case time and space complexity of $O(kn^2)$ for the comparing step as well as for the processing of the LOC tables to generate the output.

The use of a further $|S_1| \times |S_1|$ table $LAST$ allows to speed-up the test for non-empty $LOC_r(i,j)$ to constant time for each pair (i,j). The entry $LAST(i,j)$ refers during the iteration of r from 1 to k to the largest value $r' \leq r$ such that $LOC_{r'}(i,j)$ is non-empty. Initially, all entries in $LAST(i,j)$ refer to $LOC_1(i,j)$. If the algorithm finds a common $\mathcal{CS}$-factor with $\mathcal{CS}$-locations $[i,j]_{S_1}$ and $[k,l]_{S_r}$, then, before updating $LAST(i,j)$ to r, it tests if the old value $LAST(i,j) < r-1$. In this case, the pair $[i,j]$ is marked such that it will not be reported in the end. After all string pairs are processed, the output is written for each unmarked pair $[i,j]$ with $LAST(i,j) = k$.

If one is interested in solving only Problem 1, i.e. the detection of the common $\mathcal{CS}$-factors without the $\mathcal{CS}$-locations, the space complexity can be reduced to $\Theta(n^2)$. This is done by dropping tables $LOC_2, \ldots, LOC_k$ for the orthologous clusters and only using the entries in tables $LAST$ and LOC_1.

3.4.9 Gene Clusters in a Subset of Multiple Genomes

With the algorithms used for multiple strings, we are now able to find gene clusters that occur simultaneously in any given number of genomes. Unfortunately, with an increasing number of genomes, the probability to have a conserved gene cluster in all genomes decreases rapidly. For the use on biological data, it is hence even more interesting to find gene clusters that appear in only a subset of the given genomes. More formally spoken, we search for gene clusters that appear in a subset of at least k' out of k given genomes. Therefore, it is necessary to soften the requirements of the definition of common $\mathcal{CS}$-factors:

Definition (common $\mathcal{CS}$-factor of k' out of k strings). Given a collection of k strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$ over an alphabet Σ and a threshold $k' \leq k$, a subset $C \subseteq \Sigma$ is a *common $\mathcal{CS}$-factor of k' out of k strings* of $\mathcal{S}$ if and only if C has a $\mathcal{CS}$-location in at least k' strings $S_{i_1}, \ldots, S_{i_{k'}} \in \mathcal{S}$, $1 \leq i_r \leq k$, $1 \leq r \leq k'$.

The corresponding problems are defined similar to Problems 1 and 2 (see Section 3.4.1 on page 29):

Problem 3. Given a collection of k strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$ and a threshold k', find all its common $\mathcal{CS}$-factors of k' out of k strings.

Problem 4. For each common $\mathcal{CS}$-factor of k' out of k strings of $\mathcal{S}$, find all its maximal $\mathcal{CS}$-locations in each of the S_l, $1 \leq \ l \leq k$ it occurs in.

Based on the above described iterated use of the algorithms for multiple strings, their extension in order to solve Problems 3 and 4 for common $\mathcal{CS}$-factors based on the relaxed definition is rather straight-forward.

For the moment, let us restrict our attention to common $\mathcal{CS}$-factors that are present in S_1. Then it is possible to use the above described algorithm and to count in how many strings a $\mathcal{CS}$-location of a common $\mathcal{CS}$-factor was detected. Therefore, another table $COUNT(i,j)$ is used that counts the number of strings a common $\mathcal{CS}$-factor was found in. Initially, all entries in $COUNT(i,j)$ are set to 1. Whenever, during the processing of string pair (S_1, S_r), the algorithm finds a common $\mathcal{CS}$-factor with a $\mathcal{CS}$-location $[i,j]_{S_1}$ and $[k,l]_{S_r}$ for the first time, $COUNT(i,j)$ is incremented by one. A $\mathcal{CS}$-location $[i,j]_{S_1}$ is seen for the first time if $LAST(i,j) \neq r$ before being updated to r. Then, the output is written for each pair $[i,j]$ with $COUNT(i,j) \geq k'$.

In the general case, a common $\mathcal{CS}$-factor does not necessarily occur in S_1. So, in a next step it is necessary to apply this algorithm again to the collection $\mathcal{S}$ without string S_1. This procedure is repeated $k - k'$ times, until the number of remaining strings in $\mathcal{S}$ falls below k' yielding a worst case time complexity of $\Theta(k(1+k-k')n^2)$. The space complexity is $O(kn^2)$ if non-redundant output is written. If only Problem 3 is to be solved, the space complexity can be reduced to $\Theta(n^2)$, just as in the non-relaxed case.

Summarizing we can conclude that an efficient incorporation of paralogous genes into the former gene cluster model based on common intervals leads to a new model for which time complexity to detect the clusters increases from linear to quadratic time. In both cases the time complexity grows linearly with the number of compared genomes. Therefore, it seems to be possible to incorporate further refinements of the gene cluster model and still staying feasible in computation time and space.

3.5 A Related Approach: Gene Teams

Introducing the concept of *gene teams* as a formal model for gene clusters, Bergeron *et al.* [10] chose an alternative way of improving the model of gene clusters based on common intervals. Relaxing the requirement of direct consecution of genes in a cluster (also referred to as errors from strictness in Section 3.3.3), they allowed the genes to be separated by gaps whose length does not exceed a given threshold overcoming the problem that a single misplaced gene disqualifies the commonality between two or more otherwise similar genomic regions.

To formalize the concept of gene teams, they denote by Σ the set of n genes on a chromosome (or string) C. A function P_C: $\Sigma \to \mathbb{R}$ associates to each gene $g \in \Sigma$ a real number $P_C(g)$, called its position. Therefore, the function P_C induces a permutation on any subset S of Σ, ordering the genes in S from the gene of lowest position to the gene of highest position. Given two genes g and g', the distance between the two genes on chromosome C is given by $\Delta_C(g,g') = |P_C(g) - P_C(g')|$. For example, if $\Sigma = \{1,2,3,4,5\}$ take the following chromosome C, where genes not in Σ are denoted by asterisks: $C = 3 * * 5\ 4\ 1 * 2$. If the position of a gene is given by the number of genes preceding it, then $\Delta_C(3,4) = |0-4| = 4$ and the permutation induced on the subset $\{1,3,5\}$ is $(3\ 5\ 1)$.

For a given subset $S \subseteq \Sigma$, and $(g_1 \dots g_k)$ the permutation induced on S on a chromosome C, for $\delta > 0$, S is called a *δ-chain* of C if $\Delta_C(g_j, g_{j+1}) \leq \delta$, for $1 \leq j \leq k$. This definition is somehow similar to the definition of character sets of a substring (see Section 3.4.1) since it describes a set of genes appearing in close proximity of each other, but in the case of δ-chains it forbids multiple occurrences of a gene and for $\delta > 1$ does not require a direct consecution of the genes. Then a subset $S \subseteq \Sigma$ is a *δ-set* of two chromosomes C and C' if S is a δ-chain in both chromosomes, and similar to a common $\mathcal{CS}$-factor, a gene cluster is defined as a *δ-team* (or gene team) of C and C', that is a maximal *δ-set* with respect to inclusion. For example consider the following chromosomes: $C = 3 * * 5\ 4\ 1 * 2$ and $C' = 1\ 2 * * * 3 * 4\ 5$. For $\delta = 1$ the only δ-team is the set $\{4,5\}$. With $\delta = 2$ we get $\{1,2\}$ and $\{4,5\}$, and with $\delta = 3$ the resulting teams are $\{1,2\}$ and $\{3,4,5\}$. Finally, with $\delta = 3$, the δ-team $\{1,2,3,4,5\}$ contains all genes from Σ.

The definition of a gene team is straight-forwardly extendable to multiple chromosomes as well as circular chromosomes. In [10], Bergeron *et al.* present an efficient $O(mn \log^2 n)$ time algorithm that requires $O(nm)$ space, where m is the number of chromosomes and n the maximal number of contained genes.

Acting on this model of gene clusters based on gene teams, He and Goldwasser [30] generalized the concept by removing the constraint that each gene must have a unique copy, i.e. again the generalization from permutations to strings. Their definitions of δ-chain, δ-set and δ-team are similar to the original definitions. In fact, when applied to a pair of chromosomes with a one-to-one orthologous relationship, they are equal. For the development of an efficient algorithm to detect the gene teams on permutations, Bergeron *et al.* made use of the following helpful properties: If Σ_1 and Σ_2 are δ-chains of C and $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, then $\Sigma_1 \cup \Sigma_2$ is also a δ-chain. A similar property follows for δ-sets, and thus it is shown that the collection of δ-teams forms a disjoint partition of the underlying genes.

With the existence of paralogs, these properties may no longer hold, and in general this leads to different structural properties that complicate the algorithmic search for the extended δ-teams. Finally He and Goldwasser described an $O(n^2)$ time and linear space algorithm for the comparison of two chromosomes of maximal length n. Unfortunately, a major drawback of this method is that it is not efficiently extendable to more than two genomes, since it shows a worst case time complexity of $O(n^m)$, where m denotes the number of compared chromosomes. However, their application to the two bacterial genomes of *B. subtilis* and *E. coli* generated meaningful results and clearly showed again that it is essential to have an appropriate model that includes the notion of paralogous genes.

k	δ	permutation based	string based
2	0	$O(n + \texttt{output})$ [75]	$O(n^2)$ [18] $\Theta(n^2)$ [64] $\Theta(n\lvert\Sigma\rvert(\log(\lvert\Sigma\rvert/\log n)+1))$ [18]
2	≥ 0	$O(n \log^2 n)$ [10]	$O(n^2)$ [30]
≥ 2	0	$O(kn + \texttt{output})$ [32]	$O(kn^2)$ [18] $\Theta(kn^2)$ [64] $\Theta(kn\lvert\Sigma\rvert(\log(\lvert\Sigma\rvert/\log n)+1))$ [18]
≥ 2	≥ 0	$O(kn)$ [10]	$O(n^k)$ [30]

Table 3.1: Algorithm Overview. The table shows the time complexities for the algorithms described before, when applied to two and multiple (k) genomes, where $\delta \geq 0$ denotes that the genes in a cluster do not have to be in direct consecution.

3.6 Summary

In the previous sections, we discussed various aspects of the phenomenon that bacterial genomes tend to maintain several small regions of conserved gene content during the course of evolution. The knowledge that those regions of local gene order conservation contain important information for many different types of genome analysis drove the development of different strategies to detect these gene clusters. Since the first approaches to locate gene clusters were not based on a formal model, their definition differed as the case arises and the heuristic algorithms for their detection depended on settings of very specific parameters, like the significance of sequence similarity and measure of phylogenetic distances. With the first rigorous formulation of the concept of gene clusters based on the definition of common intervals over permutations, the term gene cluster started to become a formal notion. Shortly after the formal description, efficient algorithms were presented, allowing the detection of these clusters in two and afterwards in multiple genomes. Unfortunately, the first formal models turned out to be too abstract formulations of the biological concept of a gene cluster. Therefore, the focus in research was directed to the development of more specific models, which are closer to the biological concept, but also more difficult to define. On a small scale of a few or only two genomes, these models incorporating the notion of paralogous genes or genes not in direct consecution have proven that in many cases the detected clusters match with known groups of interacting genes.

For the development of a tool to detect gene clusters on a large set of $10-100$ genomes, it is decisive to choose a well suited model allowing a fast *and* reliable detection of the clusters. Therefore, from the described models (see Table 3.1) we can exclude those which do not allow the presence of paralogous genes (i.e. defined on permutations), since especially large genomes tend to contain several copies of a single gene. Due to exponential runtime in the number of genomes, the string based model of gene teams also disqualifies for an efficient implementation. Therefore, the only remaining option for multiple genomes is the gene cluster model based on common $\mathcal{CS}$-factors. In Section 3.4, we presented two algorithms finding the clusters in $O(kn^2)$ time and space for k genomes if the variant for the non-redundant output is selected. Since the time and space complexity of the algorithms grows linearly in the number of genomes, this is the most suitable model and used in the implementation for our tool Gecko (see Chapter 5). From the two possible algorithms to detect gene clusters based on common $\mathcal{CS}$-factors, we chose Algorithm CI for the implementation, since it is based on more elementary data structures, and therefore can be expected to perform more efficient at runtime.

Remaining Problems

For all algorithms detecting gene clusters defined as common $\mathcal{CS}$-factors, we face the problem that in their output still a large amount of redundant information is present. Consider the following example: Let $S_1 = (1,2,3,4,5,6)$, $S_2 = (7,8,2,3,4,5,9)$ and $S_3 = (10,5,4,3,2,11)$, the output according to Section 3.4.8 is:

1. $S_1 : [2,2] \;-\; S_2 : [3,3] \;-\; S_3 : [5,5]$
2. $S_1 : [2,3] \;-\; S_2 : [3,4] \;-\; S_3 : [4,5]$
3. $S_1 : [2,4] \;-\; S_2 : [3,5] \;-\; S_3 : [3,5]$
4. $S_1 : [2,5] \;-\; S_2 : [3,6] \;-\; S_3 : [2,5]$
5. $S_1 : [3,3] \;-\; S_2 : [4,4] \;-\; S_3 : [4,4]$
6. $S_1 : [3,4] \;-\; S_2 : [4,5] \;-\; S_3 : [3,4]$
7. $S_1 : [3,5] \;-\; S_2 : [4,6] \;-\; S_3 : [2,4]$
8. $S_1 : [4,4] \;-\; S_2 : [5,5] \;-\; S_3 : [3,3]$
9. $S_1 : [4,5] \;-\; S_2 : [5,6] \;-\; S_3 : [2,3]$
10. $S_1 : [5,5] \;-\; S_2 : [6,6] \;-\; S_3 : [2,2]$

Obviously, the gene cluster of practical interest is the cluster no 4, since all other clusters only contain a subset of characters from cluster no 4. Note that this type of redundancy only occurs on gene clusters defined as common $\mathcal{CS}$-factors, since a gene team of a set of characters is defined as a maximal δ-set with respect to inclusion. Thus, a filter or postprocessing step to eliminate such redundancies is not required for gene cluster detection algorithms defined based on the definition of gene teams.

Having eliminated such *non-maximal* gene clusters, the output of a cluster detection algorithm can be significantly reduced without loosing any valuable information. For the practical application in comparative genomics, there are several further directions in which the output of the algorithms might be optimized. All these optimizations can be performed in a postprocessing step succeeding the main algorithm and are described in detail in Section 5.2.

Chapter 4

Data Preparation

Before we focus on the practical application of our algorithms to detect gene clusters in real genomic data, in this chapter we will describe how the input data for such algorithms can be generated. As we discussed at the end of the previous chapter, the most appropriate model for cluster detection using multiple genomes is based on gene clusters described as common $\mathcal{CS}$-factors. In this model, the genomes are represented by strings of characters, where the i-th gene in a genome has its corresponding character at the i-th position in the associated string. The challenging task in the generation of the input data for our algorithms is the denotation of homologous genes in all genomes by the same character (i.e. to identify which genes belong to one family of homologs) in each of the associated strings. Therefore, in this chapter we will present two different approaches to generate these strings of characters which can be applied to our algorithms for gene cluster detection. The first approach is based on the family classification from the COG database[1] and was used in the initial evaluation stage (see Chapter 5). Since a detailed analysis of the gene cluster detection algorithms required a more flexible definition of families of homologous genes, we developed an alternative approach to group genes to families of homologs. The underlying model, as well as the algorithms and visualization of the generated families are described in the second part of this chapter.

Remember that the terms *homology*, *orthology*, and *paralogy* originally describe the phylogenetical property of descent from a common origin. Since true evidence for a common origin can only be given by exploration of the common ancestors of the genes in all intermediate forms and we do not have fossil records of these extinct forms, this is impossible to derive by bioinformatics means. Our use of these terms in the following sections will therefore be slightly more relaxed by only predicting a common origin, implying a similar function, based on the similarity of their DNA sequences.

[1] Clusters of Orthologous Groups of proteins, http://www.ncbi.nlm.nih.gov/COG

4.1 The COG Database

The database of Clusters of Orthologous Groups of proteins (COGs) was established by Tatusov and co-workers in 1997 [72] and is designed to classify genes from completely sequenced genomes on the basis of their common origin. Due to the recent progress in genome sequencing, the database was updated several times [71, 74] and now contains 138,458 genes from 66 unicellular organisms that are clustered into 4873 COGs [70].

For the classification of genes into COGs, an all-against-all sequence comparison of the protein coding gene sequences in all completely sequenced genomes was performed. For the sequence comparison, the gapped BLAST program of Altschul *et al.* [3] was used under exclusion of regions with low-complexity and predicted coiled-coil regions [72]. Then, the clustering of genes into COGs is based on the idea that any group of three genes from distantly related genomes that are more similar to each other than to any other gene from the same genomes belong to an orthologous family. More precisely, the construction of the COGs is performed as follows:

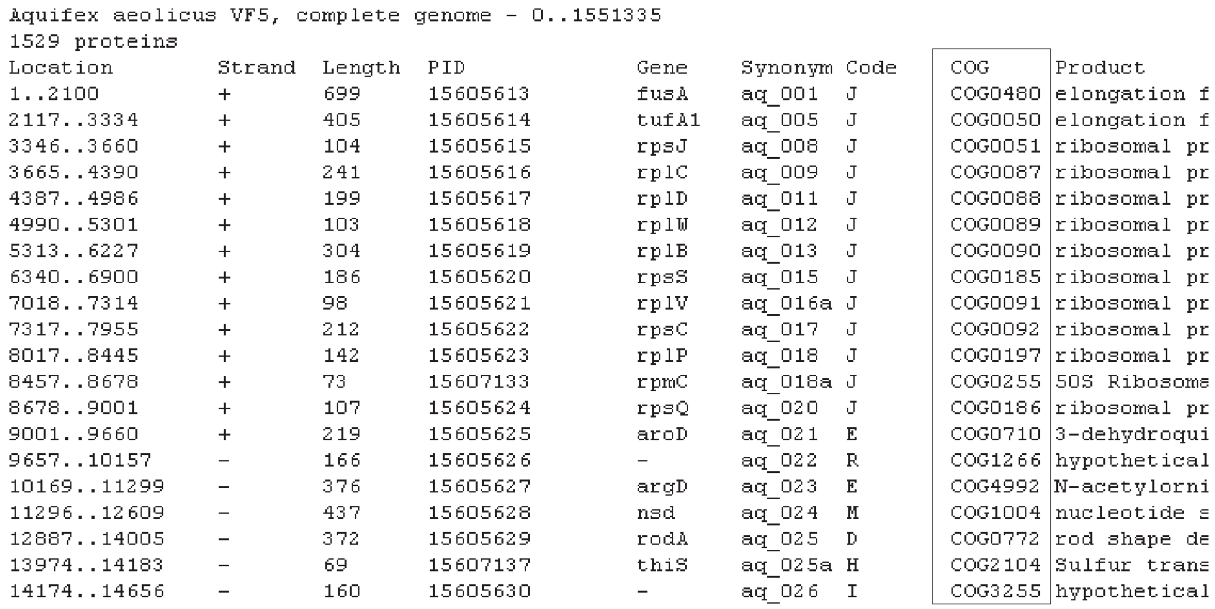
(i) After computing the all-gainst-all sequence comparison, (ii) all paralogous genes are grouped together. Here two genes are called paralogous, if they are more similar to each other than to any other gene from other genomes. (iii) Then triangles of mutually consistent, genome-specific best hits are located. (iv) These triangles are merged if two of them share a common side. In two further steps the constructed clusters are manually optimized. This becomes especially important, since the presence of fused genes and genes encoding multi-domain proteins sometimes leads to the connection of particular clusters. (v) These clusters have to be identified and separated. On the other hand, some genes have evolved via duplication after the divergence of the compared species (lineage specific gene expansion [45]). In this situation, the identification of the orthology relationship is difficult and might lead to the creation of large clusters of co-orthologs. (vi) Therefore, large COGs are reviewed in more detail using phylogenetic trees, cluster analysis and visual alignment inspection. As a result those clusters might be split into two or more COGs.

By this method, in the latest update of the database the rate of genes present in COGs varies from 43% in *Borrelia burgdorferi* to 99% in *Buchnera sp.* In the average case, the rate of genes belonging to COGs in prokaryotic genomes is approximately 80%.

Extracting the family classification

The most practical approach to transform a genome classified in the COG database into a string of (COG-)numbers, is downloading the genome from the NCBI database for complete bacterial genomes[2]. The database contains all publicly available genomes, where each genome is given in a table in which the genes are ordered along their occurrence in the

[2] http://www.ncbi.nlm.nih.gov/genomes/static/eub_g.html



```
Aquifex aeolicus VF5, complete genome - 0..1551335
1529 proteins
Location        Strand  Length  PID          Gene    Synonym Code   COG     Product
1..2100         +       699     15605613     fusA    aq_001  J      COG0480 elongation f
2117..3334      +       405     15605614     tufA1   aq_005  J      COG0050 elongation f
3346..3660      +       104     15605615     rpsJ    aq_008  J      COG0051 ribosomal pr
3665..4390      +       241     15605616     rplC    aq_009  J      COG0087 ribosomal pr
4387..4986      +       199     15605617     rplD    aq_011  J      COG0088 ribosomal pr
4990..5301      +       103     15605618     rplW    aq_012  J      COG0089 ribosomal pr
5313..6227      +       304     15605619     rplB    aq_013  J      COG0090 ribosomal pr
6340..6900      +       186     15605620     rpsS    aq_015  J      COG0185 ribosomal pr
7018..7314      +       98      15605621     rplV    aq_016a J      COG0091 ribosomal pr
7317..7955      +       212     15605622     rpsC    aq_017  J      COG0092 ribosomal pr
8017..8445      +       142     15605623     rplP    aq_018  J      COG0197 ribosomal pr
8457..8678      +       73      15607133     rpmC    aq_018a J      COG0255 50S Ribosome
8678..9001      +       107     15605624     rpsQ    aq_020  J      COG0186 ribosomal pr
9001..9660      +       219     15605625     aroD    aq_021  E      COG0710 3-dehydroqui
9657..10157     -       166     15605626     -       aq_022  R      COG1266 hypothetical
10169..11299    -       376     15605627     argD    aq_023  E      COG4992 N-acetylorni
11296..12609    -       437     15605628     nsd     aq_024  M      COG1004 nucleotide s
12887..14005    -       372     15605629     rodA    aq_025  D      COG0772 rod shape de
13974..14183    -       69      15607137     thiS    aq_025a H      COG2104 Sulfur trans
14174..14656    -       160     15605630     -       aq_026  I      COG3255 hypothetical
```

Figure 4.1: Structure of the table representation of *Aquifex aeolicus VF5* obtained from the NCBI database for complete bacterial genomes. For each gene, the entry in the column containing the associated COG cluster (boxed) can be used as identifier in the string representation of the genome. In this example, the representation of *A. aeolicus* starts with the numbers: 480, 50, 51, 87, 88, 89, ...

particular genomes. Each line in the table represents one gene together with additional information about the functional annotation, the exact location on the genome and the COG cluster the gene belongs to. By extracting the column for the COG cluster from such a table, a simple representation of a genome as a string of numbers can be obtained (see Figure 4.1).

