Sackler Faculty of Exact Sciences, School of Computer Science

Graph Modification Problems and their Applications to Genomic Research

THESIS SUBMITTED FOR THE DEGREE OF
"DOCTOR OF PHILOSOPHY"

by

Roded Sharan

The work on this thesis has been carried out
under the supervision of **Prof. Ron Shamir**

Submitted to the Senate of Tel-Aviv University
August 2002

Acknowledgments

This thesis summarizes a significant period of my life dedicated to study and research. I would not have succeeded without the unconditional love and support of my wife Michal, who shared the good and bad moments with me; nor without the encouragement of my parents who have given me their strength and wisdom. The thesis is also dedicated to my brothers, Arbel and Eytam, and my grandparents who motivated and encouraged me in every step.

I deeply thank my advisor Ron Shamir who has been my guide and mentor for these past six years, and taught me all my research skills. Ron, you have been a source of inspiration both academically and personally.

I am in great debt to all my research mates in the lab whose company made this period so much fun: Itsik Pe'er who shared with me these wonderful years right from the beginning, and whom I admire for his friendship and bright ideas; Rani Elkon who gave me so much of his knowledge in biology; Amos Tanay with whom I worked closely in the last year, a skilled programmer and a great partner for coffee; Tzvika Hartman who has a wonderful approach to science; Chaim Linhart; Dekel Tsur; and Irit Gat-Viks who was always full of knowledge and ideas. I also thank Naama Arbili, Adi Maron-Katz, Erez Hartuv and Deepak Ajwani who helped me in different stages of developing CLICK and EXPANDER and were great to work with.

I am thankful to all my collaborators: Pavol Hell who inspired me in many ways and drove me to new horizons in graph theory; Yossi Shiloh who led me in the fascinating research on Ataxia-Telangiectasia; Dan Graur; Doron Lancet; Mike Fellows; Valerie King who introduced me the magical world of dynamic graph algorithms; Amir Ben-Dor; Zohar Yakhini who taught me a lot on how to ask the right questions; Tal Pupko who shared with me his knowledge on molecular

evolution; Hans Lehrach; Michael Gurevitz; Dalia Gordon; and Haim Kaplan. I thank Ralf Herwig, Golan Yona, Antje Krause, Pablo Tamayo and Michael Eisen who helped me with their advice and data.

Finally, I would like to thank the foundations that funded my work. I deeply thank the Ministry of Science, Israel for granting me an Eshkol fellowship throughout my studies. I thank the Gutwirth foundation for supporting me at the beginning of my Ph.D.; the School of Computer Science and the Maus family for awarding me prizes on my research; and the Deutsch foundation for travel support.

Abstract

Edge modification problems call for making small changes to the edge set of an input graph in order to obtain a graph with a desired property. These problems play an important role in computer science and have applications in several fields, including molecular biology. In many application areas a graph is used to model experimental data, and then edge modifications correspond to correcting errors in the data: Adding an edge corrects a false negative error, and deleting an edge corrects a false positive error.

This thesis deals with theoretical and practical modification problems. We first study the complexity and approximability of edge modification problems on some structured classes of graphs. We show that most of the studied problems are computationally hard, but some have efficient solutions when restricting the degrees in the input graph. We then give a polynomial approximation algorithm for the classical minimum fill-in problem which has applications in numerical algebra. We provide fast algorithms for recognizing certain properties on dynamically changing graphs, with applications to physical mapping of DNA. We study a graph sandwich problem arising in phylogeny reconstruction and devise an efficient algorithm for it. Finally, we develop a new clustering algorithm which combines probabilistic and graph theoretic reasoning. The algorithm was implemented and we report on its successful application in a variety of gene expression experiments as well as other biological problems.

Contents

Chapter 1

Introduction

In this chapter we introduce graph modification problems, and provide background on previous studies on such problems. We summarize the results of the thesis, and close with preliminaries and basic definitions on graph theoretic notions.

1.1 Motivation and Background

Edge modification problems on graphs play an important role in computer science and have applications in several fields, including molecular biology. This thesis consists of two main parts: The first, theoretical part studies the complexity and approximability of edge modification problems. The second, applied part highlights the applications of such problems to genomic research.

Problem definition: Edge modification problems call for making small changes to the edge set of an input graph in order to obtain a graph with a desired property. They include completion, deletion and editing problems. Let Π be a graph property. In the Π-*Editing* problem the input is a graph $G = (V, E)$, and the goal is to find a minimum set $F \subset V \times V$ such that $G' = (V, E \triangle F)$ satisfies Π, where $E \triangle F$ denotes the symmetric difference between E and F, i.e., $E \triangle F \equiv (E \setminus F) \cup (F \setminus E)$. In the Π-*Deletion* problem only edge deletions are permitted, i.e., $F \subseteq E$. The problem is equivalent to finding a maximum subgraph of G with property Π. In the Π-*Completion* problem one is only allowed to add edges, i.e., $F \cap E = \emptyset$. Equivalently, we seek a minimum supergraph of G with property Π.

Motivation: Graph modification problems are fundamental in graph theory. Already in 1979, Garey and Johnson mentioned 18 different types of vertex and edge modification problems [69, Section A1.2]. Edge modification problems have applications in several fields, including molecular biology and numerical algebra. In many application areas a graph is used to model experimental data, and then edge modifications correspond to correcting errors in the data: Adding an edge corrects a false negative error, and deleting an edge corrects a false positive error. We summarize below some of these applications. Definitions of the graph classes are given in Section 1.3.

Interval modification problems have important applications in physical mapping of DNA (see [22, 33, 80, 84]). Since direct sequencing of large DNA molecules is currently infeasible, they are first cut into smaller fragments. In this process the order of the fragments is lost, and a major problem is to reconstruct it. One way to reconstruct the order is to test for any two fragments whether they overlap, and use this information for deducing the fragments' order. One can model the resulting problem as follows: Construct a graph G whose vertices correspond to fragments and there is an edge between two vertices if and only if their corresponding fragments overlap. Ideally, G would be an interval graph and the reconstruction problem would translate into that of finding a realization for G. However, experimental data is error-prone and, hence, G is only close to being an interval graph. Depending on the technology used and the kind of experimental errors, completion, deletion and editing problem arise, both for interval graphs and for unit interval graphs.

The chordal completion problem, also called the *minimum fill-in problem*, arises when numerically performing a Gaussian elimination on a sparse symmetric positive-definite matrix [164]. Since the time of the computation and its storage needs depend on the sparseness of the matrix, it is desirable to find an elimination order such that a minimum number of new non-zero elements is introduced into the matrix. Rose [164] showed that this problem is equivalent to the minimum fill-in problem.

The chordal deletion problem was proposed in trying to solve the CLIQUE problem. Some heuristics for finding a large clique (see, e.g., [193]) aim to find a maximum chordal subgraph of the input graph. On such subgraph a maximum clique can be found in polynomial time.

Cluster graph editing problems arise in cluster analysis (cf. [17]). When using a graph theoretic approach to clustering, one builds from the raw data a similarity

graph whose vertices correspond to elements and there is an edge between two vertices if and only if the similarity of their elements exceeds a predefined threshold (see, e.g., [96, 92]). Ideally, the resulting graph would be a union of vertex-disjoint cliques. In practice, it is only close to being such, due to data errors. The task of clustering then translates to finding an optimal editing set for this graph.

Previous results: Strong negative results are known for *vertex* deletion problems: Lewis and Yannakakis [130] showed that for any property which is non-trivial and hereditary, the maximum induced subgraph problem is NP-complete. Furthermore, Lund and Yannakakis [134] proved that for any such property, and for every $\epsilon > 0$, the maximum induced subgraph problem cannot be approximated with ratio $2^{\log^{1/2-\epsilon} n}$ in quasi-polynomial time, unless $\tilde{P} = \tilde{NP}$ (we denote throughout by n and m the number of vertices and edges in a graph, respectively).

For edge modification problems no such general results are known, although some attempts have been made to go beyond specific graph properties [10, 11, 58]. In 1979 Garey and Johnson [69] posed the complexity of Chordal Completion as a major open problem. Yannakakis subsequently proved that Chain Completion is NP-complete and reduced the latter problem to Chordal Completion, thereby proving its NP-completeness [194]. As noted in [80], the NP-completeness of Interval Completion and Unit Interval Completion also follows from [194]. The complexity of a variety of other edge modification problems was studied by many authors. Most problems were found to be NP-hard. Figure 1.1 summarizes the complexity results for some graph classes. A detailed description of those results appears in Chapter 2.