Advantages and Disadvantages of using the COG database

Besides the easy access of the clustering information, a further advantage of using the COG database to create the string representation of the genomes is that this database is being frequently updated while more complete genome sequences become available. In addition, the classification of genes to their COGs is manually inspected, such that misclassifications appear less often than in a fully automated and unsupervised process.

On the other hand, not all sequenced prokaryotic genomes are available in the database. Especially many *Actinobacteria* like *Corynebacterium diphteriae*, *Corynebacterium efficiens*, *Nocardia farcinica*, *Streptomyces coelicolor*, and *Symbiobacterium thermophilum* are so far not considered in the COG classification, although some of the genome sequences are publicly available since a few years.

A further disadvantage of the COG database is the absence of interaction options to control the grouping of genes into COG clusters. For some applications, like the detection of genes with a conserved neighborhood, the COG classification is sometimes too specific, i.e. genes that have a similar function are grouped into different families. For example, the typical ABC transport system is built of three components (periplasmic component, ATPase component, and permease component) which are in most cases encoded by genes of the three particular families. Here, the COG database contains 36 different families for periplasmic components, 54 families for ATPase components, and 67 families for permease components.

Since the distribution of genes with a very similar function into different gene families complicates the location of conserved genomic neighborhoods, it is desirable to have a more flexible model of COGs allowing to put those genes into one gene family that share a common function, e.g. encode for one particular component.

4.2 A Relaxed Family Definition

The absence of many *Actinobacteria*, which are of high interest for a detailed evaluation of the algorithm in cooperation with members of the Institute of Genome Research at Bielefeld University, together with the clustering in the COG database being too specific, led to the decision to develop an alternative approach that allows a fully automated and parameterized grouping of genes into families of homologs, only based on the evaluation of their sequence similarity. Therefore, we present a new definition of a family of homologous genes, where homology in this case is defined by a combination of parameterized tests for paralogy and orthology. Based on this definition of gene families, we describe an algorithm that establishes this family classification based on the results of an all-gainst-all *TBLASTN* comparison.

4.2.1 Notation

Given a collection of k genomes $\mathcal{G} = (G_1, G_2, \ldots, G_k)$, $|G_i|$ is the number of genes in G_i and $G_i[j]$ denotes the j-th gene of genome G_i with $|G_i[j]|$ the length of its amino-acid sequence. Applying an all-against-all *TBLASTN* comparison [3], we get for each pair of genes ($G_i[j]$,$G_{i'}[j']$), $1 \leq i, i' \leq k$, $1 \leq j \leq |G_i|$, $1 \leq j' \leq |G_{i'}|$, a *TBLASTN* hit $Hit(G_i[j], G_{i'}[j'])$. The *TBLASTN* hit contains three different values: $Hit_p(G_i[j], G_{i'}[j'])$ is the rate of matching amino-acids, $Hit_l(G_i[j], G_{i'}[j'])$ the length of the calculated alignment, and $Hit_e(G_i[j], G_{i'}[j'])$ denotes the E-value of the hit. Furthermore we call

$$Hit_c(G_i[j], G_{i'}[j']) := \frac{Hit_l(G_i[j], G_{i'}[j'])}{\max(|G_i[j]|, |G_{i'}[j']|)}$$

the *coverage* of $Hit(G_i[j], G_{i'}[j'])$.

Definition (paralog). Given two genes $g = G_i[j]$ and $g' = G_{i'}[j']$ of $\mathcal{G}$, $i = i'$, $j \neq j'$, and the thresholds p_0, e_0, and c_0, g and g' are called *paralogous* if and only if

$$Hit_c(g, g') > c_0 \texttt{ and } (Hit_p(g, g') > p_0 \texttt{ or } Hit_e(g, g') < e_0).$$

We call two genes of the same genome paralogous, if their *TBLASTN* hit has a sufficiently large coverage and either the rate of matching amino-acids exceeds the chosen threshold or the probability that this hit occurred just by chance (E-value) is below the selected threshold.

Instead of relying only on the E-value as measurement of the significance of a *TBLASTN* hit, we also regard the rate of matching amino-acids to decide whether two genes are conserved paralogs or not. This becomes especially important for genes consisting only of a very short amino-acid sequence, since by the definition of the E-value genes of shorter amino-acid sequences are more likely to occur by chance, and therefore the E-value might not reflect the true significance of the hit. Another important aspect in the test-condition for paralogs is the term regarding the coverage of a *TBLASTN* hit. Many genes are composed of smaller pieces of amino-acid sequences called *domains.* Usually, domains encode a certain sub-function of a gene, and it is quite common that if a sub-function is required in many different processes of an organism, one will find several genes containing such a highly similar domain, while the general functions of the genes may differ. Therefore, choosing for example a threshold of $c_0 > 0.7$ requires that the length of the computed alignment for two genes has to cover at least 70% (generally at least three of four domains) of the amino acid sequence of both genes.

Definition (ortholog). Given two genes $g = G_i[j]$ and $g' = G_{i'}[j']$ of $\mathcal{G}$, $i \neq i'$, and the thresholds p_0, e_0, and c_0, g and g' are called *orthologous* if and only if

$$Hit_c(g, g') > c_0 \texttt{ and } (Hit_p(g, g') > p_0 \texttt{ or } Hit_e(g, g') < e_0).$$

In general, the definition of paralogs and orthologs are identical, except that orthologs cannot be found inside the same genome and paralogs not between different genomes. But observe, that in the practical application it will be possible to set the thresholds for the parameters p_0, e_0, and c_0 differently for paralogs and orthologs.

4.2.2 Creating the Families of Homologs

The general idea of our strategy to group genes into families of homologs is similar to the construction of the COG clusters described in Section 4.1. In contrast to the COG approach, here we have a more flexible description of paralogous and orthologous genes and our grouping procedure does not rely on the identification of best hits between species with a constrained phylogenetical relation (best hits creating a triangle structure in phylogenetically distant species).

After performing the all-against-all *TBLASTN* search, in a first step the *TBLASTN* hits between two genes g and g' are made symmetric. This becomes necessary, since for g and g' always at least two hits ($Hit(g, g')$ and $Hit(g', g)$) exist. Since the *TBLASTN* program is not symmetric in the order of input parameters the two hits might slightly differ in their coverage, E-value and matching rate. These differences may cause inconsistencies in determining if the two genes are paralogs or orthologs, especially if the values are in a close proximity around the selected thresholds p_0, e_0, and c_0. To enforce symmetry, the hits are modified by averaging as follows:

$$\begin{aligned}
\widetilde{Hit}_c(g, g') &= \frac{Hit_c(g', g) + Hit_c(g, g')}{2},\\
\widetilde{Hit}_p(g, g') &= \frac{Hit_p(g', g) + Hit_p(g, g')}{2},\\
\widetilde{Hit}_e(g, g') &= 10^{(\log_{10}(Hit_e(g',g))+\log_{10}(Hit_e(g,g')))/2}.
\end{aligned}$$

The creation of symmetric hits becomes more difficult if for two genes more than one pair of hits is reported by the *TBLASTN* program. In this case, only the pair which is least likely to occur by chance, i.e. the pair with the lowest E-value, is considered for determining orthology or paralogy.

In the next step, the families of paralogs for each of the given genomes are constructed using a single-linkage clustering (see Algorithm 5).

First, a gene g is chosen that was not assigned to any family before, and a new family f is created that initially consists only of the gene g. Then, all genes are added to the family f that are paralogs of g. For each gene newly added to a family f, it is tested if there exist other paralogs that are not present in f so far. If so, these genes are added to f as well. Finally, f is stored in the list L that holds the result of the clustering.

This procedure guarantees that if two genes are paralogs according to our definition, they are grouped into the same family. During the clustering, the restriction regarding the coverage of a *TBLASTN* hit is used to suppress the problem of chaining, which is a typical side-effect of single linkage clustering, especially when genes contain only a small but highly conserved matching region, e.g. a single conserved domain. Clearly, the selection of appropriate threshold values for p_0, e_0, and c_0 is essential for retrieving clusters of an appropriate quality, and is discussed in Section 6.3.

In the second grouping step, for each family of paralogs, corresponding (orthologous) families of paralogs in the other genomes are determined. Therefore we introduce the parameter r_0 referred to as *overlap ratio*. We call two families of paralogs f and f' a *pair*

Algorithm 5 Constructing Families of Paralogs

```
create an empty list $L$
for each genome $G_i$, $1 \leq i \leq k$ do
   for each gene $g$ from $G_i$ do
      if $g$ is not added to a paralog family then
         create a new family $f$
         add $g$ to a temporary list $l$
         while $l$ is not empty do
            $g \leftarrow$ first element from $l$
            add $g$ to $f$
            while $g$ has a paralog $g'$ not in $f$ do
               add $g'$ to $l$
            end while
            remove $g$ from $l$
         end while
         add $f$ to $L$
      end if
   end for
end for
```

of orthologous families if at least r_0 percent of the genes in f have an orthologous gene in f', where f' is the family of larger size, i.e. it contains at least as many genes as f. For an efficient iterative detection for pairs of orthologous families, the families of paralogs in L are sorted by their size, i.e. the number of contained genes, such that the largest family is at the first position in the list. Then families of homologs are built as follows (see Algorithm 6):

Algorithm 6 Constructing Families of Homologs

```
sort the families of paralogs in $L$ by their size in descending order
create an empty list $L'$
while $L$ is not empty do
   $f \leftarrow$ first family in $L$
   create an empty temporary list $l$
   for each family $f'$ in $L$ that forms a pair of orthologous families with $f$ do
      store the genes of $f'$ in $l$
      remove $f'$ from $L$
   end for
   add all genes from $l$ to $f$
   add $f$ to $L'$
   remove $f$ from $L$
end while
```

In the beginning, the largest (the first) family of paralogs $f \in L$ is taken and all genes from each family $f' \in L$ that form a pair of orthologous families with f are copied to a temporary list l. Afterwards, these families f' are removed from L and the family of homologs is created by adding all genes from the temporary list l to f. Finally, f is moved from L to the result list L'.

These steps are repeated until L is empty. Due to the change of the gene content in the families now stored in the list L' it is possible that there are further pairs of orthologous families present in L'. Therefore, the algorithm is applied again, now with L' as input list, as long as there are no further pairs of orthologous families found. After the procedure terminates, the last calculated list L' contains the final families of homologs.

To use the computed family classification for our gene cluster detection algorithm, each family in L' is assigned a unique identifier, such that the genomes can then be represented by sequences over the alphabet of these family identifiers. For simplicity, each family consisting of only one gene, is represented by the identifier '0', indicating that this gene has no homolog in any other of the given sequences.

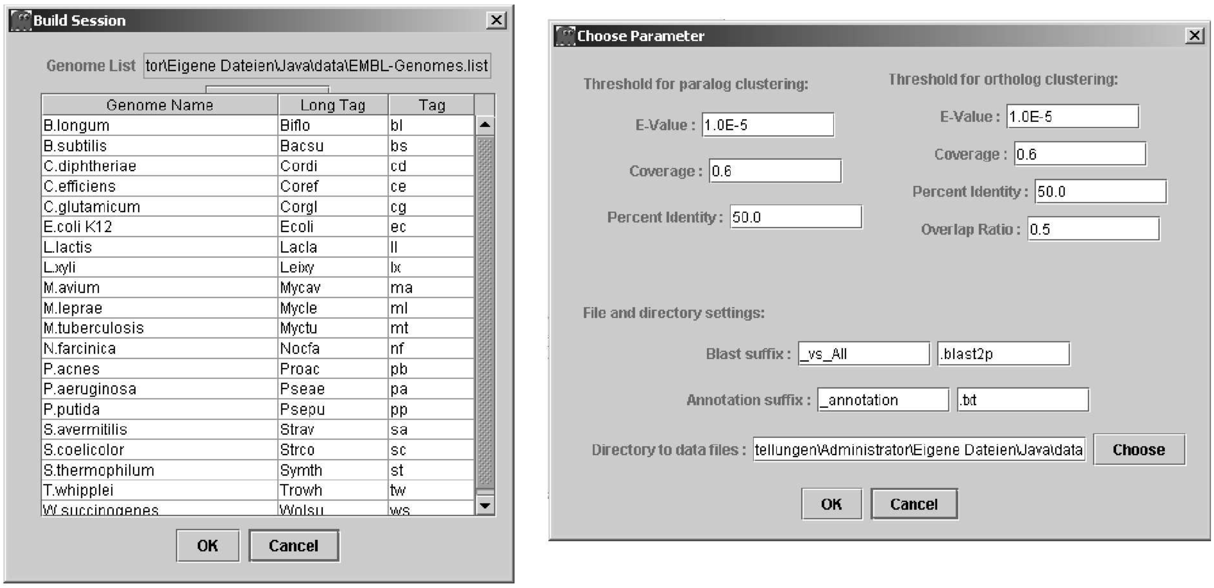
4.2.3 The GhostFam Tool

To provide an easy access to the family construction algorithm based on our relaxed gene family definition, we developed the tool GhostFam as a part of our gene cluster detection tool Gecko (see next Chapter). Besides the efficient implementation of the algorithms from the previous section, the focus in the development of GhostFam was directed to the graphical representation of the resulting families for an easy evaluation, verification, and manual manipulation. With the graphical user interface allowing a comfortable setup of all algorithmic parameters and the illustration of the computed results, GhostFam serves as an elementary module for data preparation in the Gecko tool.

Input Data and Threshold Setting

To successfully create a new session, GhostFam provides an interactive dialog to set the essential parameters for the localization of the input data files. These data files contain the information about the genomes to be analyzed as well as the gene content and the results of the all-against-all *TBLASTN* comparisons for each genome. After defining the input files, a second dialog allows the setting of the thresholds for the definition of orthologous and paralogous genes (see Figure 4.2). At the beginning of a new session, these thresholds are always set to their default values. The default values were experimentally obtained during the evaluation phase described in Chapter 6. During a GhostFam session it is also possible to modify these thresholds, but then the classification algorithms have to be restarted.

After confirming the parameter selection, the tool automatically starts the first grouping algorithm, i.e. the classification of the genes into their families of paralogs. Before continuing with the second grouping algorithm that creates the families of homologs, there



Build Session

Genome List: tor\Eigene Dateien\Java\data\EMBL-Genomes.list

Genome Name	Long Tag	Tag
B.longum	Biflo	bl
B.subtilis	Bacsu	bs
C.diphtheriae	Cordi	cd
C.efficiens	Coref	ce
C.glutamicum	Corgl	cg
E.coli K12	Ecoli	ec
L.lactis	Lacla	ll
L.xyli	Leixy	lx
M.avium	Mycav	ma
M.leprae	Mycle	ml
M.tuberculosis	Myctu	mt
N.farcinica	Nocfa	nf
P.acnes	Proac	pb
P.aeruginosa	Pseae	pa
P.putida	Psepu	pp
S.avermitilis	Strav	sa
S.coelicolor	Strco	sc
S.thermophilum	Symth	st
T.whipplei	Trowh	tw
W.succinogenes	Wolsu	ws

Choose Parameter

Threshold for paralog clustering:

E-Value : 1.0E-5

Coverage : 0.6

Percent Identity : 50.0

Threshold for ortholog clustering:

E-Value : 1.0E-5

Coverage : 0.6

Percent Identity : 50.0

Overlap Ratio : 0.5

File and directory settings:

Blast suffix : _vs_All .blast2p

Annotation suffix : _annotation .txt

Directory to data files : tellungen\Administrator\Eigene Dateien\Java\data

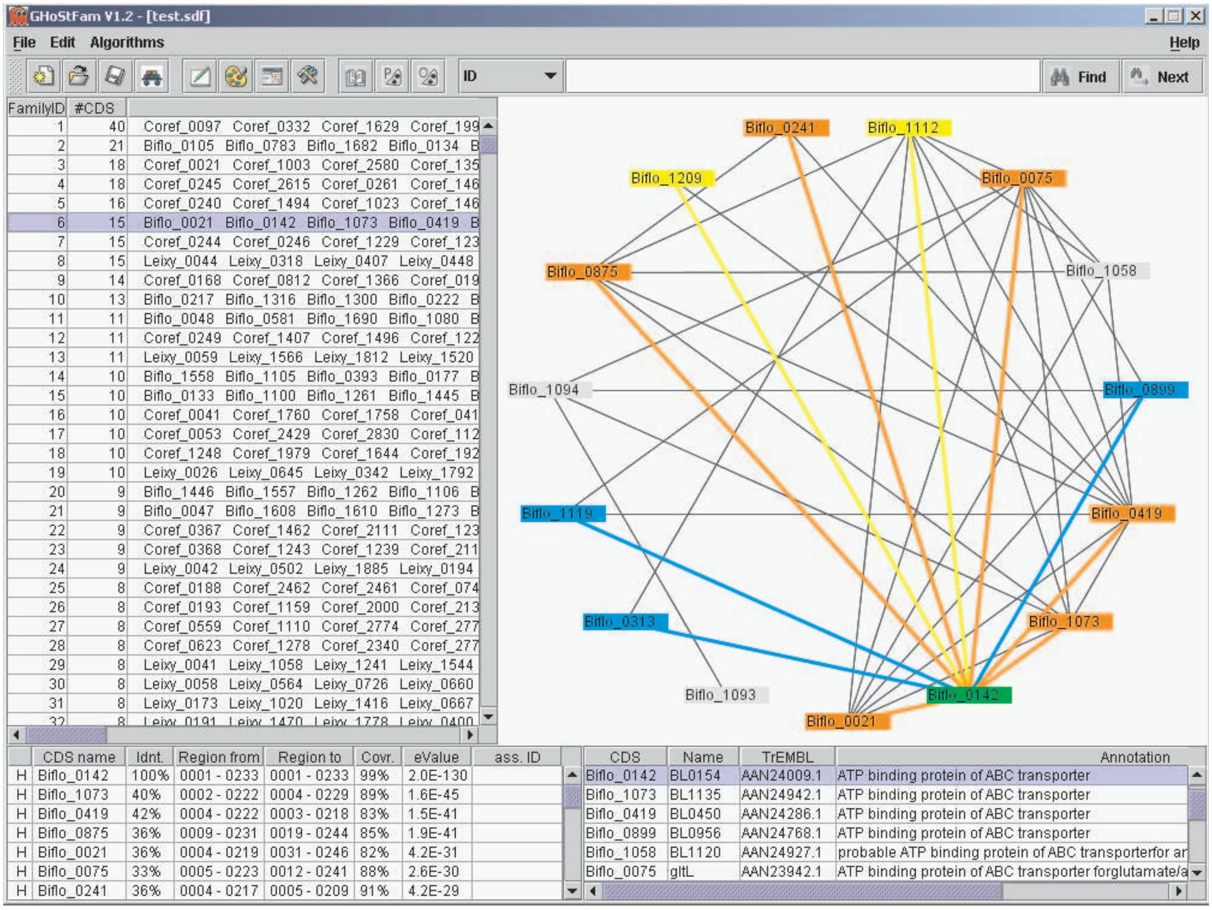
Figure 4.2: GhostFam dialogs for parameter input. The left part of the figure shows a list of available genomes read from an input data file. On the right side, the dialog for determining the thresholds for the family classification as well as file and directory settings are displayed.

is the option to perform manual corrections of the automatically created families of paralogs (see below in Paragraph 'Manual Editor'). This editor can also be used to optimize the final classification after the second grouping step.

Graphical Family Representation

The result of the classification is displayed in a table in the main window of GhostFam (see Figure 4.3). From this table, each family can be selected for a more detailed inspection. Therefore, the upper right part of the main window graphically displays a selected family based on the quality of the *TBLASTN* hits between its contained genes. In the graphic, each gene of the family is placed on a virtual cycle and a black line is drawn between two genes, whenever the two genes are either paralogs or orthologs.

For a more detailed analysis of the role of a single gene in the family, it can be selected in the graph and its connections to other genes can be read from the table in the lower left of the window. In this table, all hits of this gene to other genes of the family as well as to genes in other families are listed. Additionally, the hits of the selected gene to other genes inside the family are colored in the graph according to their significance. For an easy evaluation, the *TBLASTN* hits are divided into three classes of significance. Low significant hits are hits not fulfilling the test condition for paralogs (resp. orthologs) in Section 4.2.1. These hits are drawn in blue color in the graph and are abbreviated by an



FamilyID	#CDS	
1	40	Coref_0097 Coref_0332 Coref_1629 Coref_199
2	21	Biflo_0105 Biflo_0783 Biflo_1682 Biflo_0134 B
3	18	Coref_0021 Coref_1003 Coref_2580 Coref_135
4	18	Coref_0245 Coref_2615 Coref_0261 Coref_146
5	16	Coref_0240 Coref_1494 Coref_1023 Coref_146
6	15	Biflo_0021 Biflo_0142 Biflo_1073 Biflo_0419 B
7	15	Coref_0244 Coref_0246 Coref_1229 Coref_123
8	15	Leixy_0044 Leixy_0318 Leixy_0407 Leixy_0448
9	14	Coref_0168 Coref_0812 Coref_1366 Coref_019
10	13	Biflo_0217 Biflo_1316 Biflo_1300 Biflo_0222 B
11	11	Biflo_0048 Biflo_0581 Biflo_1690 Biflo_1080 B
12	11	Coref_0249 Coref_1407 Coref_1496 Coref_122
13	11	Leixy_0059 Leixy_1566 Leixy_1812 Leixy_1520
14	10	Biflo_1558 Biflo_1105 Biflo_0393 Biflo_0177 B
15	10	Biflo_0133 Biflo_1100 Biflo_1261 Biflo_1445 B
16	10	Coref_0041 Coref_1760 Coref_1758 Coref_041
17	10	Coref_0053 Coref_2429 Coref_2830 Coref_112
18	10	Coref_1248 Coref_1979 Coref_1644 Coref_192
19	10	Leixy_0026 Leixy_0645 Leixy_0342 Leixy_1792
20	9	Biflo_1446 Biflo_1557 Biflo_1262 Biflo_1106 B
21	9	Biflo_0047 Biflo_1608 Biflo_1610 Biflo_1273 B
22	9	Coref_0367 Coref_1462 Coref_2111 Coref_123
23	9	Coref_0368 Coref_1243 Coref_1239 Coref_211
24	9	Leixy_0042 Leixy_0502 Leixy_1885 Leixy_0194
25	8	Coref_0188 Coref_2462 Coref_2461 Coref_074
26	8	Coref_0193 Coref_1159 Coref_2000 Coref_213
27	8	Coref_0559 Coref_1110 Coref_2774 Coref_277
28	8	Coref_0623 Coref_1278 Coref_2340 Coref_277
29	8	Leixy_0041 Leixy_1058 Leixy_1241 Leixy_1544
30	8	Leixy_0058 Leixy_0564 Leixy_0726 Leixy_0660
31	8	Leixy_0173 Leixy_1020 Leixy_1416 Leixy_0667
32	8	Leixy_0191 Leixy_1470 Leixy_1778 Leixy_0400

	CDS name	Idnt.	Region from	Region to	Covr.	eValue	ass. ID
H	Biflo_0142	100%	0001 - 0233	0001 - 0233	99%	2.0E-130	
H	Biflo_1073	40%	0002 - 0222	0004 - 0229	89%	1.6E-45	
H	Biflo_0419	42%	0004 - 0222	0003 - 0218	83%	1.5E-41	
H	Biflo_0875	36%	0009 - 0231	0019 - 0244	85%	1.9E-41	
H	Biflo_0021	36%	0004 - 0219	0031 - 0246	82%	4.2E-31	
H	Biflo_0075	33%	0005 - 0223	0012 - 0241	88%	2.6E-30	
H	Biflo_0241	36%	0004 - 0217	0005 - 0209	91%	4.2E-29	

CDS	Name	TrEMBL	Annotation
Biflo_0142	BL0154	AAN24009.1	ATP binding protein of ABC transporter
Biflo_1073	BL1135	AAN24942.1	ATP binding protein of ABC transporter
Biflo_0419	BL0450	AAN24286.1	ATP binding protein of ABC transporter
Biflo_0899	BL0956	AAN24768.1	ATP binding protein of ABC transporter
Biflo_1058	BL1120	AAN24927.1	probable ATP binding protein of ABC transporterfor ar
Biflo_0075	gltL	AAN23942.1	ATP binding protein of ABC transporter forglutamate/a

Figure 4.3: Graphical Family Representation. The upper left table contains the list of all constructed families. From this table, a single family can be selected for the graphical representation on the right. For a selected gene from the displayed family (here *Bifidobacterium longum* 142, colored in green), one can find its annotation marked in the lower-right table and in the lower-left table the *TBLASTN* hits against all other genes. The hits to genes in the cluster are colored in the graph by their quality (orange: highly significant, yellow: significant, blue: not significant, grey: no hit found).

’L’ in the first column of the lower left table containing the hits. Those hits fulfilling the test condition for paralogs (resp. orthologs) are drawn in yellow color and abbreviated with ’S’ (significant). With an additional set of threshold values for the parameters of the test condition, highly significant hits (’H’) can be highlighted in orange color. Significant *TBLASTN* hits to genes in other families are marked with ’!’, while those highly significant hits are represented by ’!!’. This classification allows an easy evaluation of the connectivity of a created gene family. Especially the occurrence of highly significant hits to genes not in the family or the absence of significant hits inside a family hint to some sub-optimal classification.

In the lower right table of the main window, all genes from a family are listed together with their functional annotation, and their gene name and TrEMBL-ID[3], if available in the input data file.

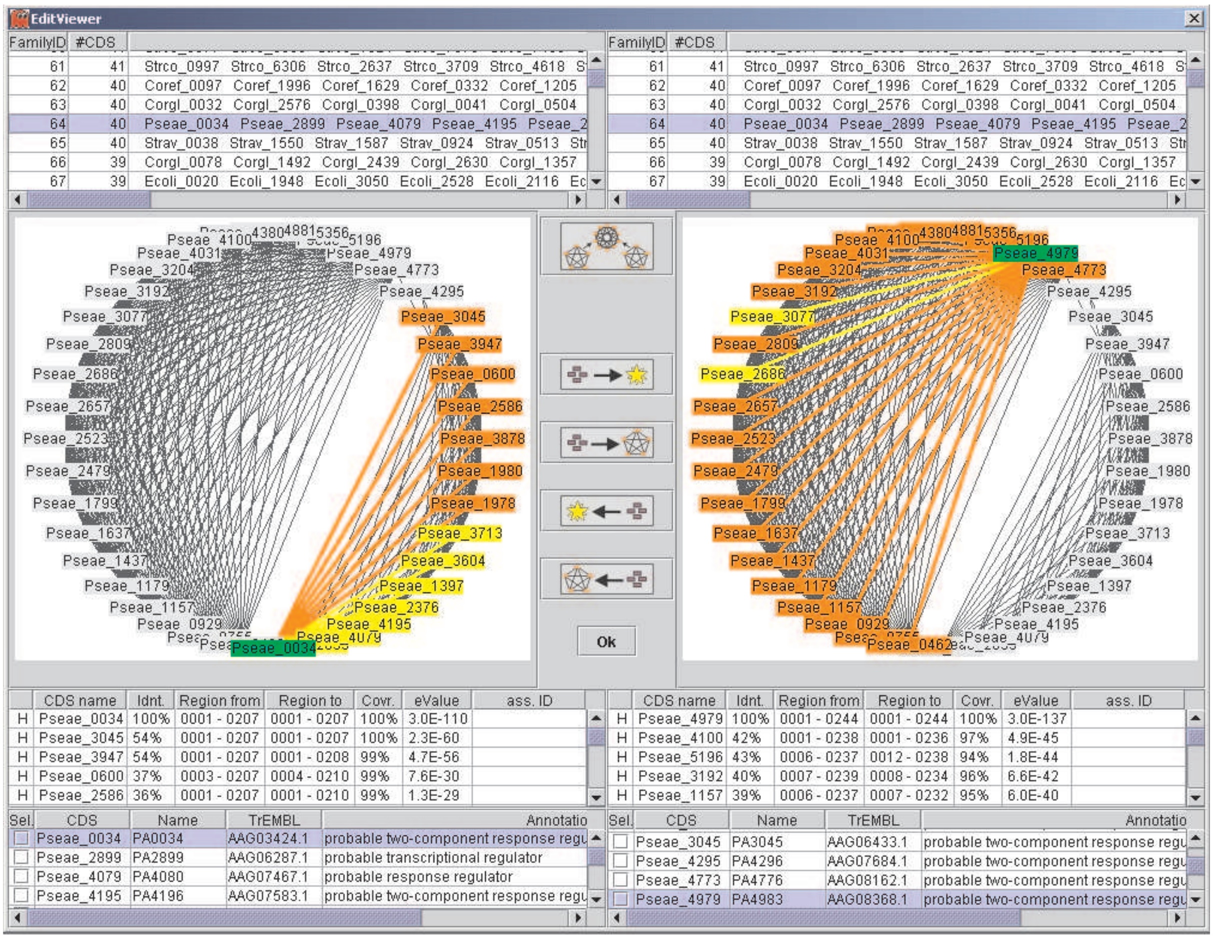
Manual Editor

Since an automatic classification of genes into families of homologs, only based on pairwise evaluation of their sequence similarity, cannot guarantee to result in the biologically most plausible partition, the computed list of families might contain some sub-optimally clustered gene families. Therefore, GhostFam also provides an editor (see Figure 4.4) to modify single family classifications manually. This manual editor is designed to be used at two different stages of the family classification process. First, after the grouping of genes into their families of paralogs, and for a final polishing of the gene families, once the families of homologs are generated.

In principle, the user interface for the manual editor is a duplicated version of the main window of GhostFam. It allows three different types of operation: Splitting a gene family into two new families, merging two families into one, and moving genes from one family to another. With these three elementary operations, all necessary steps to achieve a desired configuration can be performed.

Due to the side-effect of chaining in the single-linkage clustering of genes into their families of paralogs, splitting a family into two or more can be expected to be the most common operation in the editor. A typical example is given in Figure 4.4. The figure clearly depicts a gene family (no. 64) that consists of two groups of genes that are highly connected among each other, but show almost no similarity between each other. In this situation the editor allows to delineate the two groups and to split them into two families. The result is shown in Figure 4.5.

[3] http://www.ebi.ac.uk/trembl/index.html



FamilyID	#CDS	
61	41	Strco_0997 Strco_6306 Strco_2637 Strco_3709 Strco_4618 S
62	40	Coref_0097 Coref_1996 Coref_1629 Coref_0332 Coref_1205
63	40	Corgl_0032 Corgl_2576 Corgl_0398 Corgl_0041 Corgl_0504
64	40	Pseae_0034 Pseae_2899 Pseae_4079 Pseae_4195 Pseae_2
65	40	Strav_0038 Strav_1550 Strav_1587 Strav_0924 Strav_0513 Str
66	39	Corgl_0078 Corgl_1492 Corgl_2439 Corgl_2630 Corgl_1357
67	39	Ecoli_0020 Ecoli_1948 Ecoli_3050 Ecoli_2528 Ecoli_2116 Ec

FamilyID	#CDS	
61	41	Strco_0997 Strco_6306 Strco_2637 Strco_3709 Strco_4618 S
62	40	Coref_0097 Coref_1996 Coref_1629 Coref_0332 Coref_1205
63	40	Corgl_0032 Corgl_2576 Corgl_0398 Corgl_0041 Corgl_0504
64	40	Pseae_0034 Pseae_2899 Pseae_4079 Pseae_4195 Pseae_2
65	40	Strav_0038 Strav_1550 Strav_1587 Strav_0924 Strav_0513 Str
66	39	Corgl_0078 Corgl_1492 Corgl_2439 Corgl_2630 Corgl_1357
67	39	Ecoli_0020 Ecoli_1948 Ecoli_3050 Ecoli_2528 Ecoli_2116 Ec

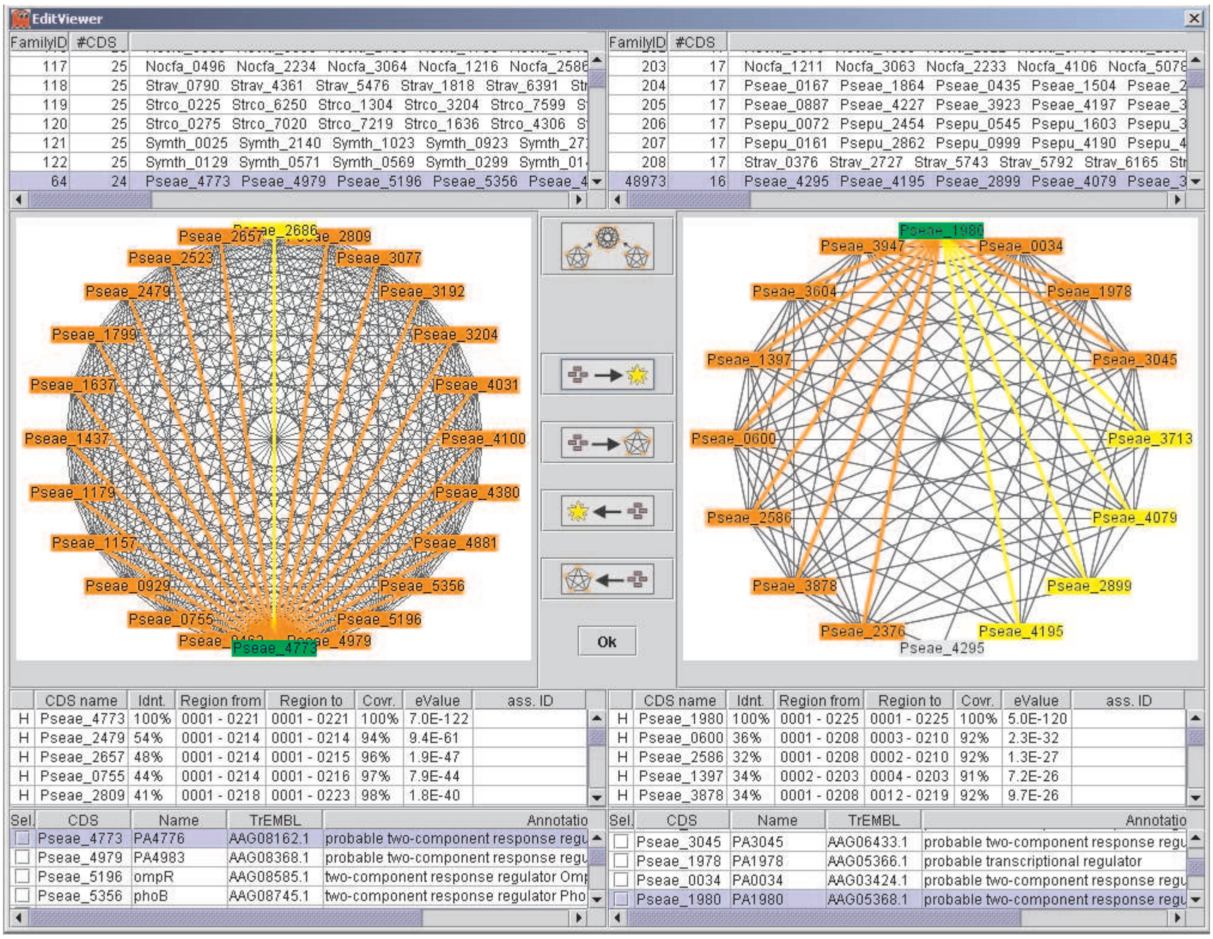
	CDS name	Idnt.	Region from	Region to	Covr.	eValue	ass. ID
H	Pseae_0034	100%	0001 - 0207	0001 - 0207	100%	3.0E-110	
H	Pseae_3045	54%	0001 - 0207	0001 - 0207	100%	2.3E-60	
H	Pseae_3947	54%	0001 - 0207	0001 - 0208	99%	4.7E-56	
H	Pseae_0600	37%	0003 - 0207	0004 - 0210	99%	7.6E-30	
H	Pseae_2586	36%	0001 - 0207	0001 - 0210	99%	1.3E-29	

	CDS name	Idnt.	Region from	Region to	Covr.	eValue	ass. ID
H	Pseae_4979	100%	0001 - 0244	0001 - 0244	100%	3.0E-137	
H	Pseae_4100	42%	0001 - 0238	0001 - 0236	97%	4.9E-45	
H	Pseae_5196	43%	0006 - 0237	0012 - 0238	94%	1.8E-44	
H	Pseae_3192	40%	0007 - 0239	0008 - 0234	96%	6.6E-42	
H	Pseae_1157	39%	0006 - 0237	0007 - 0232	95%	6.0E-40	

Sel.	CDS	Name	TrEMBL	Annotatio
□	Pseae_0034	PA0034	AAG03424.1	probable two-component response regu
□	Pseae_2899	PA2899	AAG06287.1	probable transcriptional regulator
□	Pseae_4079	PA4080	AAG07467.1	probable response regulator
□	Pseae_4195	PA4196	AAG07583.1	probable two-component response regu

Sel.	CDS	Name	TrEMBL	Annotatio
□	Pseae_3045	PA3045	AAG06433.1	probable two-component response regu
□	Pseae_4295	PA4296	AAG07684.1	probable two-component response regu
□	Pseae_4773	PA4776	AAG08162.1	probable two-component response regu
□	Pseae_4979	PA4983	AAG08368.1	probable two-component response regu

Figure 4.4: GhostFam Manual Editor. Both sides of the editor show the same gene family that clearly consists of two different groups of genes. To split the gene family, on one side, the genes which should create the new family have to be selected by double-click in the graphic or in the bottom table. By clicking on the corresponding button the selected genes are moved into a new family. The result of the split is shown in the next figure.



FamilyID	#CDS	
117	25	Nocfa_0496 Nocfa_2234 Nocfa_3064 Nocfa_1216 Nocfa_2586
118	25	Strav_0790 Strav_4361 Strav_5476 Strav_1818 Strav_6391 Str
119	25	Strco_0225 Strco_6250 Strco_1304 Strco_3204 Strco_7599 S
120	25	Strco_0275 Strco_7020 Strco_7219 Strco_1636 Strco_4306 S
121	25	Symth_0025 Symth_2140 Symth_1023 Symth_0923 Symth_27
122	25	Symth_0129 Symth_0571 Symth_0569 Symth_0299 Symth_01
64	24	Pseae_4773 Pseae_4979 Pseae_5196 Pseae_5356 Pseae_4

FamilyID	#CDS	
203	17	Nocfa_1211 Nocfa_3063 Nocfa_2233 Nocfa_4106 Nocfa_5078
204	17	Pseae_0167 Pseae_1864 Pseae_0435 Pseae_1504 Pseae_2
205	17	Pseae_0887 Pseae_4227 Pseae_3923 Pseae_4197 Pseae_3
206	17	Psepu_0072 Psepu_2454 Psepu_0545 Psepu_1603 Psepu_3
207	17	Psepu_0161 Psepu_2862 Psepu_0999 Psepu_4190 Psepu_4
208	17	Strav_0376 Strav_2727 Strav_5743 Strav_5792 Strav_6165 Str
48973	16	Pseae_4295 Pseae_4195 Pseae_2899 Pseae_4079 Pseae_3

	CDS name	Idnt.	Region from	Region to	Covr.	eValue	ass. ID
H	Pseae_4773	100%	0001 - 0221	0001 - 0221	100%	7.0E-122	
H	Pseae_2479	54%	0001 - 0214	0001 - 0214	94%	9.4E-61	
H	Pseae_2657	48%	0001 - 0214	0001 - 0215	96%	1.9E-47	
H	Pseae_0755	44%	0001 - 0214	0001 - 0216	97%	7.9E-44	
H	Pseae_2809	41%	0001 - 0218	0001 - 0223	98%	1.8E-40	

	CDS name	Idnt.	Region from	Region to	Covr.	eValue	ass. ID
H	Pseae_1980	100%	0001 - 0225	0001 - 0225	100%	5.0E-120	
H	Pseae_0600	36%	0001 - 0208	0003 - 0210	92%	2.3E-32	
H	Pseae_2586	32%	0001 - 0208	0002 - 0210	92%	1.3E-27	
H	Pseae_1397	34%	0002 - 0203	0004 - 0203	91%	7.2E-26	
H	Pseae_3878	34%	0001 - 0208	0012 - 0219	92%	9.7E-26	

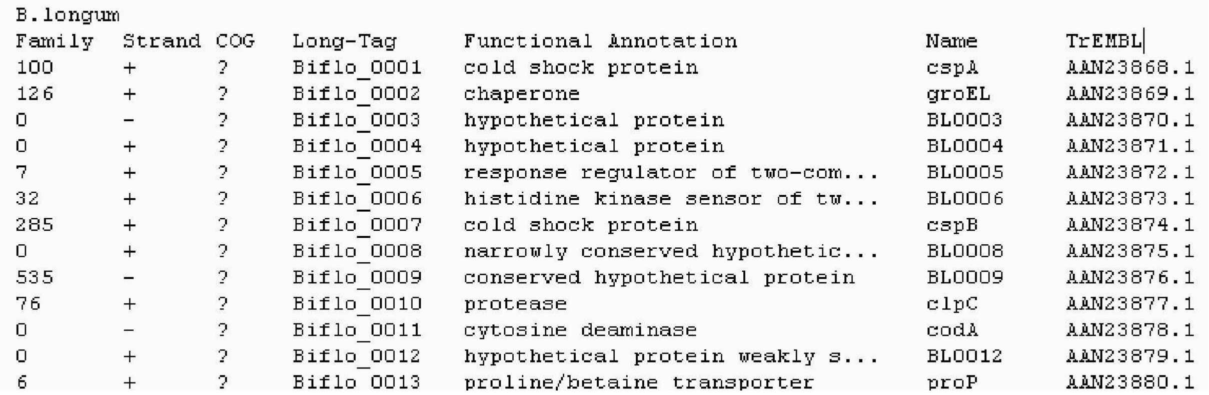
Sel.	CDS	Name	TrEMBL	Annotatio
□	Pseae_4773	PA4776	AAG08162.1	probable two-component response regu
□	Pseae_4979	PA4983	AAG08368.1	probable two-component response regu
□	Pseae_5196	ompR	AAG08585.1	two-component response regulator Omp
□	Pseae_5356	phoB	AAG08745.1	two-component response regulator Pho

Sel.	CDS	Name	TrEMBL	Annotatio
□	Pseae_3045	PA3045	AAG06433.1	probable two-component response regu
□	Pseae_1978	PA1978	AAG05366.1	probable transcriptional regulator
□	Pseae_0034	PA0034	AAG03424.1	probable two-component response regu
□	Pseae_1980	PA1980	AAG05368.1	probable two-component response regu

Figure 4.5: The result of the suggested split from the previous figure. Now each family is almost completely connected, i.e. all genes in the family can be considered to be homologs.

From GhostFam to Gecko

After the successful grouping of all genes into their families of homologs, the export tool of GhostFam creates a data file which can be used as input for the gene cluster detection with Gecko. In the data file, each gene is represented by the number of the gene family it belongs to (see Figure 4.6). For simplicity, each gene family consisting of only one gene is denoted by '0', indicating that no homolog was found in the processed genomes.



```
B.longum
Family   Strand COG   Long-Tag     Functional Annotation              Name       TrEMBL
100        +     ?    Biflo_0001   cold shock protein                 cspA       AAN23868.1
126        +     ?    Biflo_0002   chaperone                          groEL      AAN23869.1
0          -     ?    Biflo_0003   hypothetical protein               BL0003     AAN23870.1
0          +     ?    Biflo_0004   hypothetical protein               BL0004     AAN23871.1
7          +     ?    Biflo_0005   response regulator of two-com...   BL0005     AAN23872.1
32         +     ?    Biflo_0006   histidine kinase sensor of tw...   BL0006     AAN23873.1
285        +     ?    Biflo_0007   cold shock protein                 cspB       AAN23874.1
0          +     ?    Biflo_0008   narrowly conserved hypothetic...   BL0008     AAN23875.1
535        -     ?    Biflo_0009   conserved hypothetical protein     BL0009     AAN23876.1
76         +     ?    Biflo_0010   protease                           clpC       AAN23877.1
0          -     ?    Biflo_0011   cytosine deaminase                 codA       AAN23878.1
0          +     ?    Biflo_0012   hypothetical protein weakly s...   BL0012     AAN23879.1
6          +     ?    Biflo 0013   proline/betaine transporter        proP       AAN23880.1
```

Figure 4.6: Example of an export data file created by GhostFam.