Variants of the completion problem, in which the input graph is pre-colored and the objective is to find a supergraph satisfying a specified property, such that it is properly colored by the input coloring, were also shown to be NP-complete. Goldberg et al. [80] proved that the colored unit interval completion problem is NP-complete. Golumbic et al. [84] and Fellows et al. [62, 23] proved independently that the colored interval completion problem is NP-complete. Bodlaender and de Fluiter [22] strengthened this result by showing that the latter problem is NP-complete even if the number of colors is at most 4. They also gave a quadratic algorithm (in the number of vertices) for solving the colored completion problem on 3-colored graphs. The colored chordal completion problem was proved by Bodlaender et al. [23] to be NP-complete. McMorris et al. [141] showed that this problem is polynomial when the number of colors is fixed.

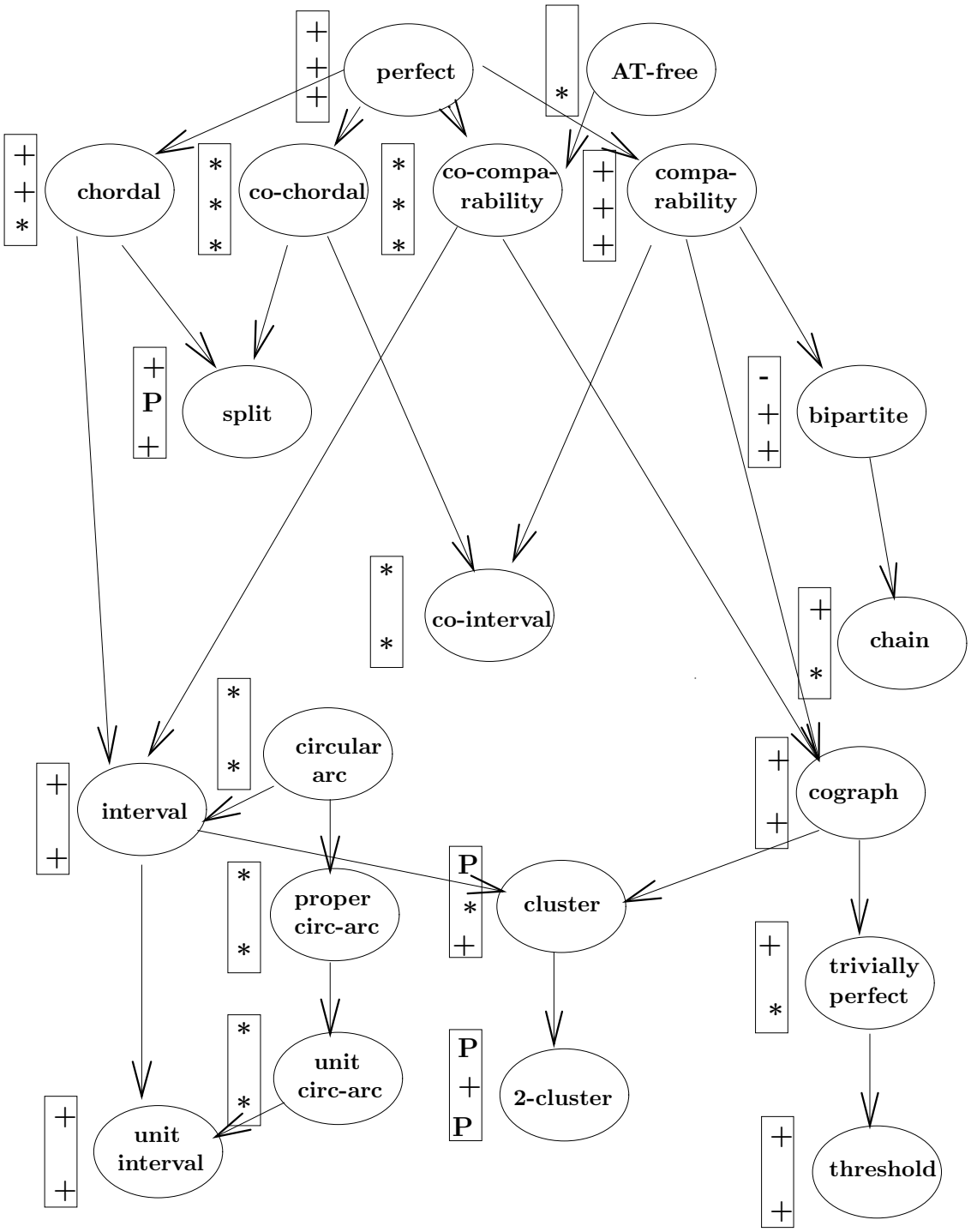


Figure 1.1: The complexity status of edge modification problems for some graph classes. A→B indicates that class A contains class B. The box to the left of each class contains the status of the completion (top), editing (middle) and deletion (bottom) problems. +: NP-hard, previously known; ∗: NP-hard, new result; P: polynomial; −: not meaningful.

A generalization of colored graph completion problems is to find a supergraph satisfying a given property, which does not include any of a predefined set of forbidden edges. Problems of this type are called *sandwich* problems. Golumbic and Shamir [86] proved that the interval sandwich problem is NP-complete. Their proof can be modified to show that the unit interval sandwich problem is also NP-complete. Golumbic et al. [85] showed that sandwich problems for chordal graphs, comparability graphs, permutation graphs, circular-arc graphs, and several other families of graphs, are NP-complete. They also proved that the sandwich problem is polynomial for split graphs, threshold graphs (this was first shown by Hammer et al. [90]) and other families of graphs.

Since most edge modification problems discussed above are NP-complete, it is natural to investigate their parametric complexity. In the parametric variant of the problems, the input contains an additional parameter k and one has to determine if an input instance can be solved using at most k edge modifications. Clearly this can be done in $n^{O(k)}$ time by enumeration. For fixed k and growing n, an algorithm with complexity $2^{O(k)}n^{O(1)}$ is superior. Parameterized complexity theory, initiated by Downey and Fellows [49], studies the complexity of such problems. It defines a hierarchy of parameterized decision problem classes, with appropriate reducibility and completeness notions (see [49] for definitions and details). Parameterized problems that have algorithms of complexity $O(f(k)n^{\alpha})$ (with α a constant) are called *fixed parameter tractable*. Thus, for example, *vertex cover* and *pathwidth* are fixed parameter tractable [21, 48, 122] but *independent set* [3] and *bandwidth* [20] are hard for certain levels in the hierarchy.

Kaplan and Shamir [117] have given a polynomial algorithm for the interval sandwich decision problem restricted to bounded degree input graphs, whenever the solution has bounded clique size or bounded degree. The results in [22] however, imply that the problem of finding an interval sandwich graph with a small clique is hard in the parametric sense, if the parameter is the size of the clique. In [118] Kaplan et al. proved that Chordal Completion and Unit-interval Completion are fixed parameter tractable, where the parameter is the number of added edges. The problem of altering a graph to one having a specified property, by deleting at most i vertices, deleting at most j edges, and adding at most k edges, where i, j, k are fixed integers, was proved by Cai [28] to be fixed parameter tractable for any hereditary property that has a finite forbidden set characterization.

Approximation algorithms exist for several edge modification problems. Agrawal et al. [5] have given an $O(m^{1/4}\log^{3.5} n)$ approximation algorithm for the minimum chordal supergraph problem (where one wishes to minimize the total number of edges in the resulting graph). For the minimum interval supergraph problem the best extant approximation algorithm by Rao and Richa [160] achieves an approximation ratio of $O(\log n)$. A general, constant factor approximation algorithm was given by Natanzon for editing and deletion problems on bounded degree graphs with respect to properties characterized by a finite set of forbidden induced subgraphs [150]. On the negative side, it was shown in [33] that the minimum number of edge editions needed in order to convert a graph into a caterpillar cannot be approximated in polynomial time to within an additive term of $O(n^{1-\epsilon})$, for $0 < \epsilon < 1$, unless P=NP. Another inapproximability result, given by Natanzon [150], proves that it is NP-hard to approximate any of the three comparability modification problems to within a factor of 18/17.

1.2 Summary of Results

In this thesis we study theoretical aspects of edge modification problems as well as specific variants of these problems arising in applications to genomic research. On the theoretical side, we give results on the complexity, parametric complexity and approximability of these problems. We also study the complexity of recognizing some graph properties on dynamically changing graphs. On the practical side we develop a clustering algorithm and apply it successfully to a variety of biological datasets. We also study a graph sandwich problem with applications in phylogeny reconstruction. The main concrete results are summarized below.