Chapter 5

Application

In this chapter, we describe the practical application of our algorithms to detect gene clusters in real genomic data. For the design of a gene cluster detection software, several issues regarding speed, accuracy, flexibility, and usability have to be weighted and optimized against each other in order to develop a tool of maximal practical usability. On this background, our tool for gene cluster detection in prokaryotic genomes (Gecko) is designed as one solution for the optimization of these given issues. Since a proper evaluation and interpretation of the results from a gene cluster detection approach still needs a large amount of experience and expert knowledge, Gecko is built to meet the requirements for a software especially used by the molecular biologist, thus bringing the information to those who can best interpret it.

An overview of the different consecutive processes performed for gene cluster detection with the Gecko software can be found in Figure 5.1. The program starts with the reading of the input data files containing the string representation of the genomes together with the functional annotation of each gene, if available. With these files, obtained from the COG database or the GhostFam tool, in an initial step the sequences to be analyzed as well as the algorithmic parameters have to be determined. After running the gene cluster detection algorithm on the chosen sequences, the result is postprocessed to allow an efficient analysis of the output. Since the generation of the input data is described in detail in Chapter 4, and the algorithmic part is discussed in Chapter 3, here we focus on the sequence selection and parameter input, as well as on the postprocessing (cluster grouping, pattern detection) and visualization.

5.1 Sequence Selection and Parameter Input

When starting a new Gecko session, the first decision to make is the selection whether to work on a gene family classification based on the COG database or created by the GhostFam tool. In both cases, the chosen data file is read, parsed, and the list of

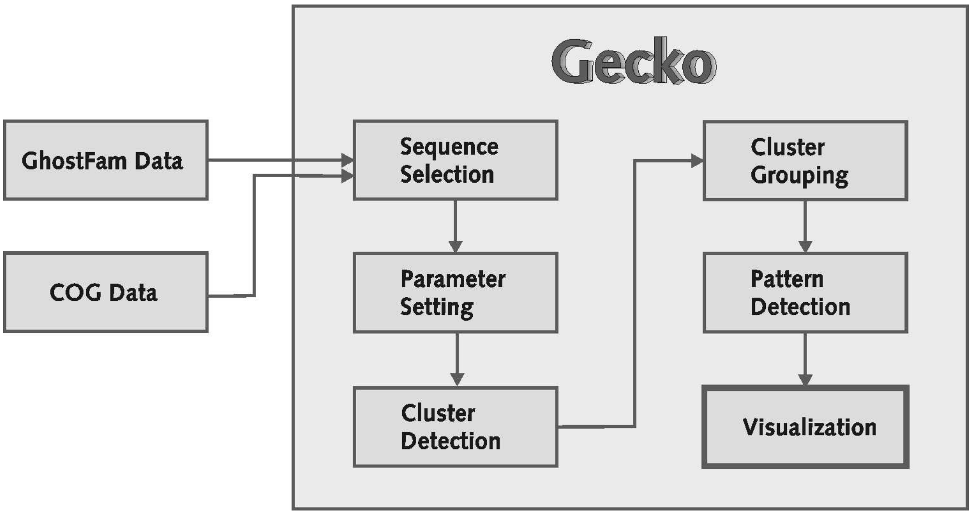


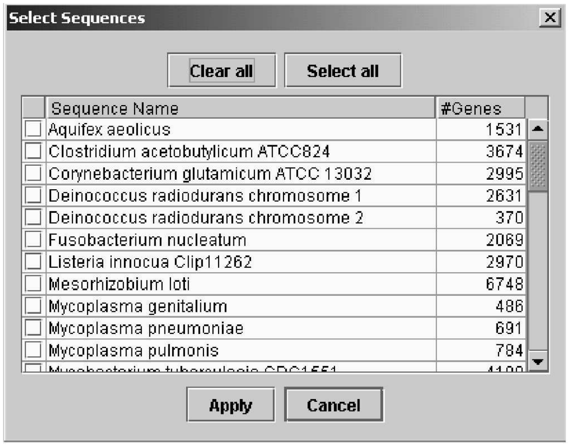
Figure 5.1: Data flow in GECKO. A new GECKO session always starts with the sequence selection from the imported data files. After choosing the algorithmic parameters, the cluster detection is started. In the cluster grouping and pattern detection step, the algorithmic output is postprocessed to achieve an optimized visualization.

contained genomes is displayed in a dialog. From this dialog, the sequences to be used for gene cluster detection have to be selected (see Figure 5.2, left).

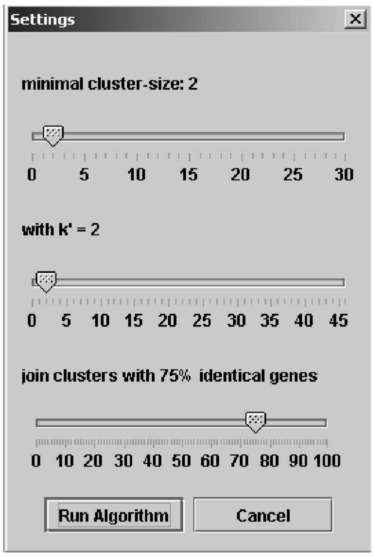
The choice of sequences to be used for the cluster detection is a crucial point for the evaluation of the reported results. As described in Section 3.1, the conservation of gene order depends of the evolutionary distances of the analyzed species. On the one hand, if the aim of the cluster detection is to find groups of genes that are probably functionally related, it is important to choose genomes of a sufficient phylogenetic distance. On the other hand, if the detected gene clusters are used to analyze the evolutionary development of a genomic region in a particular lineage, the computed results become more significant the more genomes from this lineage are used in the cluster detection.

Additionally in the selection dialog, for each genome the number of contained genes is listed. This number can give a first hint in indicating whether an organism is able to synthesize all its required products for the housekeeping functions by itself, or depends on an environment providing these metabolites. For the analysis of the presence or absence of particular gene clusters encoding elementary functions of the organism, the total number of genes in a genome also provides a certain estimation about the number of gene clusters to be expected.

After the composition of the genome set for the gene cluster detection algorithm is determined, four essential parameters (*minimal cluster size*, *minimum number of sequences* k', *identity rate*, and *uncharacterized genes*) customizing the detection algorithm have to be set (see Figure 5.2, right).



☐	Sequence Name	#Genes
☐	Aquifex aeolicus	1531
☐	Clostridium acetobutylicum ATCC824	3674
☐	Corynebacterium glutamicum ATCC 13032	2995
☐	Deinococcus radiodurans chromosome 1	2631
☐	Deinococcus radiodurans chromosome 2	370
☐	Fusobacterium nucleatum	2069
☐	Listeria innocua Clip11262	2970
☐	Mesorhizobium loti	6748
☐	Mycoplasma genitalium	486
☐	Mycoplasma pneumoniae	691
☐	Mycoplasma pulmonis	784
☐	Mycobacterium tuberculosis CDC1551	4190



minimal cluster-size: 2

with k' = 2

join clusters with 75% identical genes

Figure 5.2: The dialogs for the sequence selection (left), and the input of the algorithm parameters for the gene cluster detection (right).

The *minimal cluster size* can be set in a range of 1 up to 30 and describes the minimum number of different genes that have to be contained in a gene cluster. In practice, this parameter can be seen as a filter, removing all clusters from the output consisting of less genes than the given value for the parameter. The typical default value for the minimal cluster size is two. A smaller value ('1') would lead to the detection of a gene cluster even if only a single ortholog is conserved. If the value for the minimal cluster size is chosen above '2', observe that also for the reconstruction of fragmented gene clusters no cluster fragment of size below the minimal cluster size is available.

The *minimum number of sequences* (k') indicates, in how many sequences a gene cluster has to appear at least (see also Section 3.4.8). The range for the values of this parameter reaches from two up to the total number of sequences to be analyzed. A proper setting for this parameter is important from two different points of view. On the one hand, the value for k' linearly scales the runtime of the cluster detection algorithm, and on the other hand it significantly determines the interpretability of the reported result. If the results of the cluster detection are used for the prediction of gene functions due to their conserved neighborhood, it is important to choose a k' that is larger than the number of pairwise closely related species in the genome set. Otherwise, it can be expected that a large proportion of the reported gene clusters are found due to the close evolutionary relatedness of their genomes rather than due to their functional association.

The *identity rate* is essential in the cluster grouping step during the postprocessing of the algorithms output and therefore discussed in Section 5.2.

Finally, the parameter *uncharacterized genes* is a boolean value indicating whether a preprocessing of the input data file is performed, in which all genes that have no homolog in any other of the analyzed genomes are removed from their genomes. This preprocessing is especially useful if the prediction of the open reading frames (ORFs) of a genome is not reliable. In this case, an originally conserved genomic region may erroneously be represented interleaved by falsely predicted ORFs causing a disruption of the genomic neighborhood. Such interleaved regions can often no longer be detected by the applied algorithms. On the other hand, by deleting these genes from the genome, previously not directly adjacent genes now become immediate neighbors. This creation of new artificial neighborhoods can result in an increased number of falsely reported clusters, such that the decision whether or not to allow or remove the genes without homologs from the genomes has to be made for each input data file as the case arises.

5.2 Cluster Grouping

After running the gene cluster detection algorithm, for an efficient evaluation and interpretation of the results, it is necessary to perform a postprocessing of the reported gene clusters. The example in the algorithmic summary in Section 3.6 clearly shows the large amount of redundant clusters being reported, due to a missing restriction regarding a maximality criterion of the reported clusters. In the following, two postprocessing procedures are described to optimize the output of the gene cluster detection. In a first step, all non-maximal gene clusters are removed from the result, and the second procedure attempts to discover and reconstruct gene clusters that have been partially disrupted during the course of evolution.

5.2.1 Finding Maximal Gene Clusters

With the reduction of the reported gene clusters to only maximal clusters, it is possible to significantly reduce the amount of redundant output of the cluster detection algorithm without losing any important information. For a systematic filtering of redundant gene clusters, in the following we formally define the notion of maximality on gene clusters. Recall from the definition in Section 3.4 that a gene cluster is modelled by a common $\mathcal{CS}$-factor, and Algorithm CI, which is our selected algorithm for the gene cluster detection, outputs a common $\mathcal{CS}$-factor by its maximal $\mathcal{CS}$-locations from the corresponding genomes.

Definition (location set). Given a collection of k strings $\mathcal{S} = (S_1, S_2, \ldots, S_k)$ and a common $\mathcal{CS}$-factor C of $\mathcal{S}$, the set $L(C)$ of all maximal $\mathcal{CS}$-locations of C in the S_i, $1 \leq i \leq k$ is called its *location set.*

Note that in the location set of a common $\mathcal{CS}$-factor all $\mathcal{CS}$-locations from all strings are collected in a single set.

Definition (covering). A $\mathcal{CS}$-location $l := [i, j]_S$ *covers* a $\mathcal{CS}$-location $l' := [i', j']_S$ in the same string S, if and only if $i \leq i'$ and $j' \leq j$.

Example. Let $S_1 := (5, 1, 2, 3, 6)$ and $S_2 := (7, 3, 2, 1, 8)$, Algorithm CI outputs the common $\mathcal{CS}$-factors by their location set:

$$\begin{aligned}
\{1\} &: L(\{1\}) = \{[2, 2]_{S_1}, [4, 4]_{S_2}\},\\
\{2\} &: L(\{2\}) = \{[3, 3]_{S_1}, [3, 3]_{S_2}\},\\
\{3\} &: L(\{3\}) = \{[4, 4]_{S_1}, [2, 2]_{S_2}\},\\
\{1, 2\} &: L(\{1, 3\}) = \{[2, 3]_{S_1}, [3, 4]_{S_2}\},\\
\{2, 3\} &: L(\{2, 3\}) = \{[3, 4]_{S_1}, [2, 3]_{S_2}\},\\
\{1, 2, 3\} &: L(\{1, 2, 3\}) = \{[2, 4]_{S_1}, [2, 4]_{S_2}\}.
\end{aligned}$$

Obviously, the common $\mathcal{CS}$-factor of interest is $\{1, 2, 3\}$ since all other common $\mathcal{CS}$-factors are only subsets of it, and each $\mathcal{CS}$-location of their location sets is covered by a $\mathcal{CS}$-location in the location set of $\{1, 2, 3\}$.

With the definitions above, we can now give a formal description of the common $\mathcal{CS}$-factor of interest from the previous example:

Definition (maximal $\mathcal{CS}$-factor). Given the set of all common $\mathcal{CS}$-factors $\mathcal{C}(\mathcal{S})$ of a collection of strings $\mathcal{S}$, we call a common $\mathcal{CS}$-factor $C \in \mathcal{C}(\mathcal{S})$ a *maximal $\mathcal{CS}$-factor* if and only if there exists no $C' \in \mathcal{C}(\mathcal{S})$, $C' \neq C$, such that $C \subset C'$ and each $\mathcal{CS}$-location $l \in L(C)$ is covered by a $\mathcal{CS}$-location $l' \in L(C')$. We denote by $\mathcal{M}(\mathcal{S})$ the set of all maximal $\mathcal{CS}$-factors in $\mathcal{S}$.

Note that non-maximal $\mathcal{CS}$-factors can already be identified during the gene cluster detection, as soon as a larger, covering $\mathcal{CS}$-factor is found. That is why in our implementation, the removal of non-maximal $\mathcal{CS}$-factors is performed in parallel with the gene cluster detection, in order to save space.

5.2.2 Finding Fragmented Gene Clusters

In the next postprocessing step, gene clusters that might have been fragmented by genomic rearrangements over time are linked together. To achieve this, the set of all maximal $\mathcal{CS}$-factors $\mathcal{M}(\mathcal{S})$ is searched for $\mathcal{CS}$-factors that are supersets of other $\mathcal{CS}$-factors from $\mathcal{M}(\mathcal{S})$. The idea behind this postprocessing is that larger conserved regions of a few genomes are found fragmented in several other genomes. Nevertheless, the functions of the genes from these fragments can be expected to be still related to the functions of the genes from the larger conserved region.

Definition (primary $\mathcal{CS}$-factor). Given the set of all maximal $\mathcal{CS}$-factors $\mathcal{M}(\mathcal{S})$ of a collection of strings $\mathcal{S}$, a maximal $\mathcal{CS}$-factor $C \in \mathcal{M}(\mathcal{S})$ is called a *primary $\mathcal{CS}$-factor* if and only if there exists no $C' \in \mathcal{M}(\mathcal{S})$, $C' \neq C$, such that $C \subset C'$. We denote by $\mathcal{P}(\mathcal{S})$ the set of all primary $\mathcal{CS}$-factors in $\mathcal{S}$.

A primary $\mathcal{CS}$-factor is a gene cluster that is not a subset of any other reported gene cluster. On the other hand, in the set of all maximal clusters, there might exist clusters that are fragments (subclusters) of several primary clusters. For a better visualization of the whole collection of conserved gene clusters it is desirable to have the fragments associated to (all of) their primary clusters:

Definition (joined cluster set). Given a primary $\mathcal{CS}$-factor $C \in \mathcal{P}(\mathcal{S})$ of a collection of strings $\mathcal{S}$ and the set of all maximal $\mathcal{CS}$-factors $\mathcal{M}(\mathcal{S})$, the set $\mathcal{J}(C) := \{C' \in \mathcal{M}(\mathcal{S}) \mid C' \subseteq C\}$ is called the *joined cluster set* of C.

In the second step of the postprocessing phase, for each primary $\mathcal{CS}$-factor its joined cluster set is created.

Example. Given three sequences $S_1 = (1,2,3,5,4,1,2)$, $S_2 = (4,2,1,6,4,1)$, and $S_3 = (4,1,7,3,1,2)$ with $\mathcal{M}(\mathcal{S}) = \{\{1,2,3\},\{1,2\},\{1,2,4\},\{1,4\}\}$ for $k' = 2$ and $\mathcal{P}(\mathcal{S}) = \{\{1,2,3\},\{1,2,4\}\}$, the resulting joined cluster set are $\mathcal{J}(C)^1 = \{\{1,2,3\}, \{1,2\}\}$ and $\mathcal{J}(C)^2 = \{\{1,2,4\}, \{1,2\}, \{1,4\}\}$. Note, that $\{1,2\}$ is a maximal $\mathcal{CS}$-factor of both joined clusters.

For the practical use, another (even more) interesting aspect is the joining of cluster fragments containing a certain rate of genes that are not present in the maximal conserved region. The presence of such genes may indicate a change in the functional role of a gene or a whole gene cluster, or probably an incorrect homology classification. To model such extensions, we introduce a parameter p, referred to as *identity rate*, that allows a formal definition for joining gene clusters containing some non-matching genes:

Definition (p-primary $\mathcal{CS}$-factor). Given the set of all maximal $\mathcal{CS}$-factors $\mathcal{M}(\mathcal{S})$ of a collection of strings $\mathcal{S}$ and a non-negative identity rate p, $0 \leq p \leq 1$, a maximal $\mathcal{CS}$-factor $C \in \mathcal{M}(\mathcal{S})$ is called a *p-primary $\mathcal{CS}$-factor* if and only if there exists no $C' \in \mathcal{M}(\mathcal{S})$, $C' \neq C$, such that $C' \subset_p C$, where $A \subset_p B$ if and only if $|A \cap B| \geq p \cdot |A|$. We denote by $\mathcal{P}_p(\mathcal{S})$ the set of all p-primary $\mathcal{CS}$-factors in $\mathcal{S}$.

Similar to the combination of a primary $\mathcal{CS}$-factor with its conserved fragments to a joined cluster set, we model a set of p-identical clusters as follows:

Definition (p-joined cluster set). Given a p-primary $\mathcal{CS}$-factor $C \in \mathcal{P}_p(\mathcal{S})$ of a collection of strings $\mathcal{S}$ and the set of all maximal $\mathcal{CS}$-factors $\mathcal{M}(\mathcal{S})$, the set $\mathcal{J}_p(C) := \{C' \in \mathcal{M}(\mathcal{S}) \mid C' \subset_p C\}$ is called the *p-joined cluster set* of C.

In the example above, $\mathcal{J}(C)^1$ and $\mathcal{J}(C)^2$ become a single p-joined cluster set for identity rates $p \leq 0.66$. Note that for $p = 1$, a p-joined cluster set becomes a joined cluster set as defined above.

The final result of the second postprocessing phase is the creation of a list of all p-joined cluster sets from the output of the gene cluster detection. In this list, each maximal $\mathcal{CS}$-factor is present as an element of at least one p-joined cluster set. This list of all p-joined cluster sets is the final result of our gene cluster detection approach.

In the following, we will show how to highlight gene clusters, showing particular biological characteristics, and we will introduce into the gene cluster visualization, which is besides the algorithmic part the second central aspect of Gecko.

5.3 Pattern Detection

The search for characteristic biological patterns in the list of all computed gene clusters (p-joined cluster sets), is a first step towards an automatic classification of the significance of a reported gene cluster. A sorting by significance (however defined on gene clusters) allows a more efficient evaluation of the list of all reported clusters. Obviously, the number of genes contained in a cluster, and the number of genomes containing the cluster are objective criteria to estimate how interesting a reported cluster might be. Therefore, in the visualization (see next section) the list of the clusters is shown sorted first by the number of genes a cluster contains and second by the number of genomes it covers. This measurement alone turned out be a first acceptable way of sorting the clusters, but it also shows disadvantages if the content (the genes) of the gene clusters is totally neglected.

Therefore, three different biologically motivated patterns regarding the content of a gene cluster are described in the following. The detection of these patterns in the located clusters is performed subsequent to the last postprocessing step.

5.3.1 Gene Replacement

The first gene cluster pattern, which is especially straightforward to model and locate, is the *gene replacement pattern*. The occurrence of such a pattern strongly points to small

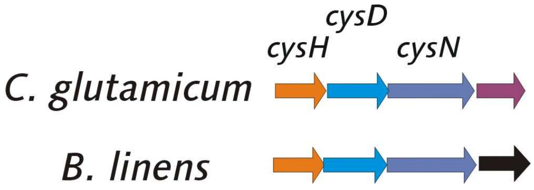


Figure 5.3: Gene replacement pattern between *C. glutamicum* and *B. linens*. The gene cluster in both genomes consists of three conserved genes (*cysH, cysD, cysN*) and one different gene (*cysY*, pink) in *C. glutamicum* and (*cysG*, black) in *B. linens*.

deviations in the functional role of a gene cluster, consider the example in Figure 5.3.

In this example, a gene cluster of *C. glutamicum* and *B. linens* clearly depicts the replacement of the last gene in the gene cluster from these genomes. The functional role of the whole cluster in both organisms is located in the sulfur utilization, but the deviation in the last gene of the cluster indicates that a small change in the role of the cluster has occurred during their divergence in evolution. In fact, a more detailed analysis of the sulfur utilization process of the two organisms revealed that they use different types of sulfur in their utilization process. While *C. glutamicum* uses the *cysY* gene to reduce sulfite, at the same point *B. linens* uses a *cysH* gene to add a methyl group to the so far created sulfur metabolite.

For a precise description of what we call a gene replacement pattern, the following definition formally describes our model of this pattern:

Definition (gene replacement pattern). Given a p-joined cluster set $\mathcal{J}_p(C) := \{C_1, C_2, \ldots, C_n\}$ of n maximal $\mathcal{CS}$-factors. A *gene replacement pattern* occurrs in $\mathcal{J}_p(C)$ if and only if there exists a pair (C_i, C_j) of maximal $\mathcal{CS}$-factors from $\mathcal{J}_p(C)$ with $size(C_i) = size(C_j) \geq 3$ and $|C_i \backslash C_j| = 1$.

The detection of a gene replacement pattern in the list of all gene clusters can be performed in a straightforward fashion. Each p-joined cluster set is searched for maximal $\mathcal{CS}$-factors of the same number of (different) genes. All maximal $\mathcal{CS}$-factors of the same size (and at least three genes) are compared pairwise whether their set of genes is identical except for one position. If such a pattern is found in a cluster, the cluster as well as the pair of maximal $\mathcal{CS}$-factors are marked for later visualization (see Section 5.4).

5.3.2 Cluster Separation

A further topic in the analysis of the content of a gene cluster is the search for gene clusters that in some genomes contain the genes in one single contiguous region, and that are divided into two or more regions in another genome. This separation of a gene cluster in

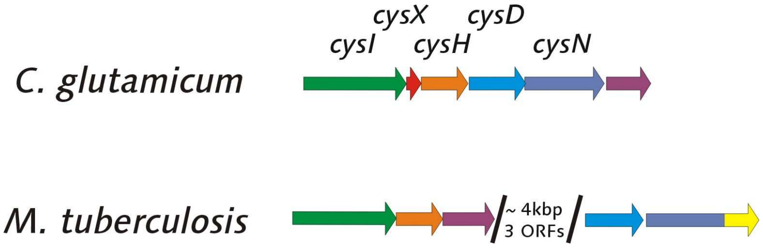


Figure 5.4: Cluster Separation Pattern between *C. glutamicum* and *M. tuberculosis*. Both genomes hold the genes for sulfur utilization in conserved genomic regions. While in *M. tuberculosis* this region is divided into two parts separated by three genes not associated with the sulfur utilization, in *C. glutamicum* this region is completely conserved.

two or more pieces may indicate that the biological mechanisms in the species are similar, but the encoded products are needed at different points in time.

In the example in Figure 5.4, the operon for sulfur utilization in *C. glutamicum* is found in one single conserved region. This organization implies that the encoded products are required in direct succession. On the other hand, in *M. tuberculosis* this cluster is separated into two different parts, indicating that the sulfur utilization is performed in two steps not necessarily in direct consecution. Of course, a second possible explanation for such pattern could be found in the evolutionary distance of the organisms. If the involved organisms are closely related, the proximity of the genes might still be remaining from their recent common ancestor, and the genomes might not have had enough time to be shuffled more than by the single disruption.

For an efficient search for such structural characteristic, we define a *cluster separation pattern* as follows:

Definition (cluster separation pattern). Given a p-joined cluster set $\mathcal{J}_p(C) := \{C_1, C_2, \ldots, C_n\}$ of n maximal $\mathcal{CS}$-factors. Let $\mathcal{L} := \bigcup_{i=1}^{n} L(C_i)$ be the set of all $\mathcal{CS}$-locations and $\mathcal{G} := \bigcup_{i=1}^{n} C_i$ the set of all genes of $\mathcal{J}_p(C)$. A *cluster separation pattern* occurrs in $\mathcal{J}_p(C)$, if and only if there exists a $\mathcal{CS}$-location $[i,j]_S \in \mathcal{L}$ with $\mathcal{CS}([i,j]_S) = \mathcal{G}$ and a pair of $\mathcal{CS}$-locations $([i_1,j_1]_{S'}, [i_2,j_2]_{S'})$, $S \neq S'$, with $\mathcal{CS}([i_1,j_1]_{S'}) \cup \mathcal{CS}([i_2,j_2]_{S'}) = \mathcal{G}$ and $\mathcal{CS}([i_1,j_1]_{S'}) \neq \mathcal{G} \neq \mathcal{CS}([i_2,j_2]_{S'})$.

Less formally spoken, a cluster separation pattern occurs if a p-joined cluster set contains a $\mathcal{CS}$-location in one genome and a pair of $\mathcal{CS}$-locations in a second genome that cover all the genes from the cluster. In the pair of $\mathcal{CS}$-locations from the second sequence, none of the $\mathcal{CS}$-locations alone is allowed to contain all genes from the cluster, but it is not explicitly forbidden that those $\mathcal{CS}$-locations may partially overlap in their gene content.

The detection of this pattern in the output of all p-joined clusters is performed in two

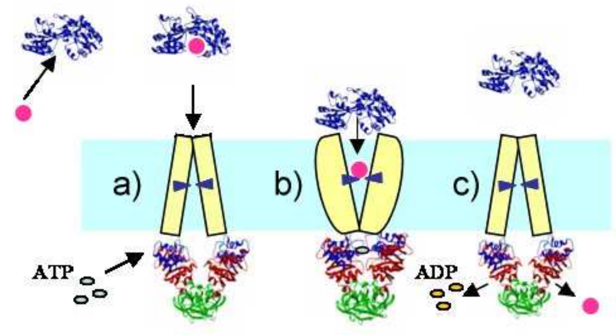


Figure 5.5: Transporting a signal molecule through the cell wall by an ABC transporter. (a) The substrate or signal molecule (red circle) binds at the substrate binding site (blue) outside the cell, while the opening of the transport channel in the cell wall (light-blue bar) requires ATP inside the cell. (b) Transport through the cell wall. (c) Release of the substrate inside the cell, while ATP is reduced to ADP.

consecutive steps. In a first step, it is tested for each cluster whether it contains a $\mathcal{CS}$-location that covers all genes from the cluster. If such a $\mathcal{CS}$-location is found, the set of all locations is evaluated to determine if any combination of two $\mathcal{CS}$-locations in another sequence contains all genes of the p-joined cluster set. Again like the first pattern, the successful location of the second pattern is indicated by a marked field in the final list of all clusters shown in the visualization.

5.3.3 Gene Duplication

The last pattern describing the content of a gene cluster detected by the GECKO program is the *gene duplication pattern*. This pattern simply indicates that the detected gene cluster contains at least one gene more than once in a single genome, i.e. at least one pair of paralogs is present. In the following, we will distinguish between two different types of gene duplication patterns. The first type concerns clusters in which a gene duplication occurs inside a single conserved genomic region, where as the second type of clusters describes a region of internal gene duplication, where a whole set of genes, maybe an operon, is duplicated.

A typical example for the occurrence of both types of the gene duplication pattern are the so called *ABC transporters*. ABC transporters are used to transfer molecular signals through the membranes (cell walls) which provide a natural barrier to movement of polar molecules. The ABC transporters concentrate and transport molecules into or out of a cell, but require a source of energy. The mechanism of an ABC transporter is shown in Figure 5.5[1].

[1] Figure from: http://chen.bio.purdue.edu/projects.html

An ABC transporter consists of at least three different components: An ATPase component, the membrane helices and a substrate binding site. The substrate binding site binds a substrate making the membrane permeable for these molecules. The permeability is realized by the membrane helices which build a channel for the transportation of molecules through the membrane. The required energy for the transportation process is delivered from the ATPase component based on a reduction of ATP to ADP.

Since this transport through a cell wall is a very elementary process that is frequently required in many organisms, usually there are several copies of the set of genes present in each genome. Therefore, it is of special interest to locate these gene clusters that contain regions of internal duplication, since these regions can be predicted to encode a similar function. On the other hand, a gene cluster encoding for an ABC transport system is also a good example for a gene cluster showing the occurrences of a paralogous gene inside a single conserved region. Depending on the type of molecule to be transported into the cell, the number of required membrane helices or the amount of required energy can differ. A preferable solution for an organism, e.g. to ensure that enough energy to perform the required function is available, is the conservation of a set of genes, where the ATPase component is encoded twice, i.e. as a paralog inside a conserved cluster.

Based on the definition of a gene cluster as a p-joined cluster set, we can model the gene duplication pattern as follows:

Definition (gene duplication pattern). Given a p-joined cluster set $\mathcal{J}_p(C) := \{C_1, C_2, \ldots, C_n\}$ of n maximal $\mathcal{CS}$-factors and the set of all its $\mathcal{CS}$-locations $\mathcal{L} := \bigcup_{i=1}^{n} L(C_i)$. A *gene duplication pattern* occurs in $\mathcal{J}_p(C)$ if and only if there exists a $\mathcal{CS}$-location $[i,j]_S \in \mathcal{L}$ with $|\mathcal{CS}([i,j]_S)| < 1 + j - i$, or if there exists a location set $L(C_i)$ of a maximal $\mathcal{CS}$-factor in $\mathcal{J}_p(C)$ that contains two $\mathcal{CS}$-locations in the same sequences S, $[a,b]_S \neq [a',b']_S$.

In the first case of the definition it is tested whether the total number of genes in a conserved region of a genome exceeds the number of *different* genes of the corresponding gene cluster. If so, the only possible explanation is a gene duplication inside the single conserved region. In the second case it is tested whether a conserved fragment (maximal $\mathcal{CS}$-factor) of the cluster has more than one occurrence ($\mathcal{CS}$-location) in a single sequence. If this is true, a region of internal duplication in the gene cluster is present.

In practice, the detection of both variants of the gene duplication pattern can be performed in a straightforward fashion. For the reason of time efficiency, the second type of the pattern (a region of internal duplication) is tested first. For a single maximal $\mathcal{CS}$-factor only the number of $\mathcal{CS}$-locations per sequence have to be counted. If there is more than one, the p-joined cluster fulfills the gene duplication pattern. If this first test is negative, then each $\mathcal{CS}$-location is tested whether its length exceeds the number of different genes in the maximal $\mathcal{CS}$-factor.

5.4 Visualization

One of the most important requirements for a tool that extracts information from a large data set is the capability to visualize the computed results in a style which is appropriate for the needs of the expected users. The Gecko tool was designed in close cooperation with molecular biologists, thus allowing to develop a user interface for the algorithmic part and a visualization of the computed results, meeting the requirements of that group of scientists as optimal as possible.

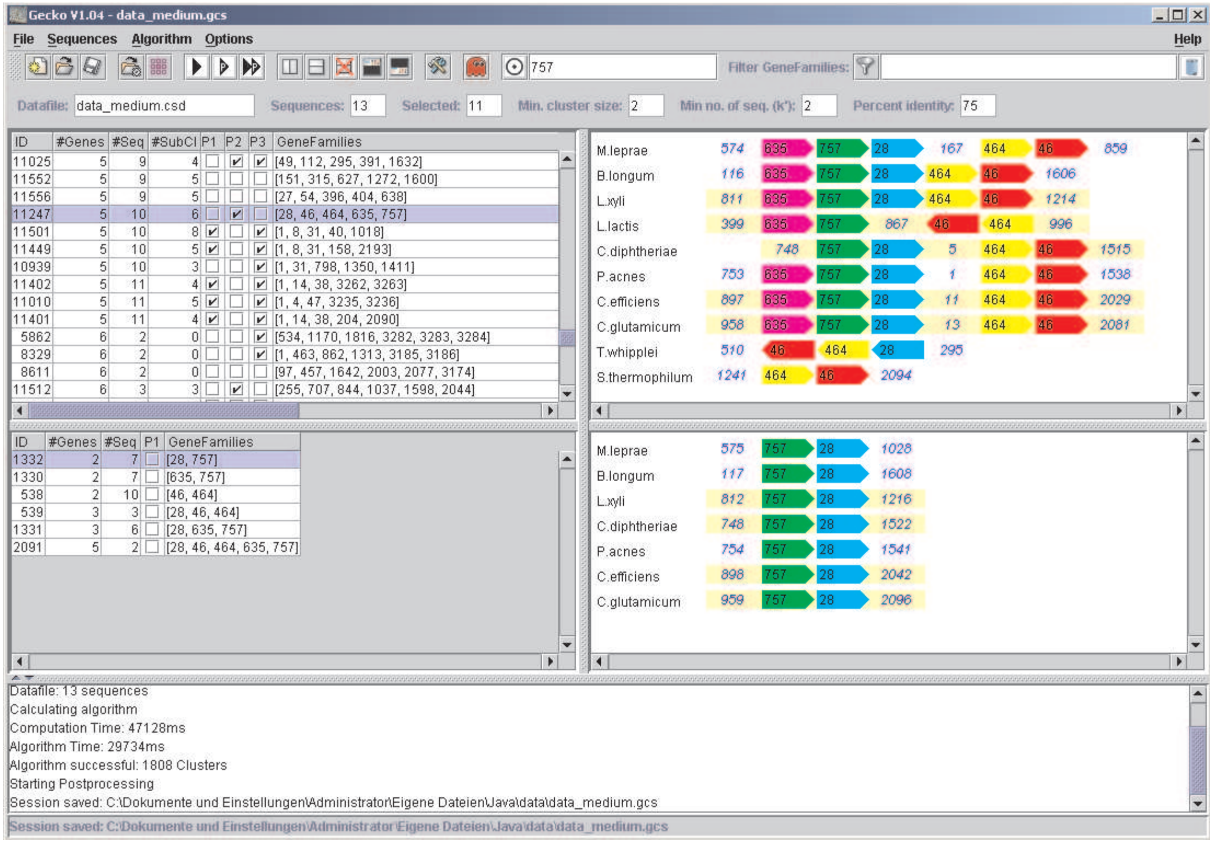
As described in the previous section, the final result of an application of the gene cluster detection algorithms together with the postprocessing and the pattern detection on a given set of genomes is a list of gene clusters, ordered by their size (number of different genes in the cluster) and the number of sequences it occurs in. From the pattern detection, each cluster contains three different marks indicating if the corresponding pattern can be found inside the cluster. In the main window of Gecko, the sorted list of all computed gene clusters can be found in the upper left table, see Figure 5.6.

From this list, the user can select a single gene cluster for a more detailed inspection. The selected cluster is drawn in a graphical representation right next to the table. In this representation of a gene cluster, each conserved gene from the cluster is drawn by an arrow, where the pointing direction of an arrows indicates on which strand of the genome the gene is encoded. The numbers inside the arrows show the gene family or COG cluster a gene belongs to. For a better overview, the arrows are colored, such that all genes belonging to the same gene family in a cluster are always drawn in the same color. The blue numbers in front, behind and between the arrows show how many genes not in the cluster can be found at the corresponding position.

Since a complete gene cluster (p-joined cluster set) is often composed of a set of conserved fragments (maximal $\mathcal{CS}$-factors), it might be of certain interest to analyze a single fragment of the whole cluster in detail. Therefore, a table in the lower left of the main window contains the list of all such fragments from which a single fragment can be selected for a graphical representation below the graphic of the main cluster.

A further helpful feature available in both graphics is the option to draw the conserved regions centered by one gene family. The number of the family can be entered in the corresponding field in the toolbar, or it can be determined by right-clicking the mouse on the gene family. In the centering process, the genomes are shifted to the right or left, such that the first occurrence of the centering gene in a genome is always found in one column in the graphic. If the genes in that column do not point in the same direction, the orientation of the whole genome is flipped. This change in the coding direction is indicated by a light yellow background for the particular genome.

For the detailed evaluation of a single conserved region it is possible to get access to the functional annotation as well as additional information about the gene name and the


Gecko V1.04 - data_medium.gcs
File
Sequences
Algorithm
Options
Help
Filter GeneFamilies:
M.leprae 574 635 757 28 167 464 46 859
B.longum 116 635 757 28 464 46 1606
L.xyli 811 635 757 28 464 46 1214
L.lactis 399 635 757 867 46 464 996
C.diphtheriae 748 757 28 5 464 46 1515
P.acnes 753 635 757 28 1 464 46 1538
C.efficiens 897 635 757 28 11 464 46 2029
C.glutamicum 958 635 757 28 13 464 46 2081
T.whipplei 510 46 464 28 295
S.thermophilum 1241 464 46 2094
M.leprae 575 757 28 1028
B.longum 117 757 28 1608
L.xyli 812 757 28 1216
C.diphtheriae 748 757 28 1522
P.acnes 754 757 28 1541
C.efficiens 898 757 28 2042
C.glutamicum 959 757 28 2096

757

Datafile: data_medium.csd Sequences: 13 Selected: 11 Min. cluster size: 2 Min no. of seq. (k'): 2 Percent identity: 75

ID	#Genes	#Seq	#SubCl	P1	P2	P3	GeneFamilies
11025	5	9	4	□	☑	☑	[49, 112, 295, 391, 1632]
11552	5	9	5	□	□	□	[151, 315, 627, 1272, 1600]
11556	5	9	5	□	□	□	[27, 54, 396, 404, 638]
11247	5	10	6	□	☑	□	[28, 46, 464, 635, 757]
11501	5	10	8	☑	□	☑	[1, 8, 31, 40, 1018]
11449	5	10	5	☑	□	☑	[1, 8, 31, 158, 2193]
10939	5	10	3	□	□	☑	[1, 31, 798, 1350, 1411]
11402	5	11	4	☑	□	☑	[1, 14, 38, 3262, 3263]
11010	5	11	5	☑	□	☑	[1, 4, 47, 3235, 3236]
11401	5	11	4	☑	□	☑	[1, 14, 38, 204, 2090]
5862	6	2	0	□	□	☑	[534, 1170, 1816, 3282, 3283, 3284]
8329	6	2	0	□	□	☑	[1, 463, 862, 1313, 3185, 3186]
8611	6	2	0	□	□	□	[97, 457, 1642, 2003, 2077, 3174]
11512	6	3	3	□	☑	□	[255, 707, 844, 1037, 1598, 2044]

ID	#Genes	#Seq	P1	GeneFamilies
1332	2	7	□	[28, 757]
1330	2	7	□	[635, 757]
538	2	10	□	[46, 464]
539	3	3	□	[28, 46, 464]
1331	3	6	□	[28, 635, 757]
2091	5	2	□	[28, 46, 464, 635, 757]

```
Datafile: 13 sequences
Calculating algorithm
Computation Time: 47128ms
Algorithm Time: 29734ms
Algorithm successful: 1808 Clusters
Starting Postprocessing
Session saved: C:\Dokumente und Einstellungen\Administrator\Eigene Dateien\Java\data\data_medium.gcs
```

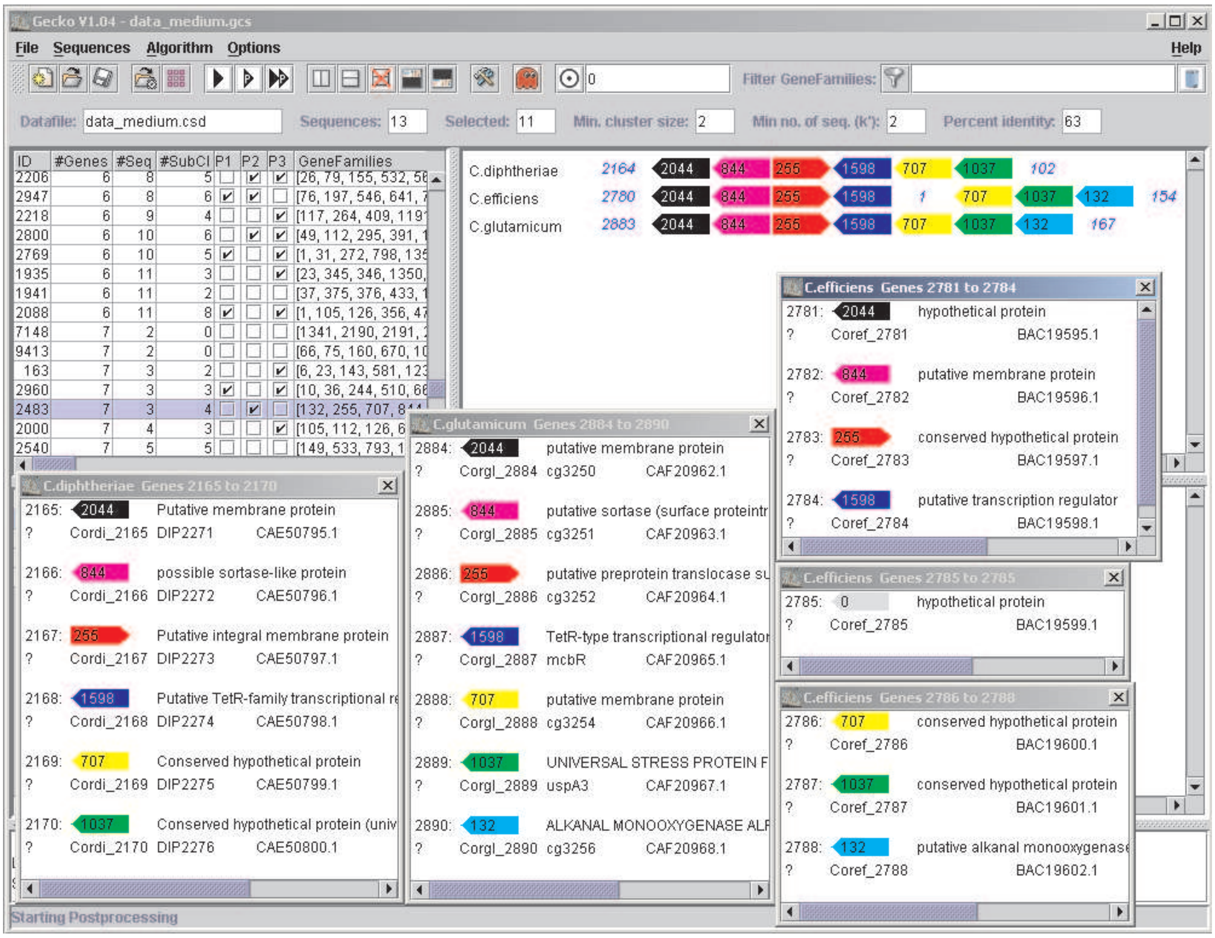
Session saved: C:\Dokumente und Einstellungen\Administrator\Eigene Dateien\Java\data\data_medium.gcs

Figure 5.6: The main window of Gecko. The table in the upper left contains the list of all gene clusters calculated from the “data_medium.csd” data file with the parameters displayed in the toolbar. For the graphical representation, the gene cluster no. 11247 is selected and drawn right next to the table. Here the conserved regions are centered along the gene family 757 (green color). As marked in the checkbox ’P2’ and easily detectable in the graphic, this cluster shows the cluster separation pattern, since in *B. linens* and *L. xyli* the cluster is found in a single conserved region which is separated into two parts in many other genomes, e.g. in *C. glutamicum*. Additionally, a marked checkbox ’P1’ would indicate the occurrence of the gene replacement pattern, whereas ’P3’ is marked when a gene duplication is found in the cluster. The six cluster fragments of which the complete cluster is composed are shown in the lower left table.

functional COG category of all genes in that region by a left-click on one gene of this region. Therefore, a new dialog appears displaying all available information about the genes from this region (see Figure 5.7).

Another advantage of using the GhostFam tool for data preparation is the option to manually verify the grouping of the genes into their families of homologs given a located gene cluster. Here a typical scenario is the presence of an additional or the absence of an expected gene in a conserved gene cluster. To clarify the role of such a gene, the result of the classification in GhostFam can be recalled to evaluate a possible false family assignment or to discover falsely predicted ORFs. Especially the latter case was often found during the application of the gene cluster detection of real genomic data (see Chapter 6). Here, our data for the genome of *C. efficiens*, seemed to contain a larger number of such 'additional' ORFs.

For a fast location of gene clusters containing one or more gene families of interest, the toolbar also provides a filter option for the list of all clusters. If a located gene cluster is of particular interest, e.g. for documentation, Gecko additionally offers the option to export the graphical representation to a compressed bitmap file.