Complexity: In Chapter 2 we study the complexity of edge modification problems on some structured classes of graphs. We provide several results on the complexity and approximability of these problems. On the negative side, we show, among other results, that deletion problems are NP-hard for chain, chordal and asteroidal triple free graphs; and that Cluster Editing is NP-hard. These results are summarized in Figure 1.1. We also prove that deletion problems are NP-hard with respect to any graph class that can be characterized by a set of connected triangle-free forbidden subgraphs, the smallest of which has a tail. Examples for such graph

classes are cographs, cluster graphs, trivially perfect graphs and threshold graphs. Furthermore, we prove that it is NP-hard to approximate Cluster Deletion to within some constant factor.

On the positive side, we provide a polynomial algorithm for 2-Cluster Deletion and give polynomial results for bounded degree input graphs. Specifically, we show that Chain Deletion and Editing, Split Deletion, and Threshold Deletion and Editing are polynomial when the input degrees are bounded. We also give a 0.878-approximation algorithm for a weighted variant of 2-Cluster Editing. Most of these results were published in [150] and [172].

Minimum Fill-In Approximation: Chapter 3 deals with the minimum fill-in problem, which calls for finding a minimum triangulation of a given graph. The problem has important applications in numerical algebra and has been studied intensively since the 1970s. We give the first polynomial approximation algorithm for the problem. Our algorithm constructs a triangulation whose size is at most eight times the optimum size squared. The algorithm builds on the recent parameterized algorithm of Kaplan, Shamir and Tarjan for the same problem. For bounded degree graphs we give a polynomial approximation algorithm with a polylogarithmic approximation ratio. Furthermore, we improve the parameterized algorithm. We also derive an approximation algorithm for Chain Completion. This study was published in [149].

Dynamic Algorithms: Chapter 4 presents dynamic algorithms for recognizing certain graph properties on dynamically changing graphs. The dynamic algorithm is required to maintain a representation of a graph throughout a series of on-line modifications (insertions or deletions of a vertex or an edge), as long as the graph satisfies some property, and to detect when it ceases to satisfy the property. In the first part of the chapter we give a fully dynamic algorithm for proper interval graph recognition and representation. The algorithm handles a modification involving d edges in time $O(d + \log n)$. (In case of an edge modification $d = 1$, and in case of a vertex modification d equals its degree.) We prove a close lower bound of $\Omega(\log n/(\log\log n + \log b))$ for an edge operation in the cell probe model of computation with word-size b. In addition, we give algorithms requiring $O(d)$ time per operation for variants of the problem where either only addition operations are al-

lowed, or only deletion operations are allowed. The latter algorithms are optimal with respect to all operations, with the possible exception of vertex deletion. This study was published in [99].

The second part provides a fully dynamic algorithm for cograph recognition, which works in $O(d)$ time per operation involving d edges. The algorithm maintains a modular decomposition tree of the dynamic graph and uses it for the recognition. We derive from this result fully dynamic algorithms for threshold recognition and for trivially perfect graph recognition. These algorithms are optimal with respect to all operations, with the possible exception of vertex deletion.

Phylogeny Reconstruction: In chapter 5 we study the problem of reconstructing evolutionary history based on incomplete data. In the perfect phylogeny model for studying evolution every species has an associated vector of characters, each having one of several states. The goal is to reconstruct a tree in which the species are at the leaves and each internal node is associated with a character vector representing an ancestral species, such that the set of all species having the same state in any character induces a connected subtree.

We study the following variant of perfect phylogeny: The input is a species-characters matrix. The characters are binary and directed, i.e., a species can only gain characters. The difference from standard perfect phylogeny is that for some species the state of some characters is unknown. The question is whether one can complete the missing states in a way admitting a perfect phylogeny. The problem arises in classical phylogenetic studies, when some states are missing or undetermined. Quite recently, studies that infer phylogenies using inserted repeat elements in DNA gave rise to the same problem. Extant solutions for it take time $O(n^2m)$ for n species and m characters. We provide a formulation of the problem as a graph sandwich problem, and give a near-optimal $\tilde{O}(nm)$-time algorithm for it. We also study the problem of finding a single, general solution tree, from which any other solution can be obtained by node-splitting. We provide an algorithm to construct such a tree, or determine that none exists. These results were published in [155] and [156].

Clustering Gene Expression Data: Chapter 6 presents a novel clustering algorithm, called CLICK (CLuster Identification via Connectivity Kernels), which is

applicable to gene expression analysis as well as to other biological problems. The algorithm utilizes graph-theoretic and statistical techniques to identify tight groups (kernels) of highly similar elements, which are likely to belong to the same true cluster. Several heuristic procedures are then used to expand the kernels into the full clusters. CLICK has been implemented and we report on its successful application to a variety of biological datasets, ranging from gene expression, cDNA oligo-fingerprinting to protein sequence similarity. In all those applications it outperformed extant algorithms according to several common figures of merit. CLICK is also very fast, allowing clustering of thousands of elements in minutes, and over 100,000 elements in a couple of hours on a standard workstation. These results were published in [175] and [171].

One application of CLICK on which we report in detail is a study of expression data related to the Ataxia-Telangiectasia degenerative disease, done in collaboration with Prof. Y. Shiloh (Tel-Aviv University) and QBI Enterprises [161]. A-T is a complex multisystem disease resulting from deficiency of the ATM protein kinase. Most notably, A-T cells exhibit profound defects in their responses to ionizing radiation. A-T patients show progressive degeneration of the cerebellum and thymus. In this study, gene expression profiles were constructed for the cerebellum, thymus, and cerebrum of ATM- knockout mice and of wild-type animals, with and without prior X-irradiation. The resulting gene expression patterns were clustered using CLICK. Marked differences were observed in the post- irradiation response between the three tissues and the two genotypes. Unexpectedly, ATM-deficient thymus and cerebellum from unirradiated animals displayed constitutive activation or repression of numerous genes that the corresponding wild-type tissues showed only after irradiation. This constitutive response to sustained internal genotoxic stress, which correlates with tissue degeneration in human A-T patients, points to an important new characteristic of A-T.

We also show the utility of CLICK in extracting other biological information from gene expression data: We apply CLICK successfully for the identification of common regulatory motifs in the upstream regions of co-regulated genes. Furthermore, we demonstrate how CLICK can be used to accurately classify tissue samples into disease types, based on their expression profiles, achieving success ratios of over 90% on two real datasets. These results were published in [173].

Finally, we present a new java-based graphical tool, called EXPANDER (EXPres-

sion ANalyzer and DisplayER), for gene expression analysis and visualization [174]. This software provides graphical user interface to several clustering methods including CLICK, K-Means, hierarchical clustering and self organizing map. It enables visualizing the raw expression data and the clustered data in several ways. The EXPANDER tool is used in several dozens of laboratories world-wide.

Another application of CLICK in a large scale project of sequencing a superfamily of genes is reported in [67].

1.3 Preliminaries

In this thesis we focus on graph modification problem with respect to subclasses of perfect graphs and other structured classes. Below we provide basic terminology and definitions that will be used throughout the thesis. Section 1.3.1 gives basic graph theoretic definitions and Section 1.3.2 defines these graph classes. For additional definitions of graph properties and much more on the graph classes discussed here see, e.g., [25, 82].

1.3.1 Definitions

All graphs in this thesis are simple and contain no self-loops. Let $G = (V, E)$ be a graph. We denote its set of vertices also by $V(G)$, and its set of edges also by $E(G)$. Throughout we use n and m to denote the number of vertices and edges, respectively, in a graph. A weighted graph $G = (V, E, w)$ is a graph whose edges are assigned real weights according to a function $w : E \to \mathcal{R}$.

For a new vertex $z \notin V$ and a set of edges E_z between z and vertices of V, we denote by $G \cup z$ the graph $(V \cup \{z\}, E \cup E_z)$ obtained by adding z to G. For a vertex $z \in V$ we denote by $G \setminus z$ the graph $(V \setminus \{z\}, E \setminus (\{z\} \times V))$ obtained by removing z from G.