Gecko V1.04 - data_medium.gcs

Datafile: data_medium.csd Sequences: 13 Selected: 11 Min. cluster size: 2 Min no. of seq. (k'): 2 Percent identity: 63

ID	#Genes	#Seq	#SubCl	P1	P2	P3	GeneFamilies
2206	6	8	5	□	☑	☑	[26, 79, 155, 532, 56
2947	6	8	6	☑	☑	□	[76, 197, 546, 641, 7
2218	6	9	4	□	□	☑	[117, 264, 409, 1191
2800	6	10	6	□	☑	☑	[49, 112, 295, 391, 1
2769	6	10	5	☑	□	☑	[1, 31, 272, 798, 135
1935	6	11	3	□	□	☑	[23, 345, 346, 1350,
1941	6	11	2	□	□	□	[37, 375, 376, 433, 1
2088	6	11	8	☑	□	☑	[1, 105, 126, 356, 47
7148	7	2	0	□	□	□	[1341, 2190, 2191, 2
9413	7	2	0	□	□	□	[66, 75, 160, 670, 10
163	7	3	2	□	□	☑	[6, 23, 143, 581, 123
2960	7	3	3	☑	□	☑	[10, 36, 244, 510, 66
2483	7	3	4	□	☑	□	[132, 255, 707, 844
2000	7	4	3	□	□	☑	[105, 112, 126, 6
2540	7	5	5	□	□	□	[149, 533, 793, 1

```
C.diphtheriae   2164  2044  844  255  1598        707  1037        102
C.efficiens     2780  2044  844  255  1598    1   707  1037  132         154
C.glutamicum    2883  2044  844  255  1598        707  1037  132   167
```

C.diphtheriae Genes 2165 to 2170

```
2165:  2044          Putative membrane protein
?      Cordi_2165  DIP2271       CAE50795.1

2166:  844           possible sortase-like protein
?      Cordi_2166  DIP2272       CAE50796.1

2167:  255           Putative integral membrane protein
?      Cordi_2167  DIP2273       CAE50797.1

2168:  1598          Putative TetR-family transcriptional re
?      Cordi_2168  DIP2274       CAE50798.1

2169:  707           Conserved hypothetical protein
?      Cordi_2169  DIP2275       CAE50799.1

2170:  1037          Conserved hypothetical protein (univ
?      Cordi_2170  DIP2276       CAE50800.1
```

C.glutamicum Genes 2884 to 2890

```
2884:  2044          putative membrane protein
?      Corgl_2884  cg3250        CAF20962.1

2885:  844           putative sortase (surface proteintr
?      Corgl_2885  cg3251        CAF20963.1

2886:  255           putative preprotein translocase su
?      Corgl_2886  cg3252        CAF20964.1

2887:  1598          TetR-type transcriptional regulator
?      Corgl_2887  mcbR          CAF20965.1

2888:  707           putative membrane protein
?      Corgl_2888  cg3254        CAF20966.1

2889:  1037          UNIVERSAL STRESS PROTEIN F
?      Corgl_2889  uspA3         CAF20967.1

2890:  132           ALKANAL MONOOXYGENASE ALF
?      Corgl_2890  cg3256        CAF20968.1
```

C.efficiens Genes 2781 to 2784

```
2781:  2044          hypothetical protein
?      Coref_2781                BAC19595.1

2782:  844           putative membrane protein
?      Coref_2782                BAC19596.1

2783:  255           conserved hypothetical protein
?      Coref_2783                BAC19597.1

2784:  1598          putative transcription regulator
?      Coref_2784                BAC19598.1
```

C.efficiens Genes 2785 to 2785

```
2785:  0             hypothetical protein
?      Coref_2785                BAC19599.1
```

C.efficiens Genes 2786 to 2788

```
2786:  707           conserved hypothetical protein
?      Coref_2786                BAC19600.1

2787:  1037          conserved hypothetical protein
?      Coref_2787                BAC19601.1

2788:  132           putative alkanal monooxygenase
?      Coref_2788                BAC19602.1
```

Starting Postprocessing

Figure 5.7: Annotation windows in Gecko. For each conserved region in a genome, an annotation window can be displayed. In this window, all information about the functional role of a gene together with its gene name, TrEMBL-ID, and location on the genome is displayed. If the input data file is based on the COG database, also the COG functional category is shown. These windows are also available for the interleaving regions of a gene cluster, here displayed for the single interleaving gene *coref 2705* of *C. efficiens*. Since this gene has no homolog in any other sequence, i.e. its identifier is '0', this gene is probably a falsely predicted ORF.

Chapter 6

Experimental Results

In the first part of this chapter, we will evaluate the performance of the gene cluster detection algorithm (Algorithm CI) implemented in the Gecko software. To prove that the algorithm runs within the claimed time complexity, the evaluation was performed on an artificially created data set, especially designed to test the worst case running time in different scenarios.

Subsequently in this chapter, we will present some results from the application of the gene cluster detection algorithm to real genomic data. Here we will show the positive effect on the quality and number of the detected gene clusters by modelling genomes based on the representation as strings instead of permutations. Afterwards, we evaluate the distribution of structural patters in the detected gene clusters to test whether these patterns are helpful criteria to express a special significance for a particular cluster.

Unfortunately, due to the lack of experimentally verified reference data, a proof of the correctness of each reported gene cluster by hand is not feasible. Therefore, in the last part of this chapter we will compare some results obtained by Gecko with a gene cluster (the tryptophan biosynthesis operon) that is well known from the literature, and a cluster that is intensively studied – also by wet-lab experiments – at the Institute of Genome Research at Bielefeld University.

6.1 Running Time Evaluation

To show that our implementation of the gene cluster detection algorithm (Algorithm CI) runs in the claimed time complexity, we tested the three different scenarios (increasing string length n, increasing number of strings k, and decreasing number of required strings k') on an artificially created data set. All tests were performed on a single 900 MHz UltraSPARC-III+ CPU running the Solaris 9 operating system. To ensure that the Java Virtual Machine has enough main memory available to run Gecko, it was started with the option `-mx1500m`.

length n	time t $[ms]$	$\sqrt{t}$	$\frac{n}{\sqrt{t}}$
100	8	2.83	35.35
200	29	5.39	37.13
500	195	13.96	35.80
1,000	782	27.96	35.76
2,000	3,360	57.97	34.50
3,000	7,561	86.95	34.51
4,000	13,564	116.47	34.34
5,000	22,990	151.63	32.97
10,000	83,550	289.05	34.59

Table 6.1: Running time evaluation of Algorithm CI for a fixed $k = 2$ and varying string lengths n. The second column shows the absolute running time of Algorithm CI for each chosen n, and in the third column the square root of this value is shown. The fourth column shows the ratio of computed sequence length per time.

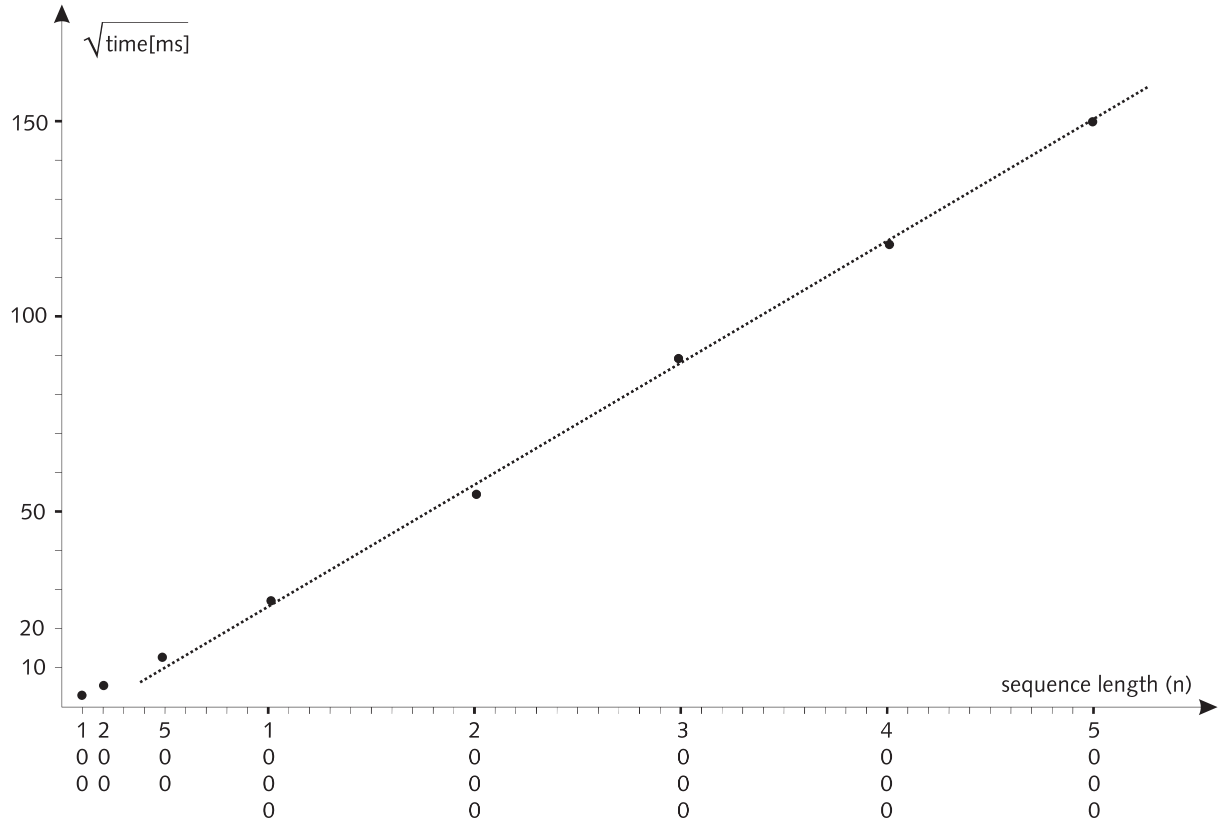


Figure 6.1: The square root of the running time of Algorithm CI is plotted for the different lengths of the input strings. All data points can be found on a straight line (dotted), proving the quadratic correlation of running time and string length.

The first test was designed to prove that the worst case running time of Algorithm CI, applied to a single pair of strings (genomes), scales quadratically with $n = \max(|S_i|)$, $1 \leq i \leq k$, the maximal number of characters contained in the longest string. Therefore, we created two strings of length n, where $S_1 = (1, 2, ..., n)$ and $S_2 = (n+1, n+2, ..., 2n)$. This can be considered the worst-case for Algorithm CI, since each $\mathcal{CS}$-location of the two strings is maximal, i.e. has to be considered for the detection of a common $\mathcal{CS}$-factor (see Section 3.4.5). Since the two strings contain no common character, obviously no common $\mathcal{CS}$-factor will be detected. This setting was chosen to avoid an influence of the first postprocessing procedure, which runs in parallel with the cluster detection, on the measured running time of the algorithm.

For the evaluation, we chose different string lengths from 100 up to 10,000 characters. The running time of Algorithm CI for the different values of n is shown in Table 6.1 and was averaged over three test runs. These values and their graphical representation in Figure 6.1 clearly depict the quadratic increase in the running time of the cluster detection algorithm with a linear increase in the string length.

The second test was designed to prove that Algorithm CI, if applied to a set of k strings, runs in $O(kn^2)$ time to compute all common $\mathcal{CS}$-factors occurring in all k strings. Therefore, we applied Algorithm CI to a test set of k strings, each of length $n = 1000$ and the minimum number of strings k' was fixed to k. Like in the first test scenario, the k strings in the test set are created in a way such that two strings have no common character, i.e. no common $\mathcal{CS}$-factor is detectable:

$$S_l = (n \cdot (l-1) + 1, n \cdot (l-1) + 2, \ldots, n \cdot (l-1) + n), \quad 1 \leq l \leq k.$$

Note that for the application of Algorithm CI to each pair of strings from this set the alphabet size $|\Sigma|$ does not exceed $2n$, even if the maximal number used for the identification of the characters in the strings is $n \cdot k$ (the last character in S_k). Therefore, in the implementation of Algorithm CI the characters from a pairwise comparison are always mapped onto characters from $\Sigma := \{1, 2. \ldots, 2n\}$.

Again, the shown running time for each k in Table 6.2 is the average of three individual tests. The values from the table are visualized in Figure 6.2. As clearly depicted in the figure, for $k \geq 20$ the running time increases linearly with the number of selected strings, proving together with the first test the predicted time complexity of $O(kn^2)$. For less than 20 strings, the preprocessing of the first string that creates the tables NUM and POS (see Section 3.4.5) contributes a small constant amount of time to the overall running time, such that for these measures the running time is above the straight line in the Figure 6.2.

In the third test scenario, we will show that our implementation of the gene cluster detection algorithm in Gecko finds all gene clusters that occur in k' out of k strings in $O(n^2k(1+k-k'))$ time. Therefore, we chose a test set of $k = 20$ strings of length $n = 2000$

strings k	time t $[ms]$	$\frac{k}{t[s]}$
2	1304	1.53
5	1351	3.70
10	2760	3.62
20	4667	4.28
40	8072	4.95
60	11716	5.12
80	15037	5.32
100	18726	5.34
150	31054	4.83
200	37884	5.28
500	93611	5.34

Table 6.2: Running time evaluation of Algorithm CI for a fixed string length $n = 1,000$ and varying number of strings k. Since $k' = k$ the common $\mathcal{CS}$-factors must occur in each of the given strings. In the last column, the string per time ratio is calculated.

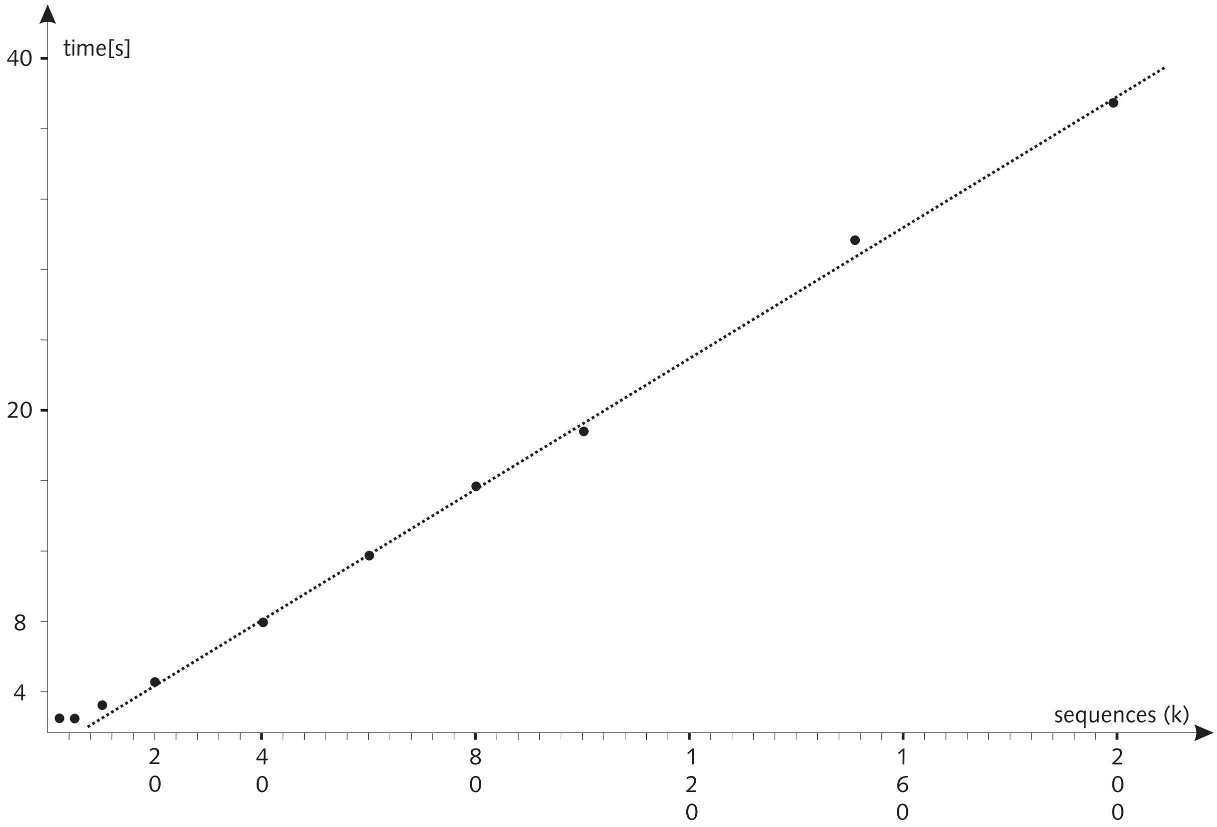


Figure 6.2: The figure shows a plot of the running time of Algorithm CI for the different numbers of strings k of length $n = 1000$.

k'	time $t[s]$	est. appl. a_e	$\frac{a_e}{t}$	perf. appl. a_p	$\frac{a_p}{t}$
2	385.2	380	0.986	209	0.543
4	372.1	340	0.914	204	0.548
6	350.6	300	0.856	195	0.556
8	324.9	260	0.800	182	0.560
10	295.3	220	0.745	165	0.559
12	257.6	180	0.699	144	0.559
14	207.5	140	0.675	119	0.573
16	155.5	100	0.643	90	0.579
18	97.5	60	0.615	57	0.584
20	35.2	20	0.568	20	0.568

Table 6.3: Running time evaluation of Algorithm CI for a fixed string length $n = 2,000$ and a fixed number of strings $k = 20$, varying the minimum number of required strings k' for the detection of a gene cluster. Column three shows the number of estimated pairwise applications a_e of Algorithm CI according to the given $O(n^2k(1+k-k'))$ time complexity. The exact number of performed applications a_p is shown in column five, and the calculated ratios for the observed running times are shown in column four, respective column six.

and varied k' in steps of size two from 2 up to 20. The computation time is shown in Table 6.3.

For the analysis of the running time results, recall that the term $k(1 + k - k') =: a_e$ estimates the number of applications of Algorithm CI to a pair of strings. In the table, the third column contains this number of estimated pairwise applications for each k'. In theory, the ratio of the number of applications per time (column four) is expected to be constant, but in the table it is clearly observable, that this ratio constantly decreases for an increasing value of k'. This means, that $O(n^2k(1+k-k'))$ is a correct upper bound for the running time of the gene cluster detection with Gecko, but there might exist tighter upper bounds.

To find such an upper bound, recall from Section 3.4.9 the description how to detect common $\mathcal{CS}$-factor located in k' out of k strings. Here, the general idea was based on the application of Algorithm CI to each pair of strings (S_1, S_r), $1 \leq r \leq k$, to locate all gene clusters occurring in S_1 and at least $k' - 1$ further strings. Since a common $\mathcal{CS}$-factor does not necessarily occur in S_1, in the next step Algorithm CI was applied to the same set of strings, but without S_1 and with k' decremented by one. Here, the given term $k(1+k-k')$ counts the number of applications of Algorithm CI not exactly, since in each iteration, not only k', but also the number of strings k is reduced by one. With this refinement, the exact number a_p of performed applications of Algorithm CI can be calculated by $a_p = \sum_{k'}^{k} k'$. This number is displayed for each k' in the fifth column of Table 6.3, and the computed ratio of performed applications per time is given in the last column. As expected from theory, now this ratio is constant for all values of k'. The result is also shown graphically in Figure 6.3.

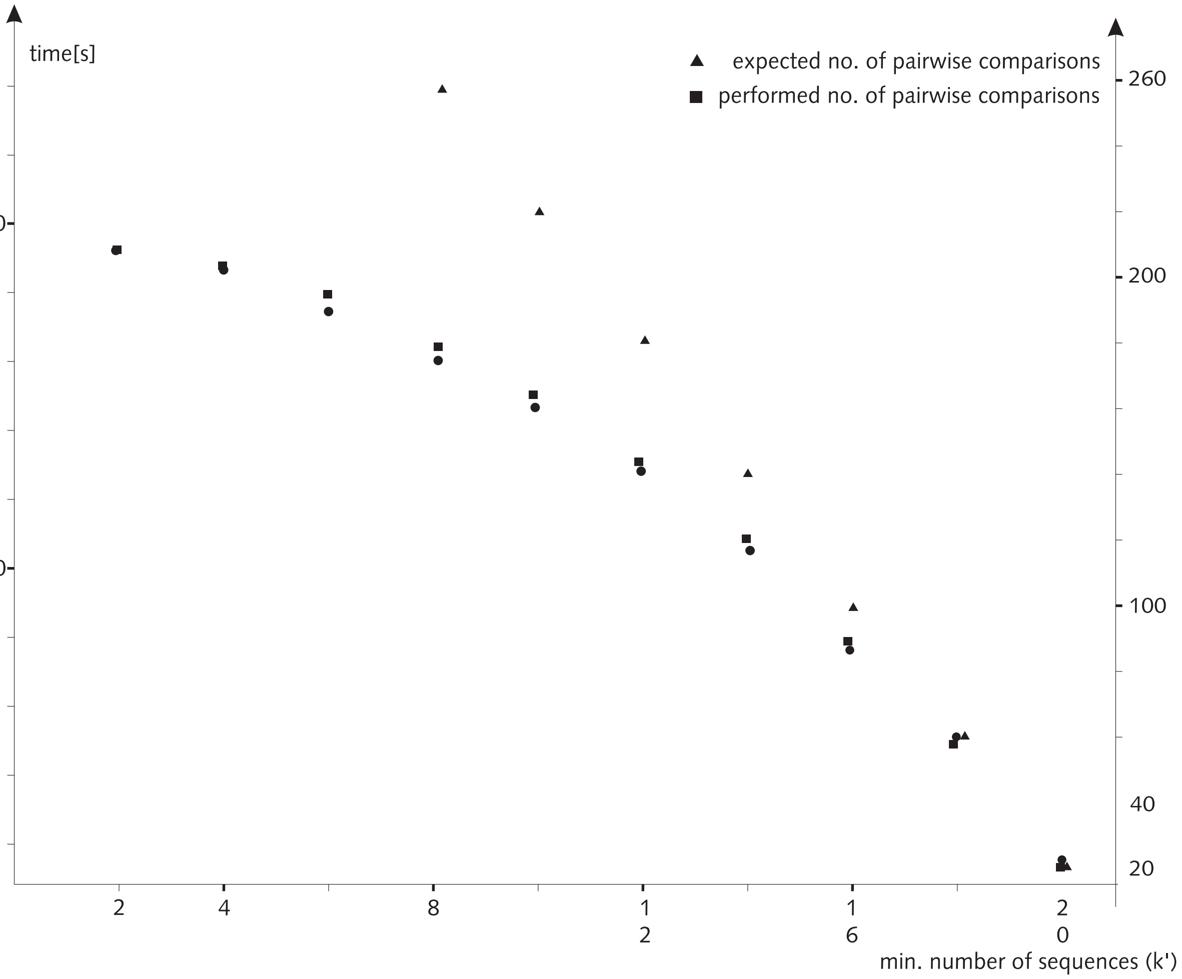


Figure 6.3: Visualization of the measured running time (dots) with different values for k', with $k = 20$ and $n = 2000$. On the right Y-axis, the estimated number of pairwise applications of Algorithm CI (triangles) is compared to the exact number of performed applications (squares).

k / *minsize* :	32 / 2	32 / 3	43 / 2	43 / 3
no paralogs	1984	792	2504	1494
inside region	243	43	365	97
duplicated region	259	115	357	256
both	248	49	560	196
total	2734	999	3786	2043

Table 6.4: Observed number of gene clusters for the different types of gene duplications.

6.2 Statistics on Real Genomic Data

For the first application of Gecko to real genomic data, we built a test set of the 43 COG-classified bacterial genomes from the NCBI database (see Section 4.1). Furthermore, we created a second test set of 32 genomes by removing the closely related genomes from the previous set. This was done to avoid misleading conclusions in the interpretation of the results, since many gene clusters in closely related species are conserved due to the lack of time for intensive genome rearrangements.

A first interesting question to answer is how many gene clusters can be found with the representation of genomes by strings instead of permutations like in previous approaches. To answer this question, we applied our gene cluster search to both sets of genomes with a fixed $k' = 2$ and without the second postprocessing step to keep the results comparable with results from other tools. For both sets, the test was run with a minimal cluster size of two to find all gene clusters of more than a single conserved homolog, and also with a minimal cluster size of three to figure out how many gene clusters consist of more than only a conserved gene pair.

For the evaluation, we partitioned the reported gene clusters according to the type of the observed gene duplication into the following four groups: no paralogs, paralogs inside a single conserved region, duplicated conserved regions, and both types in one cluster. The number of clusters for each group in the four different test cases are given in Table 6.4 and graphically displayed in Figure 6.4.

Besides the expected decrease in the total amount of detected gene clusters using the reduced genome set of only more distantly related species, it is an interesting observation that independently of the chosen genome sets and the minimal cluster size, the rate of gene clusters containing at least one type of paralogs is always found between 20% and 33%. These rates can be considered a lower bound for the number of gene clusters that cannot be (completely) detected by an approach based on the representation of genomes on permutations. However, the true number of missing or incompletely reported gene clusters can be expected to be even higher, since the presence of paralogs also complicates the detection of clusters that by themselves do not contain a paralog, but e.g. have a single duplicated gene outside the conserved region (see also Section 3.3.3).

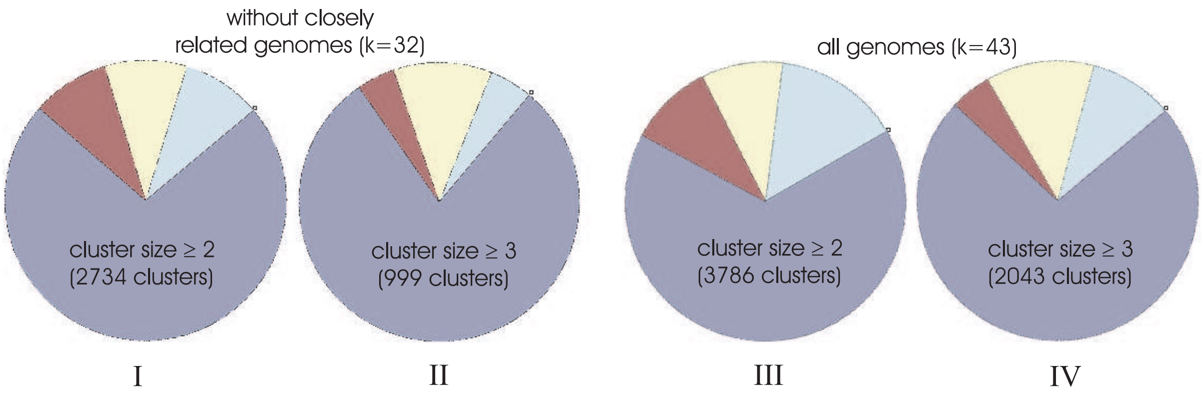


Figure 6.4: Chart representation of the observed number of gene clusters for the different types of gene duplications: no paralogs (blue), paralogs inside a single conserved region (red), duplicated conserved regions (yellow), both types (cyan). Charts I and II show the distribution of the duplication types in the reduced genome set, and Chart III and IV were computed from the results using the complete test set of 43 genomes.

Evaluation of Pattern Frequencies

The same sets of genomes were used to analyze the number of occurrences for the defined content patterns (see Section 5.3). For a compact notation, we call the gene replacement pattern *pattern1*, the cluster separation pattern *pattern2*, and the gene duplication pattern *pattern3*. Note that in this analysis we do not distinguish between the two different types of gene duplication patterns.

The gene cluster detection was performed with $k = 2$ to evaluate the pattern distribution in all p-joined cluster sets consisting of more than a single gene, and $k = 5$ to figure out how the ratios of the different types of patterns change when considering only clusters appearing conserved among a larger number of organisms. To identify the influence of the identity rate p from the second postprocessing step on the pattern distribution, each computed list of clusters was postprocessed with $p = 0.7$ and $p = 0.9$. The computed pattern frequencies are shown in Table 6.5 and visualized in Figure 6.5.

An unexpected first observation in the analysis of the pattern frequencies is the fact that the relative distribution of the content patterns for each combination of the different parameter settings on both test sets is almost identical. So far, there is no plausible reason for this observation, since even the total number of reported p-joined cluster sets varies in the wide range of 200 up to more than 2000 clusters for particular parameter settings.

On the other hand, with a larger k' and an increased identity rate p the tendency to have conserved gene clusters showing a content pattern increases in both test sets. These observations allow two different interpretations of the data. Once they indicate that especially those gene clusters which are conserved in a larger number of genomes,

k	k'	*identity rate* p	pattern 1	pattern 2	pattern 3	no pattern
43	2	0.7	233 (16%)	34 (2%)	480 (33%)	903 (62%)
43	2	0.9	360 (19%)	97 (5%)	714 (37%)	1056 (55%)
43	5	0.7	66 (21%)	17 (5%)	106 (34%)	173 (55%)
43	5	0.9	78 (22%)	25 (7%)	125 (35%)	191 (54%)
32	2	0.7	144 (14%)	16 (2%)	284 (28%)	681 (66%)
32	2	0.9	170 (14%)	36 (3%)	387 (31%)	774 (62%)
32	5	0.7	30 (15%)	8 (4%)	66 (33%)	116 (58%)
32	5	0.9	36 (17%)	10 (5%)	71 (33%)	119 (56%)

Table 6.5: Distribution of the pattern frequencies for both test sets with different values for k' and p. Note that the sum of the percent values exceed 100%, because in some gene clusters more than one content pattern was found.

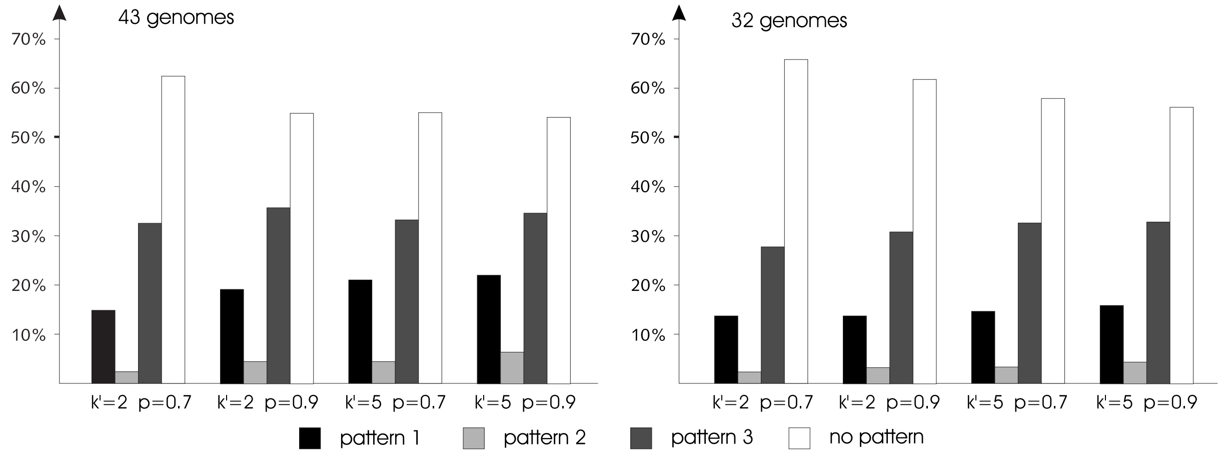


Figure 6.5: Visualization of the relative cluster distribution under the different parameter settings.

i.e. encode important functions for many species, are more often found containing a content pattern. And secondly, those gene clusters which do not contain many additional not conserved genes, are more frequently highlighted in the pattern detection phase. To this end, the detection of a content pattern points to a more important functional role of a gene cluster since in the average case those clusters are more often and more purely conserved. Therefore, the detection of a content pattern is a further valid criterion (besides the size and the number of covered genomes) for rating the significance of a gene cluster.

Furthermore, the results from the comparison of the pattern frequencies conform with the distribution of the different types of gene duplications inside a gene cluster. It is again observable that the total number of gene clusters significantly decreases whenever closely related genomes are excluded from the test set. Since in general pattern3 (the gene duplication pattern) collects all three types of gene duplications in Figure 6.4, the observation that this pattern is found in about one third of the detected clusters strongly points to the large proportion of conserved clusters that are not detectable with the representation of the genomes by permutations.

6.3 Two Real World Examples with Gecko

To demonstrate the usability of Gecko for the gene cluster detection on real data, we applied it to a test set of 20 sequences from the following genomes: *Bifidobacterium longum*, *Brevibacterium linens*, *Bacillus subtilis*, *Corynebacterium diphtheriae*, *Corynebacterium efficiens*, *Corynebacterium glutamicum*, *Escherichia coli*, *Lactococcus lactis*, *Leifsonia xyli*, *Mycobacterium avium*, *Mycobacterium leprae*, *Mycobacterium tuberculosis*, *Nocardia farcinica*, *Propionibacterium acnes*, *Pseudomonas aeruginosa*, *Streptomyces avermitilis*, *Streptomyces coelicolor*, *Thermobifida fusca*, *Tropheryma whipplei* and *Wolinella succinogenes*. The running time for the complete analysis (establishing the gene families with GhostFam and finding the clusters with Gecko) was 21 minutes on a single 900 MHz UltraSPARC-III+ CPU running the Solaris 9 operating system.

For the grouping of the genes into their families of homologs with GhostFam we used the following parameter setting: $p_0 = 50\%$, $e_0 = 10^{-5}$, $c_0 = 0.6$, $r_0 = 0.5$. This parameter setting is the default setting in GhostFam and was obtained by several test runs during the development of GhostFam. For Gecko we considered all gene clusters occurring in at least two genomes ($k' = 2$) with a minimal cluster size of two different genes. For the merging of the 2,014 detected cluster fragments we chose an identity rate of $p = 0.6$.

In the following we will discuss two specific examples. As a proof of concept, we describe the Gecko results for the well known operon for the tryptophan biosynthesis. The second example is a new gene cluster that was recently detected with the Gecko software and whose genes are involved in the assimilatory sulfate reduction.

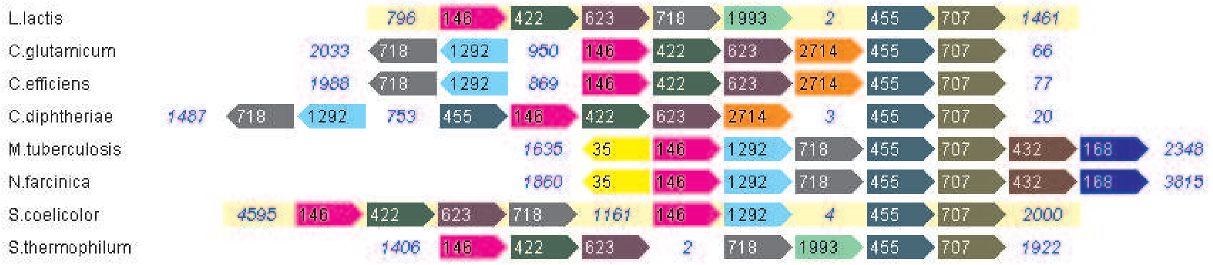

L.lactis 796 146 422 623 718 1993 2 455 707 1461
C.glutamicum 2033 718 1292 950 146 422 623 2714 455 707 66
C.efficiens 1988 718 1292 869 146 422 623 2714 455 707 77
C.diphtheriae 1487 718 1292 753 455 146 422 623 2714 3 455 707 20
M.tuberculosis 1635 35 146 1292 718 455 707 432 168 2348
N.farcinica 1860 35 146 1292 718 455 707 432 168 3815
S.coelicolor 4595 146 422 623 718 1161 146 1292 4 455 707 2000
S.thermophilum 1406 146 422 623 2 718 1993 455 707 1922

Figure 6.6: The tryptophan biosynthesis gene cluster in the *Actinobacteria C. glutamicum*, *C. efficiens*, *C. diphtheriae*, *M. tuberculosis*, *N. farcinica*, *S. coelicolor* and *S. thermophilum* as well as in the taxonomically remote eubacterium *L. lactis* as detected by Gecko. The following gene families are displayed: 35 - peroxodoxin BCPB; 146 - anthranilate synthase, aminase subunit (*trpAa*); 168 - putative membrane protein; 422 - anthranilate synthase, amidotransferase subunit (*trpAb*); 432 - putative prolipoprotein diacylglyceryl transferase; 455 - tryptophan synthase, β subunit (*trpEb*); 623 - anthranilate phosphoribosyl transferase (*trpB*); 707 - tryptophan synthase, α subunit (*trpEa*); 718 - indoleglycerol phosphate synthase (*trpD*); 1292 - putative membrane protein; 1993 - phosphoribosyl-anthranilate isomerase (*trpC*), 2714 - fusion of indoleglycerol phosphate synthase and phosphoribosyl-anthranilate isomerase (*trpDC*).

6.3.1 The Operon for Tryptophan Biosynthesis

Gene clusters, especially operons, embracing the complete gene content of a biosynthetic pathway are frequently found in bacteria. Among these, the operon for the synthesis of the amino acid tryptophan (Trp operon) has a classical status for both biochemistry and molecular genetics. Although this gene cluster can be observed as a whole-pathway operon in most bacteria sequenced so far, there are a number of instances where the Trp operon was found split into two or more parts. Therefore, it was especially interesting to investigate and reconstruct the evolutionary processes and possible lateral gene transfer events using the Trp operon as example [78, 77].

Figure 6.6 shows the visualization of the Trp operon in a number of bacteria from the *Actinobacteria*, a lineage inheriting Trp operons in different arrangements. The analysis of the gene cluster in the figure reveals that Gecko correctly shows that the Trp operon is a whole-pathway operon of

$$trpAa(146)\text{ - }trpAb(422)\text{ - }trpB(623)\text{ - }trpD(718)\text{ - }trpC(1993)\text{ - }trpEb(455)\text{ - }trpEa(707)$$

in the non-*Actinobacterium L. lactis* with an insertion of two unrelated genes between *trpC* and *trpEb*[1].

[1] The genetic nomenclature is taken from Xie *et al.* [78]

In most of the *Actinobacteria*, a *trpC*(1993) ortholog is missing and an unrelated gene (1292) is present in front of the *trpD*(718) gene. In addition, the whole-pathway Trp operon of the gene order

trpAa(146) - *trpAb*(422) - *trpB*(623) - *trpDC*(2714) - *trpEb*(455) - *trpEa*(707)

that was laterally transferred to an ancestor of all three *Corynebacterium* species is also clearly depicted. A subsequent mutational decay nowadays led to a situation where only the *trpD*(1292)-region of the original Trp operon is retained and the laterally transferred Trp operon is responsible for tryptophan biosynthesis. It is interesting to note that the *trpD* and *trpC* genes have been fused to encode one polypeptide with both protein domains in *Corynebacteria*. A presumably recent mutation event disrupted the gene order in *C. diptheriae* where three genes from an unrelated biosynthetic pathway have been inserted between *trpDC*(2714) and *trpEb*(455). All these findings were correctly revealed and depicted by Gecko.

6.3.2 A Gene Gluster for the Assimilatory Sulfate Reduction

The second example for a successful application of the gene cluster search with Gecko is a conserved set of genes participating in the assimilatory sulfate reduction in several *Actinomycetales* and *E. coli*.

As the metabolism of sulfur in *C. glutamicum* has been studied intensively by our cooperating group of Genetics at the Institute of Genome Research at Bielefeld University [37, 56, 57, 60], Gecko was used to analyze whether this cluster of genes involved in the assimilatory reduction of sulfate is retained between different bacteria. From a manual analysis of those genes it is known that a cluster of eight genes is conserved between the closely related organisms *C. glutamicum* and *C. efficiens*. For the analysis with Gecko, the genome sequences of all completely sequenced *Actinomycetales* as well as those of the two model organisms *E. coli* and *B. subtilis* were used as test set. Performing the family assignment with GhostFam, the manual editor was used for one necessary correction of the automatic family classification, since the genes encoding (P)APS reductase (*cysH*) and sulfate adenylate transferase subunit 1 (*cysD*) were wrongly clustered together due to their high sequence similarity in the parts encoding for the substrate binding domains.

Based on the family classification created with GhostFam, Gecko was used to determine if parts of the cluster are also conserved in the other genomes from the test set. The results with Gecko are shown in Figure 6.7.

Additionally, all reported parts of the gene cluster that contain the eight conserved genes from *C. efficiens* and *C. glutamicum* were manually verified to test whether the results delivered by Gecko were correct and complete. The evaluation showed that Gecko cor-

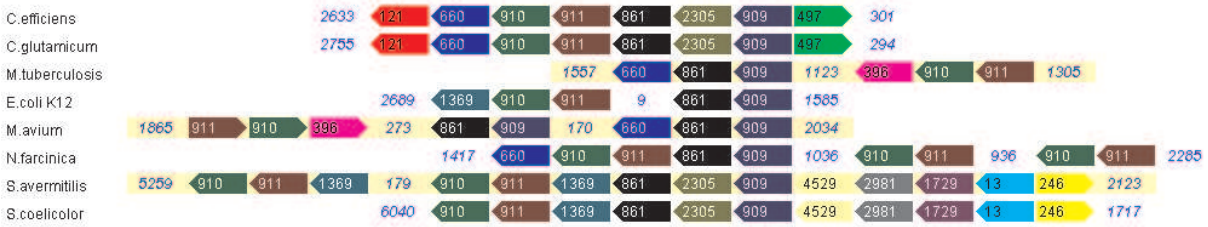

C.efficiens 2633 121 660 910 911 861 2305 909 497 301
C.glutamicum 2755 121 660 910 911 861 2305 909 497 294
M.tuberculosis 1557 660 861 909 1123 396 910 911 1305
E.coli K12 2689 1369 910 911 9 861 909 1585
M.avium 1865 911 910 396 273 861 909 170 660 861 909 2034
N.farcinica 1417 660 910 911 861 909 1036 910 911 936 910 911 2285
S.avermitilis 5259 910 911 1369 179 910 911 1369 861 2305 909 4529 2981 1729 13 246 2123
S.coelicolor 6040 910 911 1369 861 2305 909 4529 2981 1729 13 246 1717

Figure 6.7: Visualization of the search for retained parts of a gene cluster involved in the assimilatory reduction of sulfate found to be conserved between *C. glutamicum* and *C. efficiens*. All completely sequenced *Actinomycetales* and the two model organisms *E. coli* and *Bacillus subtilis* were used for the GECKO analysis. The following gene families are displayed: 13 – putative acyl-CoA dehydrogenase; 121 – predicted permease; 246 – putative integral membrane protein; 396 – conserved hypothetical protein; 497 – putative ferredoxin-NADP reductase; 660 – conserved hypothetical protein; 909 – sulfite reductase (*cysI*); 861 – phosphoadenosine phosphosulfate reductase (*cysH*); 911 – sulfate adenylyltransferase subunit 2 (*cysD*); 910 – sulfate adenylyltransferase subunit 1 (*cysN*); 1369 – adenylylsulfate kinase (*cysC*); 1729 – putative regulatory protein; 2305 – hypothetical protein; 2981 – putative secreted protein; 4529 – putative acetyltransferase (*cysE*).

rectly detects all parts of the experimentally verified cluster found in *C. glutamicum* and *C. efficiens* to be present in *N. farcinica*, *M. avium*, *M. tuberculosis*, *S. coelicolor*, *S. avermitilis*, and *E. coli*, including possible gene duplication events in *N. farcinica*, *M. avium*, and *S. avermitilis*. An additional interesting observation revealed by GECKO is the conservation of two up to now unstudied gene families (660, 2305). Both families were found as part of the gene cluster in several actinomycetal genomes. While most of the genes of the cluster, like *cysI* (909), *cysH* (861), *cysD* (911), *cysN* (910), and *cysC* (1369), have been examined quite intensively in *E. coli* [43], the function of the genes from these two gene families has yet to be determined experimentally.

Beside the correct identification of all manually verified parts of the cluster, GECKO also correctly reported this cluster to be missing in *C. diphtheriae*, *B. linens*, *L. xyli*, *M. leprae*, *P. acnes*, and *T. whipplei*. This is likely to be due to the fact that all these bacteria are either pathogens or commensals which might retrieve the needed sulfur directly from their respective hosts. A closer inspection of *B. subtilis*, where the cluster was also not found, revealed, that this organism encodes this function in a different set of genes. Since no other organism with a similar set of genes was part of the test set, the missing of this cluster in the result of GECKO was to be expected.

In the attempt to reconstruct both of the discussed gene clusters using the well known neighborhood search in STRING (see Section 3.2.1), it was not possible to reveal the whole clusters as obtained with GECKO. It was only possible to extract for each conserved operon different fragments of the complete gene cluster, depending on the chosen query gene. For example, the search for a *cysD* gene correctly discovers the complete operon structure in *S. avermitilis*, but the larger conserved region between *C. glutamicum* and *C. efficiens* also including *cysI* is only reported incompletely, while not directly enquiring for the *cysI* gene. Also the conserved parts in *N. farcinica* and *M. avium* were not reported due to the absence of those genomes from the database used by STRING.