For a set S we use $S \otimes S$ to denote $\{(s_1, s_2) : s_1, s_2 \in S, s_1 \neq s_2\}$. We say that $(S_1, \ldots, S_l)$ is a *partition* of S if the subsets $S_1, \ldots S_l$ are pairwise disjoint, and their union is S. We denote by $\overline{G}$ the *complement graph* of G, i.e., $\overline{G} = (V, \overline{E})$, where $\overline{E} = (V \otimes V) \setminus E$. If $G = (U, V, E)$ is a bipartite graph, then its *bipartite complement* is the bipartite graph $\overline{G} = (U, V, \overline{E})$, where $\overline{E} = (U \times V) \setminus E$. For a subset $A \subseteq V$

we denote by G_A the subgraph induced by the vertices of A. For a vertex $v \in V$ we denote by $N(v)$ the set of vertices adjacent to v in G. $N(v)$ is called the *open neighborhood* of v. We let $N[v] = N(v) \cup \{v\}$ denote the *closed neighborhood* of v. For a set $S \subseteq V$ we define $N(S) = \cup_{v \in S} N(v)$ and $N[S] = N(S) \cup S$. We denote by $G \cup H$ the union of two disjoint graphs G and H (with no edges connecting a vertex of G with a vertex of H). We denote by $G + H$ the graph obtained by forming the union of two disjoint graphs G and H and connecting every vertex of G to every vertex of H.

A *cut* C in G is a subset of its edges, whose removal disconnects G. The *weight* of C is the sum of weights of its edges. A *minimum weight cut* is a cut of minimum weight in G. In case of positive edge weights, a minimum weight cut C partitions the vertices of G into two disjoint non-empty subsets $A, B \subset V$, $A \cup B = V$, such that $E \cap \{(u, v) : u \in A, v \in B\} = C$.

A path with l edges is called an *l-path* and its *length* is l. A single vertex is considered a 0-path. We denote an $(l-1)$-path by P_l. The *distance* between two vertices $a, b \in V$ is the length of the shortest path connecting a and b in G. The *diameter* of G is the maximum distance between a pair of vertices in G. We call a cycle with l edges an *l-cycle*, and denote it by C_l. A *chord* in a cycle is an edge between non-consecutive vertices on it. A *chordless cycle* is a cycle of length greater than three that contains no chord. A *triangle* is a cycle of length 3. We call a graph *triangle-free* if it contains no triangles. We say that a graph has a *tail* if it contains a pair of adjacent vertices, one of degree two and the other of degree one.

Let Π be a graph property. The notation $G \in \Pi$ indicates that G satisfies Π. If F is a set of non-edges such that $G' = (V, E \cup F) \in \Pi$ and $|F| \leq k$, then F is called a *k-completion set* with respect to Π, or a Π *k-completion set*. Π *k-deletion set* and Π *k-editing set* are similarly defined.

1.3.2 Graph Classes

A graph G is called *perfect* if for every induced subgraph H of G, $\chi(H) = \omega(H)$, where $\chi(H)$ denotes the chromatic number of H, and $\omega(H)$ denotes the clique number of H.

A graph is called *chordal*, or *triangulated*, if it contains no chordless cycle.

A *comparability* graph is a graph whose edges can be transitively oriented, that is, there exists an orientation F of its edges for which $(a,b),(b,c) \in F$ implies $(a,c) \in F$.

A graph G is called an *interval graph* if its vertices can be assigned to intervals on the real line so that two vertices are adjacent in G if and only if their assigned intervals intersect. The set of intervals assigned to the vertices of G is called a *realization* of G. If the set of intervals can be chosen to be inclusion-free, then G is called a *proper interval graph*, or a *unit interval graph*.

A graph is called a *circular-arc graph* if its vertices can be assigned to arcs on a circle so that two vertices are adjacent if and only if their corresponding arcs intersect.

A graph G is called a *cluster graph* if every connected component of G is a complete graph. G is called a *2-cluster graph* if it is a cluster graph with two connected components or, equivalently, if it is a vertex-disjoint union of two cliques.

A *split graph* is a graph whose vertices can be partitioned into two subsets, such that one subset induces a clique, and the other induces an independent set.

A bipartite graph $G = (P, Q, E)$ is called a *chain graph* if there exists an ordering π of P, $\pi : P \to \{1, \ldots, |P|\}$, such that $N(\pi^{-1}(1)) \subseteq N(\pi^{-1}(2)) \subseteq \ldots \subseteq N(\pi^{-1}(|P|))$.

A graph $G = (V, E)$ is called a *threshold graph*, if there is a partition (K, I) of V such that K induces a clique, I induces an independent set, and the bipartite graph $(K, I, E \cap (K \times I))$ is a chain graph (see [136] for other equivalent definitions of this class).

An *asteroidal triple* is a set of three independent (i.e., pairwise non-adjacent) vertices such that there is a path between every two of them which avoids the closed neighborhood of the third vertex. A graph is called *asteroidal triple free*, or *AT-free*, if it contains no asteroidal triple.

A graph is called a *cograph* (*complement reducible graph*) if it contains no induced P_4. A graph is called *trivially perfect* if is a cograph and contains no induced C_4.

A *claw* is an induced $K_{1,3}$ (a 3-degree vertex connected to three 1-degree vertices). A graph is called *claw-free* if it contains no induced claw.

Chapter 2

Complexity Analysis

In this chapter we study the complexity of edge modification problems on some structured classes of graphs. We provide several results on the complexity and approximability of these problems. On the negative side, we show, among other results, that deletion problems are NP-hard for chain, chordal and asteroidal triple free graphs; and that Cluster Editing is NP-hard. We also prove that deletion problems are NP-hard with respect to any graph class that can be characterized by a set of connected triangle-free forbidden induced subgraphs, the smallest of which has a tail. Examples for such graph classes are cographs, cluster graphs, trivially perfect graphs and threshold graphs. Furthermore, we show that it is NP-hard to approximate Cluster Deletion to within some constant factor.

On the positive side, we provide a polynomial algorithm for 2-Cluster Deletion and give polynomial results for bounded degree input graphs. Specifically, we show that Chain Deletion and Editing, Split Deletion, and Threshold Deletion and Editing are polynomial when the input degrees are bounded. We also give a 0.878-approximation algorithm for a weighted variant of 2-Cluster Editing.

Most of the results in this chapter were published in [150] and [172].

2.1 Introduction

Edge modification problems call for making small changes to the edge set of an input graph in order to obtain a graph with a desired property. They include

completion, deletion and editing problems. These problems play an important role in computer science and have applications in several fields, including molecular biology. In many application areas a graph is used to model experimental data, and then edge modifications correspond to correcting errors in the data: Adding an edge corrects a false negative error, and deleting an edge corrects a false positive error. Specific applications that are discussed in this thesis include numerical algebra (Chapter 3), physical mapping of DNA (Chapter 4), phylogeny reconstruction (Chapter 5) and clustering (Chapter 6).

Since the classical result of Yannakakis, that the minimum fill-in problem is NP-complete [194], many other complexity results were obtained for edge modification problems. Some of these results are summarized in Table 2.1 (compare also Figure 1.1).

Graph class	Completion	Editing	Deletion
Perfect	NP-hard [150]	NP-hard [150]	NP-hard [150]
Chordal	NPC [194]	NPC [14]	NPC new
Interval	NPC [194, 69, 119]	-	NPC [80]
Unit Interval	NPC [194]	-	NPC [80]
Circular-Arc	NPC new	-	NPC new
Chain	NPC [194]	-	NPC new
Comparability	NPC [89]	NPC [150]	NPC [195]
AT-Free	-	-	NPC new
Cograph	NPC [58]	-	NPC [58]
Threshold	NPC [138]	-	NPC [138]
Bipartite	NPC [70]	NPC [70]	Not meaningful
Split	NPC [150]	P [91]	NPC [150]
Cluster	P	NPC new	NPC [58]
2-Cluster	P [172]	NPC [172]	P new
Caterpillar	-	NPC [33]	-
Trivially Perfect	NPC [194]	-	NPC new

Table 2.1: Summary of complexity results for some edge modification problems. 'new' indicates results obtained here. '-' indicates an open problem.

Approximation algorithms exist for several problems. Agrawal et al. [5] have

given an $O(m^{1/4}\log^{3.5} n)$ approximation algorithm for the minimum chordal supergraph problem (where one wishes to minimize the total number of edges in the resulting graph). Rao and Richa [160] have given an $O(\log n)$ approximation algorithm for the minimum interval supergraph problem. A general, constant factor approximation algorithm was given by Natanzon for editing and deletion problems on bounded degree graphs with respect to properties characterized by a finite set of forbidden induced subgraphs [150]. On the negative side, it was shown in [33] that the minimum number of edge editions needed in order to convert a graph into a caterpillar cannot be approximated in polynomial time to within an additive term of $O(n^{1-\epsilon})$, for $0 < \epsilon < 1$, unless P=NP. Also, Natanzon has proven that it is NP-hard to approximate any of the three comparability modification problems to within a factor of 18/17 [150].

Here we give several results on the complexity and approximability of edge modification problems. Most of our polynomial and NP-completeness results for specific graph classes are summarized in Table 2.1. We also prove that deletion problems are NP-hard with respect to any graph class that can be characterized by a set of connected triangle-free forbidden subgraphs, the smallest of which has a tail. This applies to complement reducible, cluster, trivially perfect and threshold graphs. Furthermore, we show that it is NP-hard to approximate Cluster Deletion to within some constant factor. We also show that Chain Deletion and Editing, Split Deletion, and Threshold Deletion and Editing are polynomial when the input degrees are bounded. Finally, we give a 0.878-approximation algorithm for a weighted variant of 2-Cluster Editing.

The chapter is organized as follows: Section 2.2 contains simple basic results that show connections between the complexity of related modification problems. Section 2.3 contains the main hardness results. Section 2.4 gives the polynomial results. Finally, Sections 2.5 and 2.6 describe the approximation algorithm and the inapproximability results.

2.2 Basic Results

In this section we summarize some easy observations on edge modification problems, which will help us deduce complexity results from results on related graph families, and concentrate on those modification problems that are meaningful.

A graph property Π is called *hereditary* if when a graph G satisfies Π every induced subgraph of G satisfies Π. Π is called *hereditary on subgraphs* if when G satisfies Π, every subgraph of G satisfies Π. Π is called *ancestral* if when G satisfies Π, every supergraph of G satisfies Π.

Proposition 2.2.1 *If property Π is hereditary on subgraphs then Π-Deletion and Π-Editing are polynomially equivalent, and Π-Completion is not meaningful.*

A problem is *not meaningful* if it is trivial on every instance. For example, since the planarity property is hereditary on subgraphs, Planarity Completion is meaningless: For every graph either it is planar or it cannot be made planar by adding edges.

Proposition 2.2.2 *If Π is an ancestral graph property then Π-Completion and Π-Editing are polynomially equivalent, and Π-Deletion is not meaningful.*

Proposition 2.2.3 *If Π and Π' are graph properties such that for every graph G and a disjoint independent set S, G satisfies Π if and only if $G \cup S$ satisfies Π', then Π-Deletion is polynomially reducible to Π'-Deletion. If in addition Π is hereditary, then Π-Completion (Π-Editing) is polynomially reducible to Π'-Completion (Π'-Editing).*

Proof: The first part of the proposition is obvious. To prove the second part we show a reduction from Π-Completion to Π'-Completion. The reduction from Π-Editing to Π'-Editing is identical. Let $< G = (V, E), k >$ be an instance of Π-Completion. We build an instance $< G' = (V', E), k >$ of Π'-Completion by adding $2k + 1$ isolated vertices to G.

We now prove validity of the reduction. If F is a Π k-completion set for G then it is also a Π' k-completion set for G', since the modified graph $(V', E \cup F)$ is a union of a graph which satisfies Π and an independent set. On the other hand, suppose that F is a Π' k-completion set for G'. Then $(V', E \cup F)$ contains an isolated vertex, and removing that vertex results in a graph satisfying Π. Since Π is hereditary, $F \cap (V \otimes V)$ is a Π k-completion set for G. ■

Corollary 2.2.4 *The following problems are NP-complete: (1) Circular-Arc Completion and Deletion; (2) Proper Circular-Arc Completion and Deletion; (3) Unit Circular-Arc Completion and Deletion.*

Proof: Obviously, for a graph G and an isolated vertex $z \notin V(G)$, G is an interval (unit interval) graph if and only if $G \cup z$ is a circular-arc (proper circular-arc and unit circular-arc) graph. The corollary now follows by reduction from the corresponding interval or unit interval modification problem. ■

Proposition 2.2.5 *If* Π *and* Π' *are graph properties such that for every graph* G *and a clique* K*,* G *satisfies* Π *if and only if* $G + K$ *satisfies* Π'*, then* Π*-Completion is polynomially reducible to* Π'*-Completion. If in addition* Π *is hereditary, then* Π*-Deletion (*Π*-Editing) is polynomially reducible to* Π'*-Deletion (*Π'*-Editing).*

Corollary 2.2.6 *Permutation modification problems are polynomially reducible to the corresponding circle modification problems.*

For a graph property Π, we define the *complementary property* $\overline{\Pi}$ as follows: For every graph G, G satisfies $\overline{\Pi}$ if and only if $\overline{G}$ satisfies Π. Some well known examples are co-chordality and co-comparability.

Proposition 2.2.7 *For every graph property* Π*,* Π*-Deletion and* $\overline{\Pi}$*-Completion are polynomially equivalent.*

Proposition 2.2.8 *For every graph property* Π*,* Π*-Editing and* $\overline{\Pi}$*-Editing are polynomially equivalent.*

Corollary 2.2.9 *The following problems are NP-complete: (1) Co-Chordal Deletion and Editing; (2) Co-Comparability modification problems; (3) Co-Interval Completion and Deletion.*

2.3 NP-Hard Modification Problems

2.3.1 Chain Graphs

In this section we prove that Chain Deletion is NP-complete. This result will be the starting point to several of our subsequent reductions. Note, that in Chain Deletion (as in Chain Completion [194]) the bipartition of the input graph is given as part of the input.

Lemma 2.3.1 *The bipartite complement of a chain graph is a chain graph.*

Proof: The claim follows from the observation that the chain containment order is reversed for the bipartite complement of a chain graph. Formally, let $G = (P, Q, E)$ be a chain graph, and let π be an ordering of the vertices in P such that $N(\pi(1)) \subseteq N(\pi(2)) \subseteq \ldots \subseteq N(\pi(|P|))$. Then for $\overline{G}$ we have $N(\pi(|P|)) \subseteq N(\pi(|P|-1)) \subseteq \ldots \subseteq N(\pi(1))$. ■

Corollary 2.3.2 *Chain Deletion is NP-complete.*

Proof: Follows from the bipartite analog of Proposition 2.2.7. ■

2.3.2 Chordal Graphs

In this section we prove that Chordal Deletion is NP-complete by reduction from Chain Deletion. We use the following characterization of chain graphs, due to Yannakakis [194]: A bipartite graph $G = (P, Q, E)$ is a chain graph if and only if it contains no pair of *independent edges*, i.e., a pair $(p_1, q_1), (p_2, q_2) \in E$ such that $(p_1, q_2), (p_2, q_1) \notin E$.

Theorem 2.3.3 *Chordal Deletion is NP-complete.*

Proof: The problem is in NP since chordal graphs can be recognized in linear time [182]. We prove NP-hardness by reduction from Chain Deletion. Let $< G = (P, Q, E), k >$ be an instance of Chain Deletion. Build the following instance $< C(G) = (V', E'), k >$ of Chordal Deletion: Let V_P and V_Q be two sets of new vertices of size k each. Define

$$V' = P \cup Q \cup V_P \cup V_Q,$$

$$E' = E \cup (P \otimes P) \cup (Q \otimes Q) \cup (P \times V_P) \cup (Q \times V_Q).$$

We show that the Chordal Deletion instance has a solution if and only if the Chain Deletion instance has a solution.

$\Rightarrow$ Suppose that F is a chain k-deletion set. We claim that F is also a chordal k-deletion set. Let $H = (V', E' \setminus F)$. Suppose to the contrary that H is not chordal, and let C be an induced cycle of length greater than 3 in H. If C contains any vertex $v \in V_P$ then the two neighbors of v on C are vertices from P, a contradiction. The same holds for V_Q. Hence, $V(C) \cap V_P = V(C) \cap V_Q = \emptyset$. Since P and Q induce cliques in H, C must be of the form (p_1, p_2, q_1, q_2), where $p_1, p_2 \in P$ and $q_1, q_2 \in Q$. But then (p_1, q_2) and (p_2, q_1) are independent edges in the chain graph $(P, Q, E \setminus F)$, a contradiction.

$\Leftarrow$ Suppose that F is a chordal k-deletion set. We shall prove that $F \cap E$ is a chain k-deletion set. Let $G' = (P, Q, E \setminus F)$. If G' is not a chain graph then it contains a pair of independent edges $(p_1, q_1), (p_2, q_2)$, where $p_1, p_2 \in P$ and $q_1, q_2 \in Q$. In $C(G)$, p_1, p_2 and also q_1, q_2 were connected by an edge and k edge-disjoint paths of length 2. Hence, each pair is still connected by a path of length at most 2 in $H = (V', E' \setminus F)$. Thus, p_1, q_1, q_2 and p_2 are on an induced cycle of length at least 4 in H, a contradiction. ■

Corollary 2.3.4 *Co-Chordal Completion is NP-complete.*

We note, that similar constructions provide simple proofs for the NP-completeness of Interval Deletion and Unit-Interval Deletion.