Chapter 7

Summary and Outlook

In this thesis we developed the new formal and biologically meaningful model of a common $\mathcal{CS}$-factor to describe a gene cluster as a set of genes found in a conserved genomic neighborhood in two or more genomes. These gene clusters play an important role in the elucidation of the functional role of gene products. With the availability of all conserved gene clusters from a whole set of prokaryotic organisms, we have an additional source of information for the understanding and reconstruction of the evolutionary events resulting in the structure and organization of bacterial genomes as observable today. We presented a formal description of the problem of detecting gene clusters in a given set of genomes and reviewed prior work on different heuristic approaches for the solution of the problem. We have studied, implemented, and improved algorithms for the efficient localization of gene clusters, finally resulting in a worst case optimal quadratic time algorithm that uses linear space. Additionally, we developed a software tool for the input data preparation, the visualization and evaluation of the computed gene clusters. Finally, we tested the behavior of our developed algorithms on artificially generated as well as real biological data.

The main benefit of our new model for gene clusters accompanied by the efficient algorithms detecting them is the speed in which the results are computed and the overall high quality of the reported gene clusters. Due to its efficiency, the implementation of the gene cluster detection in Gecko is highly suitable for being used on large sets of genomes, allowing the detection of only rarely conserved clusters. Furthermore, our developed data preparation tool GhostFam provides a simple and fast way to partition genes into families of homologs. Its parameterized definition of homology allows a highly flexible clustering of the genes and provides independence of commonly used databases like the COG database. This independence is an important feature for the detection of gene clusters in newly sequenced organisms due to the fact that the family classification of their genes only becomes available from a database with a significant delay in time.

The results of the application of the Gecko tool to real genomic data clearly revealed that our formal model for gene clusters together with the presented algorithms is a fast and reliable way to extract all gene clusters from a given set of genomes. Based on these gene clusters, well founded hypotheses regarding the functional role of a single gene or a whole conserved region can be generated. These hypotheses can be used to significantly reduce the time and cost for a further experimental verification of the predicted function of a gene or the rearrangement of functionally related genes in a conserved genomic region.

Although the results reported by GECKO when applied to real data achieved a remarkable quality, there are still some unsolved problems on which further work can be expected to show promising improvements on the reported gene clusters:

- Instead of the detection of fragmented gene clusters by their grouping into p-joined cluster sets, the incorporation of missing or additional genes into the definition of common $\mathcal{CS}$-factors might be a good alternative to locate partially conserved gene clusters. Under such a model, the second postprocessing step becomes unnecessary, and furthermore, it would become possible to detect imperfectly conserved clusters in only two genomes.

- The use of information about transcriptional regulators like promoters and transcription terminators could be another valuable source of information to be incorporated into the gene cluster reconstruction. For example, the occurrence of a transcriptional terminator inside a gene cluster suggests that the genes in that region are not transcribed together, i.e. they do not form an operon.

- The analysis of the number and distribution of the content patterns in the located gene clusters has revealed that so far approximately 60% of the clusters cannot be assigned to any content pattern. This large number indicates that there might exist other, up to now not characterized content patterns, whose detection and formalization could increase the amount of information automatically inferable from the set of reported gene clusters.

- In 2002, Durand and Sankoff constructed tests to determine the significance of gene clusters against the null hypotheses of random gene order [20]. By considering the significance of individual clusters of particular genes and the overall degree of clustering in whole genomes, their approach allows a well founded and reliable estimation of the *expectation value* of a located cluster. This value together with an extended set of content patterns can be used to create a meaningful ranking of the located gene clusters, allowing to analyze the most important gene clusters at first.

- Since the number of experimentally verified gene clusters is still quite low, it is not an easy task to decide whether a detected gene cluster is a set of functionally interacting genes, or just a group of unrelated genes that is found in a conserved neighborhood due to a lack in time to diverge. Therefore, the combination of gene clusters with data from gene expression could be a promising approach to estimate the quality of a detected cluster. In this case it would be expected that if the genes inside a conserved region are functionally related, they should show a similar expression profile.

Beyond the application of GECKO for the functional analysis of genes in bacterial genomes, there are further fields in genome comparison in which GECKO can be used to retrieve valuable information:

- The use of information from conserved genomic regions to infer evolutionary relationships in a given set of genomes is not a commonly used approach so far. Usually evolutionary trees are constructed based on evolutionary distances computed from comparisons of the 16S small-subunit rRNA. In [16], Deed *et al.* showed that due to the problem of lateral gene transfer, the evolution of the 16S small-subunit rRNA does not always tells the true story about the evolution of an organism. Here, the conservation of gene clusters can be used as a further source of evidence for the verification of predicted evolutionary scenarios [5].

- A further possible application of the detection of conserved genomic neighborhoods is the search for gene clusters in viral genomes. In this approach, which is recently started in cooperation with the Bergen Center for Computational Science, GECKO is used to locate conserved clusters in a group of newly sequenced large algae infecting viruses. Here the detected gene clusters are used to support the functional annotation of the genes.

- Finally, regarding the question whether the described gene cluster detection approach is also applicable to eukaryotic genomes, there is not a simple ‘yes’ or ‘no’. In general, there is a significant difference between the evolution of higher eukaryotes and more primitive organisms including prokaryotes and yeast. Usually, eukaryotic genomes contain relatively long conserved segments, which are regions of the chromosome with identical gene content and a linear order in both. Here, the model of common $\mathcal{CS}$-factors for gene clusters does not seem to be a good choice since there are more efficient methods to detect common substrings of two strings. But also in eukaryotes there are some mechanisms that show a certain relation to the concept of gene order conservation in prokaryotes and therefore might be interesting to be analyze using the model of common $\mathcal{CS}$-factors:

  - For example *imprinted genes*, i.e. genes that are expressed from predominantly one of the parental alleles in mammalian genomes, together with their control elements often occur in conserved gene clusters [47]. The detection and analysis of these clusters, especially the gene order inside a cluster, is an important task in understanding the development of different diseases, e.g. the Prader-Willi syndrome, the Angelmann syndrome, or the Beckwith-Wiedemann syndrome.

- A further application for the detection of functionally related genes in eukaryotes is the detection of *cis*-regulatory modules (CRMs) [1]. These modules act as promoters or enhancers on the expression level of a particular gene and therefore significantly influence its transcriptional regulation. Since the localization of the CRMs turned out to be rather difficult, a comparative analysis of co-regulated genes can be used to predict putative target sites. Here the general idea is that functionally related genes, needed in exactly the same conditions, can be found co-regulated or co-expressed. Since a similar expression level can be achieved by the use of groups of similar CRMs, the upstream regions of co-expressed genes are typical target sites for a more detailed search for conserved groups of CRMs.

We hope that this work is only a first step toward the natural use of high-level genome comparison methods as a valuable source of information for the understanding and interpretation of the code of life in all species including ourselves.

List of Figures

List of Tables

Bibliography

[1] S. Aerts, P. Van Loo, G. Thijs, Y. Moreau, and B. De Moor. Computational detection of cis -regulatory modules. *Bioinformatics*, 19(90002):5ii–14, 2003.

[2] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell.* Garland Science, New York, 4th edition, 2002.

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25(17):3389–3402, 1997.

[4] A. Amir, A. Apostolico, G. M. Landau, and G. Satta. Efficient text fingerprinting via parikh mapping. *J. Discr. Alg.*, 26:1–13, 2003.

[5] S. G. E. Andersson and K. Eriksson. Dynamics of gene order structures and genomic architectures. In D. Sankoff and J. H. Nadeau, editors, *Comparative genomics*, pages 267–280. Kluwer Academic Publishers, 2000.

[6] L. Aravind, H. Watanabe, D. J. Lipman, and E. V. Koonin. Lineage-specific loss and divergence of functionally linked genes in eukaryotes. *Proc. Natl. Acad. Sci. USA*, 97(21):11319–11324, 2000.

[7] J. H. Badger and G. J. Olsen. Critica: Coding region identification tool invoking comparative analysis. *Molecular Biology and Evolution*, 16(4):512–524, 1999.

[8] R. H. Bauerle and P. Margolin. The functional organization of the tryptophan gene cluster in salmonella typhimurium. *Proc. Natl. Acad. Sci. USA*, 56:111–118, 1966.

[9] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics, LATIN 2000*, volume 1776 of *LNCS*, pages 88–94. Springer Verlag, 2000.

[10] A. Bergeron, S. Corteel, and M. Raffinot. The algorithmic of gene teams. In *Proceedings of the Second International Workshop on Algorithms in BioInformatics, WABI 2002*, pages 464–476, 2002.

[11] M. Borodovsky and J. McIninch. Genemark: parallel gene recognition for both dna strands. *Comp. Chem.*, 17(19):123–133, 1993.

[12] M. Burset and R. Guigo. Evaluation of gene structure prediction programs. *Genomics*, 34:353–367, 1996.

[13] M. Case and N. H. Giles. Evidence for nonsense mutations in the arom gene cluster of neurospora crassa. *Genetics*, 60(1):49–58, 1968.

[14] S. T. Cole, K. Eiglmeier, J. Parkhill, K. D. James, N. R. Thomson, P. R. Wheeler, N. Honore, T. Garnier, C. Churcher, D. Harris, and et al. Massive gene decay in the leprosy bacillus. *Nature*, 409(6823):1007–1011, 2001.

[15] T. Dandekar, B. Snel, M. Huynen, and P. Bork. Conservation of gene order: a fingerprint of proteins that physically interact. *Trends Biochem. Sci.*, 23:324–328, 1998.

[16] E. J. Deeds, H. Hennessey, and E. I. Shakhnovich. Prokaryotic phylogenies inferred from protein structural domains. *Genome Res.*, 15:393–402, 2005.

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

[18] D. Didier, T. Schmidt, J. Stoye, and D. Tsur. Character sets of strings. *Algorithmica*, submitted, 2004.

[19] G. Didier. Common intervals of two sequences. In *Proceedings of the Third International Workshop on Algorithms in Bioinformatics, WABI 2003*, LNBI, pages 17–24. Springer Verlag, 2003.

[20] D. Durand and D. Sankoff. Tests for gene clustering. *J. Comput. Biol.*, 10(3/4):453–482, 2002.

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

[22] W. M. Fitch. Distinguishing homologous from analogous proteins. *Trends Genet.*, 19:99–113, 1970.

[23] W. M. Fitch. Homology a personal view on some of the problems. *Trends Genet.*, 16:227–231, 2000.

[24] D. Frishman, A. Mironov, H.-W. Mewes, and M. Gelfand. Combining diverse evidence for gene recognition in completely sequenced bacterial genomes. *Nucleic Acids Res.*, 26:2941–2947, 1998.

[25] W. Fujibuchi, H. Ogata, H. Matsuda, and M. Kanehisa. Automatic detection of conserved gene clusters in multiple genomes by graph comparison and p-quasi grouping. *Nucleic Acids Res.*, 28:4029–4036, 2000.

[26] T. Gaasterland and M. A. Ragan. Microbial genescapes: phyletic and functional patterns of orf distribution among prokaryotes. *Microb Comp Genomics*, 3(4):199–217, 1998.

[27] M. Y. Galperin and E. V. Koonin. Who's your neighbor? new computational approaches for functional genomics. *Nat. Biotechnol.*, 18:609–613, 2000.

[28] F.-B. Guo, H.-Y. Ou, and C.-T. Zhang. Zcurve: a new system for recognizing protein-coding genes in bacterial and archaeal genomes. *Nucleic Acids Res.*, 31:1780–1789, 2003.

[29] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Proceedings of the 31st Annual ACM Symposium on the Theory of Computing (ACM)*, 46:1–27, 1999.

[30] X. He and M. Goldwasser. Identifying conserved gene clusters in the presence of orthologous groups. In *Proceedings of the Eighth Annual International Conference on Computational Molecular Biology, RECOMB 2004*, pages 272–280, 2004.

[31] S. Heber and J. Stoye. Algorithms for finding gene clusters. In *Proceedings of the First International Workshop on Algorithms in BioInformatics, WABI 2001*, pages 252–263, 2001.

[32] S. Heber and J. Stoye. Finding all common intervals of k permutations. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001*, pages 207–218, 2001.

[33] M. Huynen and P. Bork. Measuring genome evolution. *Proc. Natl. Acad. Sci. USA*, 95:5849–5856, 1998.

[34] M. Huynen and E. van Nimwegen. The frequency distribution of gene family size in complete genomes. *Mol. Biol. Evol.*, 15(5):583–589, 1998.

[35] I. K. Jordan, K. S. Makarova, J. L. Spouge, Y. I. Wolf, and E. V. Koonin. Lineage-specific gene expansions in bacterial and archaeal genomes. *Genome Res.*, 11(4):555–565, 2001.

[36] R. Knippers. *Molekulare Genetik*. Thieme, Stuttgart, 6th edition, 2001.

[37] D. Koch, C. Rückert, D. A. Rey, A. Pühler, and J. Kalinowski. The *ssu* and *seu* gene clusters of *Corynebacterium glutamicum* ATCC 13032 encode a system for the utilization of sulfonates and sulfonate esters as sulfur sources. Submitted for publication, 2005.

[38] G. Kolesov, H. W. Mewes, and D. Frishman. Snapper: gene order predicts gene function. *Bioinformatics*, 18(7):1017–1019, 2002.

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

[40] E. V. Koonin and M. Y. Galperin. *Sequence-Evolution-Function: Computational Approaches in Comparative Genomics.* Kluwer Academic Publishers, 1st edition, 2002.

[41] J. O. Korbel, L. J. Jensen, C. von Mering, and P. Bork. Analysis of genomic context: prediction of functional associations from conserved bidirectionally transcribed gene pairs. *Nat Biotechnol*, 22(7):911–917, 2004.

[42] J. R. Kornegay, J. W. Schilling, and A. C. Wilson. Molecular adaptation of a leaf-eating bird: stomach lysozyme of the hoatzin. *Mol. Biol. Evol.*, 11(6):921–928, 1994.

[43] N. M. Kredich. Biosynthesis of Cysteine. In F. C. Neidhardt, R. Curtis III, J. L. Ingraham, E. C. C. Lin, K. B. Low, B. Magasanik, W. S. Reznikoff, M. Riley, M. Schaechter, and H. E. Umbarger, editors, Escherichia coli *and* Salmonella*: Cellular and Molecular Biology*, volume 2, pages 514–527. ASM Press, Washington D.C., 2nd edition, 1996.

[44] W. C. Lathe III, B. Snel, and P. Bork. Gene context conservation of a higher order than operons. *Trends Biochem. Sci.*, 25:474–479, 2000.

[45] O. Lespinet, Y. I. Wolf, E. V. Koonin, and L. Aravind. The role of lineage-specific gene family expansion in the evolution of eukaryotes. *Genome Res.*, 12(7):1048–1059, 2002.

[46] H. Lodish, A. Berk, S. Zipursky, and P. Matsudaira. *Molecular Cell Biology.* W. H. Freeman, New York, 4th edition, 2000.

[47] S. Lopes, A. Lewis, P. Hajkova, W. Dean, J. Oswald, T. Forne, A. Murrell, M. Constancia, M. Bartolomei, J. Walter, and W. Reik. Epigenetic modifications in an imprinting cluster are controlled by a hierarchy of dmrs suggesting long-range chromatin interactions. *Hum. Mol. Genet.*, 12(3):295–305, 2003.

[48] M. T. Madigan, J. M. Martinko, and J. Parker. *Brock Biology of Microorganisms.* Prentice Hall, Inc., Upper Saddle River, New Jersey, 8th edition, 1997.

[49] A. R. Mushegian and E. V. Koonin. Gene order is not conserved in bacterial evolution. *Trends Genet.*, 12:289–290, 1996.

[50] H. Ogata, W. Fujibuchi, S. Goto, and M. Kanehisa. A heuristic graph comparison algorithm and its application to detect functionally related enzyme clusters. *Nucleic Acids Res.*, 28:4021–4028, 2000.

[51] R. Overbeek, M. Fonstein, M. D'Souza, G. D. Pusch, and N. Maltsev. Use of contiguity on the chromosome to predict functional coupling. *In Silico Biol.*, http://www.bioinfo.de/isb/1998/01/0009/, 1998.

[52] 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*, 96:2896–2901, 1999.

[53] M. Pellegrini, E. M. Marcotte, M. J. Thompson, D. Eisenberg, and T. O. Yeates. Assigning protein functions by comparative genome analysis: protein phylogenetic profiles. *Proc. Natl. Acad. Sci. USA*, 98(8):4285–4288, 1999.

[54] A. R. Proctor and W. E. Kloos. The tryptophan gene cluster of staphylococcus aureus. *Journ. Gen. Microbiol.*, 64:319–332, 1970.

[55] G. R. Reeck, C. de Haen, D. C. Teller, R. F. Doolittle, W. M. Fitch, R. E. Dickerson, P. Chambon, A. D. McLachlan, E. Margoliash, and T. H. Jukes. Homology in proteins and nucleic acids: a terminology muddle and a way out of it. *Cell*, 50:667, 1987.

[56] D. A. Rey, A. Pühler, and J. Kalinowski. The putative transcriptional repressor McbR, member of the TetR-family, is involved in the regulation of the metabolic network directing the synthesis of sulfur conatining amino acids in *Corynebacterium glutamicum. J. Biotechnol.*, 103(1):51–65, 2003.

[57] D. A. Rey, C. Rückert, D. J. Koch, A. Pühler, and J. Kalinowski. The McbR repressor modulated by the effector substance S-adenosylhomocysteine controls directly the transcription of a regulon involved in sulfur metabolism of *Corynebacterium glutamicum* ATCC 13032. Submitted for publication, 2005.

[58] H. W. Rines, C. M. E., and N. H. Giles. Mutants in the arom gene cluster of neurospora crassa specific for biosynthetic dehydroquinase. *Genetics*, 61(4):789–800, 1969.

[59] I. B. Rogozin, K. Makarova, J. Murvai, E. Czabarka, Y. I. Wolf, R. L. Tatusov, L. A. Szekely, and E. V. Koonin. Connected gene neighborhoods in prokaryotic genomes. *Nucleic Acids Res.*, 30:2212–2223, 2002.

[60] C. Rückert, A. Pühler, and J. Kalinowski. Genome-wide analysis of the L-methionine biosynthetic pathway in *Corynebacterium glutamicum* by targeted gene deletion and homologous complementation. *J. Biotechnol.*, 104(1-3):213–228, 2003.

[61] F. Sanger, S. Nicklen, and A. R. Coulson. Dna sequencing with chain terminating inhibitors. *Proc. Natl. Acad. Sci. USA*, 74:5463–5467, 1977.

[62] D. Sankoff, R. Cedergren, and Y. Abel. Genomic divergence through gene rearrangement. In *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, chapter 26, pages 428–438. Academic Press, Orlando, Fla., 1990.

[63] T. Schmidt, C. Rückert, J. Kalinowski, and J. Stoye. Gecko: a tool for efficient gene cluster detection in prokaryotic genomes. *Bioinformatics*, submitted, 2005.

[64] T. Schmidt and J. Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, CPM 2004*, volume 3109 of *LNCS*, pages 347–358. Springer Verlag, 2004.

[65] B. Snel, G. Lehmann, P. Bork, and M. A. Huynen. String: a web-server to retrieve and display the repeatedly occurring neighbourhood of a gene. *Nucleic Acids Res.*, 28(18):3442–3444, 2000.

[66] C.-B. Stewart, J. W. Schilling, and A. C. Wilson. Adaptive evolution in the stomach lysozymes of foregut fermenters. *Nature*, 330:401–404, 1987.

[67] K. W. Swanson, D. M. Irwin, and A. C. Wilson. Stomach lysozyme gene of the langur monkey: tests for convergence and positive selection. *J. Mol. Evol.*, 33:418–425, 1991.

[68] J. Tamames. Evolution of gene order conservation in prokaryotes. *Genome Biol.*, 2:0020.1–11, 2001.

[69] J. Tamames, G. Casari, C. Ouzounis, and A. Valencia. Conserved clusters of functionally related genes in two bacterial genomes. *J. Mol. Evol.*, 44:66–73, 1997.

[70] 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, B. S. Rao, S. Smirnov, A. V. Sverdlov, S. Vasudevan, Y. I. Wolf, J. J. Yin, and D. A. Natale. The COG database: an updated version includes eukaryotes. *BMC Bioinformatics*, 4(1):41, 2003.

[71] R. L. Tatusov, M. Y. Galperin, D. A. Natale, and K. E. V. The COG database: a tool for genome-scale analysis of protein functions and evolution. *Nucleic Acids Res.*, 28(1):33–36, 2000.

[72] R. L. Tatusov, E. V. Koonin, and D. J. Lipman. A genomic perspective on protein families. *Science*, 278(5338):631–637, 1997.

[73] R. L. Tatusov, A. R. Mushegian, P. Bork, N. P. Brown, W. S. Hayes, M. Borodovsky, K. E. Rudd, and E. V. Koonin. Metabolism and evolution of haemophilus influenzae deduced from a whole-genome comparison with escherichia coli. *Curr Biol.*, 6(3):279–291, 1996.

[74] R. L. Tatusov, D. A. Natale, I. V. Garkavtsev, T. A. Tatusova, U. T. Shankavaram, B. S. Rao, B. Kiryutin, M. Y. Galperin, N. D. Fedorova, and E. V. Koonin. The COG database: new developments in phylogenetic classification of proteins from complete genomes. *Nucleic Acids Res.*, 29(1):22–28, 2001.

[75] T. Uno and M. Yagiura. Fast algorithms to enumerate all common intervals of two permutations. *Algorithmica*, 26:290–309, 2000.

[76] J. D. Watson and F. H. C. Crick. A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[77] G. Xie, C. A. Bonner, J. Song, N. O. Keyhani, and R. A. Jensen. Intergenomic displacement via lateral gene transfer of bacterial trp operons in an overall context of vertical genealogy. *BMC Biology*, 2:15–44, 2004.

[78] G. Xie, N. O. Keyhani, C. A. Bonner, and R. A. Jensen. Ancient origin of the tryptophan operon and the dynamics of evolutionary change. *Microbiol. Mol. Biol. Rev.*, 67(3):303–342, 2003.

[79] I. Yanai and C. DeLisi. The society of genes: networks of functional links between genes from comparative genomics. *Genome Biol.*, 3:0064.1–12, 2002.

[80] Y. Zheng, R. J. Roberts, and S. Kasif. Genomic functional annotation using co-evolution profiles of gene clusters. *Genome Biology*, 3(11):0060.1–0060.9, 2002.