2.3.3 AT-Free Graphs

Theorem 2.3.5 *AT-free Deletion is NP-complete.*

Proof: The problem is clearly in NP. The hardness proof is by reduction from Chain Deletion. Let $< G = (P, Q, E), k >$ be an instance of Chain Deletion. Build the following instance $< A(G) = (V', E'), k >$ of AT-free Deletion: Let V_q, V_w, V_z be sets of new vertices of sizes $k, k+1, k+1$, respectively. Define

$$V' = P \cup Q \cup V_q \cup V_w \cup V_z \ ,$$

$$E' = E \cup (P \otimes P) \cup (P \times V_q) \cup (P \times V_w) \cup ((V_w \cup V_z) \otimes (V_w \cup V_z)) \ .$$

We now prove validity of the reduction.

$\Rightarrow$ Let F be a chain k-deletion set. We claim that F is also an AT-free k-deletion set. Let $G' = (P, Q, E \setminus F)$ and let $A(G)' = (V', E' \setminus F)$. Suppose to the contrary that $S = \{x, y, z\}$ is an asteroidal triple in $A(G)'$. We observe the following:

- P and $V_w \cup V_z$ induce cliques in $A(G)'$. Therefore, S contains at most one vertex from P and at most one vertex from $V_w \cup V_z$.
- For any two vertices $x, y \in V_q$, $N(x) = N(y)$. Therefore, S contains at most one vertex from V_q.
- Since G' is a chain graph (and the chain containment property holds for both sides of the bipartition [194]), for every $x, y \in Q$, $N(x) \subseteq N(y)$ or $N(y) \subseteq N(x)$. Therefore, S contains at most one vertex from Q.
- If S contains a vertex from Q then $S \cap (P \cup V_q \cup V_w) = \emptyset$, since every path from a vertex in Q to a vertex in $V' \setminus Q$ intersects the closed neighborhood of every vertex in $(P \cup V_q \cup V_w)$.
- If S contains a vertex $u \in V_w$ then S cannot contain a vertex $v \in V_q$ since $N(v) \subseteq N(u)$.
- If S contains a vertex $v \in V_q \cup V_w$ then $N(v) \supseteq P$, so $S \cap P = \emptyset$.

These observations imply that $S \cap P = \emptyset$, since otherwise S could not contain any vertex from Q or from $V_q \cup V_w$, and would have therefore at most two vertices (one from P and one from V_z), a contradiction. Similarly, we conclude that $S \cap Q = \emptyset$. It follows that $|S| \leq 2$ since S may only contain one vertex from V_q and one vertex from $V_w \cup V_z$, a contradiction.

$\Leftarrow$ Let F be an AT-free k-deletion set. We show that $F \cap E$ is a chain k-deletion set. Let $G' = (P, Q, E \setminus F)$ and let $A(G)' = (V', E' \setminus F)$. Suppose to the contrary that G' is not a chain graph. Thus, G' contains two independent edges $(p_1, q_1), (p_2, q_2)$ where $p_1, p_2 \in P$ and $q_1, q_2 \in Q$. We shall identify a vertex $z \in V_z$ such that $\{q_1, q_2, z\}$ is an asteroidal triple in $A(G)'$.

Every vertex of P was adjacent in $A(G)'$ to all $k+1$ vertices of V_w. Hence, there exist $w_1, w_2 \in V_w$, $w_1 \neq w_2$, such that $(p_1, w_1) \in E' \setminus F$ and $(p_2, w_2) \in E' \setminus F$. Similarly, there exists a vertex $z \in V_z$ such that $(w_1, z), (w_2, z) \in E' \setminus F$. $\{q_1, q_2, z\}$ is an asteroidal triple since:

1. (z, w_1, p_1, q_1) is a path from z to q_1 avoiding the neighborhood of q_2.

2. (z, w_2, p_2, q_2) is a path from z to q_2 avoiding the neighborhood of q_1.

3. If $(p_1, p_2) \in E' \setminus F$ then (q_1, p_1, p_2, q_2) is a path from q_1 to q_2 avoiding the neighborhood of z. Otherwise, there exists a vertex $q \in V_q$ such that $(p_1, q), (p_2, q) \in E' \setminus F$. Thus, (q_1, p_1, q, p_2, q_2) is a path from q_1 to q_2 avoiding the neighborhood of z.

Hence, we arrive at a contradiction, implying that G' is a chain graph. ■

2.3.4 Cluster Graphs

Let $G = (V, E)$ be a graph, and let F be a cluster editing set for G. Let $G' = (V, E \triangle F)$. We denote by $P(F)$ the partition of V into disjoint subsets of vertices according to the connected components (cliques) of G'. For a partition $P = (V_1, \ldots, V_l)$ of V we denote by N_P the size of the cluster editing set implied by P:

$$N_P \equiv |\bigcup_{i=1}^{l} \{(u, v) \notin E : u, v \in V_i\}| + |\{(u, v) \in E : u \in V_i, v \in V_j, i \neq j\}| \,.$$

For two subsets of vertices $A, B \subseteq V$ we denote by $E_{A,B}$ the set of edges in E with one endpoint in A and the other in B.

We prove in this section that Cluster Editing is NP-complete by reduction from a restriction of exact cover by 3-sets which we define next:

Problem 1 (3-Exact 3-Cover (3X3C))
Instance: *A collection C of triplets of elements from a set $U = \{u_1, \ldots, u_{3n}\}$, such that each element of U is a member of at most 3 triplets.*
Question: *Is there a sub-collection $I \subseteq C$ of size n which covers U?*

The 3X3C problem is known to be NP-complete [69, Problem SP2].

Theorem 2.3.6 *Cluster Editing is NP-complete.*

Proof: Membership in NP is trivial. We prove NP-hardness by reduction from 3X3C. Let $m \equiv 30n$. Given an instance $< C, U >$ of 3X3C we build a graph

$G = (V, E)$ as follows:

$$\begin{aligned} V &= \bigcup_{S \in C} \{v_1(S), \ldots, v_m(S)\} \cup U \ , \\ E &= E_1 \cup E_2 \cup E_3 \ , \\ E_1 &= \{(v_i(S), u) : S \in C, 1 \leq i \leq m, u \in S\} \ , \\ E_2 &= \{(v_i(S), v_j(S)) : S \in C, 1 \leq i < j \leq m\} \ , \\ E_3 &= \{(u, u') : \exists S \in C \text{ s.t. } u, u' \in S\} \ . \end{aligned}$$

In words, we build a clique of size $m + 3$ around each triplet S by fully connecting S and m additional vertices. For each triplet $S \in C$ we denote $V_S = \{v_1(S), \ldots, v_m(S)\}$. The elements of V_S are called *S-vertices*. Let $q = 3|C|$. Define $N \equiv m(q - 3n)$ and $M \equiv |E_3| - 3n$. We prove that there is an exact cover of U if and only if there is a cluster editing set for G of size at most $N + M$:

$\Rightarrow$ Suppose that $I \subseteq C$ is an exact cover of U. Let $F_1 = \{(v_i(S), u) : S \notin I, 1 \leq i \leq m, u \in S\}$ and let $F_2 = \{(u, u') \in E_3 : \not\exists S \in I \text{ s.t. } u, u' \in S\}$. It is easy to verify that $F = F_1 \cup F_2$ is a cluster editing set for G, whose size is $|F| = |F_1| + |F_2| = N + M$.

$\Leftarrow$ Let F' be a cluster editing set for G with $|F'| \leq N + M$. Let F be an optimum cluster editing set for G. Then $|F| \leq |F'| \leq N + M$. We shall prove that $|F| = N + M$ and one can derive from F an exact cover of U. This implies that $|F'| = |F|$ and, hence, F' is an optimum cluster editing set from which an exact cover of U can be obtained.

Since each element of U occurs in at most 3 triplets, $q \leq 9n$. Thus, $|E_3| \leq q \leq 9n$ and $|F| \leq N + M \leq 6mn + 6n = 180n^2 + 6n < \frac{m}{2}(\frac{m}{2} - 2)$. Let $G' = (V, E \triangle F)$ be the cluster graph obtained by editing G according to F. We shall prove that for every subset $S \in C$ there exists a unique clique in G' which contains V_S. To this end, we first show that there exists a clique K_S in G' such that $|K_S \cap V_S| \geq m/2 + 3$: Suppose that the vertices of V_S are partitioned among k cliques $X_1, \ldots, X_k$ in G'. Let $s(X_i) = |V_S \cap X_i|, i = 1, \ldots, k$. Suppose to the contrary that $s(X_i) \leq m/2 + 2$ for all i. Therefore,

$$|F| \geq \frac{1}{2} \sum_{i=1}^{k} s(X_i)(m - s(X_i)) \geq \frac{1}{2} \sum_{i=1}^{k} s(X_i)(\frac{m}{2} - 2) = \frac{m}{2}(\frac{m}{2} - 2) \ .$$

A contradiction follows.

Let K_S be the clique X_i for which $s(X_i)$ is maximum ($|K_S \cap V_S| \geq m/2+3$). We next prove that $V_S \subseteq K_S \subseteq V_S \cup S$. Let $x = |K_S \setminus (V_S \cup S)|$. Consider a new partition P' of V, which is obtained from $P(F)$ by splitting K_S into $K_S \cap (V_S \cup S)$ and $K_S \setminus (V_S \cup S)$. Clearly, $N_{P(F)} - N_{P'} \geq (m/2+3)x - 3x = xm/2$. But F is an optimum cluster editing set. Therefore, $x = 0$ and $K_S \subseteq V_S \cup S$. To see that $K_S \supseteq V_S$, suppose to the contrary that there exists some index $1 \leq i \leq m$ such that $v_i(S) \notin K_S$. Let K' be the clique in G' which contains $v_i(S)$. Let P'' be a new partition of V, which is obtained from $P(F)$ by moving $v_i(S)$ from K' to K_S. Then $N_{P(F)} - N_{P''} \geq m/2 + 3 - (m/2 - 4 + 3) = 4$, a contradiction. We conclude that for every $S \in C$ there is a unique clique K_S in G' which contains V_S and is contained in $V_S \cup S$.

Examine an element $u \in U$ which is a member of (at least) two subsets $S_1, S_2 \in C$. By the previous claim, V_{S_1} and V_{S_2} are subsets of distinct cliques in G'. Hence, either $E_{V_{S_1},\{u\}} \subseteq F$ or $E_{V_{S_2},\{u\}} \subseteq F$ (or both). Let $F_1 = F \cap E_1$. Then $|F_1| \geq N$, with equality if and only if each vertex $u \in U$ is adjacent in G' to the S-vertices of exactly one subset S and $u \in S$. Moreover, since $|F| \leq N + M$ and $M \leq 6n < m$, each vertex $u \in U$ must be adjacent in G' to all the S-vertices of exactly one subset S, where $u \in S$. This follows since u must be adjacent to at least one S-vertex, and all the S-vertices are members of the same clique K_S in G'. Call this set the *S-set* of u.

Let $F_2 = F \setminus F_1$. For every two vertices $u, u' \in U$ such that $(u, u') \in E$, and the S-sets of u and u' differ, we must have $(u, u') \in F_2$. Since each subset in C contains 3 elements, G'_U is a union of cliques of size at most 3. It is easy to verify that the maximum number of edges in such a $3n$-vertex graph is $3n$, and that number is obtained if and only if G'_U is a union of triangles only. Therefore, $|F_2| = |E_3| - |E(G'_U)| \geq M$ with equality if and only if there is a partition of U into triplets of elements, such that the elements of each triplet have the same S-set. Since $|F| \leq N + M$, we must have $|F| = N + M$ and the implied partition into triplets induces an exact cover of U. ■

We note, that the same construction can be used to show that Cluster Deletion is NP-complete. Cluster Completion is trivially polynomial, as the optimum solution is formed by transforming each connected component of the input graph into a complete graph.

2.3.5 A General NP-Hardness Result

We say that a graph has a *tail* if it contains a pair of adjacent vertices, one of degree two and the other of degree one. In this section we prove that deletion problems are NP-hard with respect to any property that can be characterized by any set of connected triangle-free forbidden induced subgraphs, one of the smallest of which (in terms of the number of vertices) has a tail. We call the set of such properties $\mathcal{Q}$. Examples for graphs with property $q \in \mathcal{Q}$ include cluster graphs, cographs, threshold graphs and trivially perfect graphs.

Theorem 2.3.7 *The* Π*-Deletion problem is NP-hard with respect to any property in* $\mathcal{Q}$*.*

Proof: By reduction from 3X3C, similar to that in Theorem 2.3.6. We use the same notation and constants as in the proof of Theorem 2.3.6. Let Π be a graph property in $\mathcal{Q}$ and let H be a copy of a smallest forbidden subgraph for Π which has a tail. Let $V(H) = \{a_1, \ldots, a_h\}$, where a_1, a_2 form a tail of H, the degree of a_1 is one, and a_3 is the other neighbor of a_2. Given an instance $< C, U >$ of 3X3C we build a graph $G = (V, E)$ as follows:

$$\begin{aligned} V &= \bigcup_{S \in C} \{v_1(S), \ldots, v_m(S)\} \cup U \cup V_4 \ , \\ E &= E_1 \cup E_2 \cup E_3 \cup E_4 \ , \\ E_1 &= \{(v_i(S), u) : S \in C, 1 \leq i \leq m, u \in S\} \ , \\ E_2 &= \{(v_i(S), v_j(S)) : S \in C, 1 \leq i < j \leq m\} \ , \\ E_3 &= \{(u, u') : \exists S \in C \text{ s.t. } u, u' \in S\} \ . \end{aligned}$$

The vertex set V_4 comprises $h-4$ subsets $A_4(S), \ldots, A_h(S)$ of m^2 vertices, for each set $S \in C$. We let $A_3(S) \equiv V_S = \{v_1(S), \ldots, v_m(S)\}$. The edge set E_4 comprises the following edges: (1) For every $a, b \in A_i(S)$, $a \neq b$ we have $(a, b) \in E_4$ for $4 \leq i \leq h$,$S \in C$; and (2) for every $a \in A_i(S), b \in A_j(S)$ we have $(a, b) \in E_4$ if $(a_i, a_j) \in E(H)$, $i, j \geq 3$,$S \in C$. In words, for every triple S we form a clique around S, add a clique $A_3(S)$ of size m and $h-4$ additional cliques of size m^2, and fully connect every pair of cliques whose corresponding vertices in H are connected. We also fully connect S and $A_3(S)$. We prove that there is an exact cover of U if and only if there is a Π deletion set for G of size at most $N + M$:

$\Rightarrow$ Suppose that $I \subseteq C$ is an exact cover of U. Let $F_1 = \{(v_i(S), u) : S \notin I, 1 \leq i \leq m, u \in S\}$ and let $F_2 = \{(u, u') \in E_3 : \nexists S \in I \text{ s.t. } u, u' \in S\}$. Let $G' = (V, E \setminus F)$, where $F = F_1 \cup F_2$. It is easy to verify that $|F| = |F_1| + |F_2| = N + M$. Moreover, any triangle-free connected induced subgraph J of G' must have all its vertices in some $S \cup A_3(S) \cup \ldots \cup A_h(S)$ for the same S due to the connectivity requirement. Hence, either $|J| = 2$ or J can have at most one member in each of $S, A_3(S), \ldots, A_h(S)$ and at most $h-1$ members in total. It follows that no triangle-free connected induced subgraph of G' is isomorphic to a forbidden subgraph of Π, so F is a Π deletion set for G as required.

$\Leftarrow$ Let F be a Π deletion set of size at most $N + M$. Let $G' = (V, E \setminus F)$. As shown in the proof of Theorem 2.3.6, $N + M < m^2$. We first claim that for every $S \in C$ and for every $a \in A_3(S)$ there exist vertices $a_i(S) \in A_i(S)$, $i = 4 \ldots h$ such that the subgraph $H^a(S)$ of G' induced by these vertices and a is isomorphic to $H \setminus \{a_1, a_2\}$. For proof, consider first the original graph G. The subgraph induced on $A_4(S) \cup \ldots \cup A_h(S)$ contains m^2 vertex-disjoint copies of $H \setminus \{a_1, a_2, a_3\}$ (with each $a_i \in H$ matching some vertex in $A_i(S)$). Hence, the subgraph induced on $\{a\} \cup A_4(S) \cup \ldots \cup A_h(S)$ contains at least m^2 *edge*-disjoint copies of $H \setminus \{a_1, a_2\}$. Since $|F| < m^2$, at least one of these copies remains intact in G'. This completes the proof of the claim.

Suppose now that $u \in U$ is connected in G' to the S-vertices of two subsets. Specifically, suppose u is connected to some $a \in V_{S_1}$ and $b \in V_{S_2}$ for $S_1 \neq S_2$. Then $H^a(S_1)$,u and b constitute a subgraph isomorphic to H, with b and u forming the tail, a contradiction. We conclude that every $u \in U$ is connected in G' to the S-vertices of at most one subset S. This implies that at least $N = m(q - 3n)$ edges must have been deleted between U and S-vertices. Furthermore, since $|F| \leq N + M \leq N + 6n$, we conclude that each $u \in U$ is adjacent to some S-vertices of exactly one subset S.

Similarly, if $u \in U$ is adjacent to vertices of S and $u' \in U$ is adjacent to vertices of $S' \neq S$ in G', then $(u, u') \notin E(G')$. Using the same arguments as in the proof of Theorem 2.3.6 we conclude that $|F| \geq N + M$ with equality if and only if there is an exact cover of U. ■

Corollary 2.3.8 *Trivially Perfect Deletion is NP-complete.*

We note that the NP-completeness of Trivially Perfect Completion follows from the reduction of Yannakakis from Chain Completion to Chordal Completion [194].

2.4 Polynomial Algorithms

2.4.1 2-Cluster Deletion

We give in this section a linear-time algorithm for 2-Cluster Deletion. Let $G = (V, E)$ be an input graph. Without loss of generality, G is connected as, otherwise, either G is already a 2-cluster graph or we output *False*. The algorithm is summarized in Figure 2.1.

Let $\overline{G}$ be the complement graph of G having t connected components.
For every component C_i of $\overline{G}$ **do:**
    If C_i is not bipartite **then** output *False* and halt.
    Else find a bipartition (A_i, B_i) of C_i such that $|A_i| \geq |B_i|$.
Output the deletion set that corresponds to $(A_1 \cup \ldots \cup A_t, B_1 \cup \ldots \cup B_t)$.

Figure 2.1: An algorithm for 2-Cluster Deletion.

Theorem 2.4.1 *The algorithm solves 2-Cluster Deletion in $O(n + |E(\overline{G})|)$ time.*

Proof: **Correctness**: Since the complement of a 2-cluster graph is a complete bipartite graph, a solution exists if and only if $\overline{G}$ is bipartite. Hence, the algorithm outputs *False* if and only if no solution exists. Moreover, the partition produced by the algorithm has the property that if two vertices are assigned to the same set then they are adjacent. Therefore, the set of edges F returned by the algorithm is a 2-Cluster deletion set of G. It suffices to prove that F is optimum.

Denote $S_1 = A_1 \cup \ldots \cup A_t$ and $S_2 = B_1 \cup \ldots \cup B_t$. By the algorithm, F is the set of edges in G with one endpoint in S_1 and the other in S_2. Therefore,

$$|F| = |E_{S_1,S_2}| = |S_1||S_2| - E(\overline{G}) = |S_1|(n - |S_1|) - E(\overline{G}).$$

Let F^* be an optimum 2-deletion set of G, and let $P(F^*) = (S_1^*, S_2^*)$, where $|S_1^*| \leq |S_2^*|$. Then $|F^*| = |S_1^*|(n - |S_1^*|) - E(\overline{G})$. For every $i \leq t$, either $A_i \subseteq S_1^*$ or $B_i \subseteq S_1^*$

and, therefore, $|S_1| \leq |S_1^*| \leq n/2$. It follows that $|F| \leq |F^*|$, so F is an optimum 2-deletion set of G.

Complexity: The bottleneck in the complexity of the algorithm is computing the connected components of $\overline{G}$ and finding a bipartition for each of them. Both these operations can be performed in $O(n + |E(\overline{G})|)$ time. ■

2.4.2 Bounded Degree Graphs

In this section we give polynomial algorithms for Chain Deletion and Editing, Split Deletion, and Threshold Deletion and Editing, when restricted to bounded degree graphs. These results are derived by observing that for these properties the search space becomes bounded when the problem is restricted to bounded degree graphs.

For the results concerning editing problems we need the following lemma:

Lemma 2.4.2 ([148]) *Let Π be a hereditary graph property such that if $G = (V, E)$ satisfies Π then $G' = (V, E \setminus E_{\{v\},N(v)})$ satisfies Π for every $v \in V$ (i.e., the property remains satisfied if we remove all the edges incident on a vertex v). Then an optimum solution of Π-Editing on a d-degree bounded graph produces a graph with degree bounded by $2d$.*

Proof: The lemma follows by noting that it is never beneficial to add more than d edges incident on the same vertex, since one could instead make that vertex isolated by modifying fewer edges. ■

Proposition 2.4.3 *Chain Deletion and Chain Editing can be solved in polynomial time on bounded degree graphs.*

Proof: Let G be an input d-degree bounded graph. Observe that a chain graph with degree bounded by d has at most $2d$ vertices with degree at least one. Hence, a maximum chain subgraph of G has at most $2d$ vertices with degree at least one. This set of vertices can be found by complete enumeration in polynomial time. Similarly, by Lemma 2.4.2 an optimum solution to the editing problem produces a $2d$-degree bounded graph, which has at most $4d$ vertices with degree at least one. Hence, we

can find this set of vertices (and derive the optimum editing set) in polynomial time. ■

Theorem 2.4.4 *Split Deletion can be solved in polynomial time on bounded degree graphs.*

Proof: Observe that a d-degree bounded split graph has a maximum clique of size at most $d+1$. Hence, one can enumerate all possible partitions of the vertex set of the graph into a clique and an independent set in polynomial time. ■

Theorem 2.4.5 *Threshold Deletion and Threshold Editing can be solved in polynomial time on bounded degree graphs.*

Proof: Let $G = (V, E)$ be an input d-degree bounded graph. An optimum threshold deletion set produces a graph with degree bounded by d. By Lemma 2.4.2, an optimum threshold editing set produces a graph with degree bounded by $2d$. Hence, one can enumerate all partitions of V into a clique of size at most $d+1$ (or $2d+1$) and an independent set in polynomial time, and for each bipartition solve a chain modification problem on the corresponding bipartite graph using the result of Proposition 2.4.3. ■

Note the differences between the definition of chain modification problems, in which the bipartition is part of the input, vs. threshold and split modification problems, in which the partition is unknown. We followed here the footsteps of Yannakakis, who defined Chain Completion in the bipartite setting [194]. If the bipartition is known, then split modification problems become trivial or meaningless: The clique side should be made full, and the independent set side should be made edge-less. Similarly, in threshold modification problems, the two sides must be transformed into a clique and an independent set, and the remaining problem is precisely chain modification.

2.5 Approximating 2-Cluster Editing

In this section we study the problem of transforming a graph into a 2-cluster graph such that the total weight of unedited vertex pairs is maximized. We call this

problem *Weighted 2-Cluster Editing*. Its NP-completeness follows from the NP-completeness of 2-Cluster Editing. We give a 0.878-approximation algorithm for this problem.

Let $G = (V, E, w)$ be an input weighted graph. We define the following semi-definite relaxation of Weighted 2-Cluster Editing:

$$\begin{aligned} \max \quad & \frac{1}{2}[\sum_{(i,j)\in E}(w_{ij}(1 + v_i \cdot v_j)) + \sum_{(i,j)\notin E}(w_{ij}(1 - v_i \cdot v_j))] \\ \text{s.t.} \quad & v_i \in S_n \quad \forall i \end{aligned}$$

We claim that this is indeed a relaxation of Weighted 2-Cluster Editing, that is, for every 2-partition $P = (A, B)$ of G there exist vectors $v_1, \ldots, v_n \in S_n$ such that the total weight of unedited vertex pairs as implied by P is $\frac{1}{2}[\sum_{(i,j)\in E}(w_{ij}(1 + v_i \cdot v_j)) + \sum_{(i,j)\notin E}(w_{ij}(1 - v_i \cdot v_j))]$. Indeed, let (A, B) be a 2-partition of G. Let v_0 be any unit vector in S_n. For every $i \in A$ set $v_i = v_0$, and for every $i \in B$ set $v_i = -v_0$. The claim follows.

Our approximation algorithm solves this semi-definite relaxation and then rounds the solution obtained by choosing a random hyperplane with normal z, and assigning vertex i to A if and only if $v_i \cdot z > 0$.

Theorem 2.5.1 *The algorithm approximates Weighted 2-Cluster Editing with an expected approximation ratio of* 0.878.

Proof: Follows directly from [79, Theorem 6.1]. ■

2.6 Inapproximability Results

In this section we prove that it is NP-hard to approximate Cluster Deletion to within some constant factor. The proof is via a gap preserving reduction from a restricted version of SET-COVER which is defined next:

Problem 2 (Minimum Restricted Exact Cover (REC))
Instance*: A set of elements $U = \{u_1, \ldots, u_t\}$, and a collection C of subsets of U which satisfies the following conditions:*

- $\bigcup_{S \in C} S = U$.
- *There exists a constant* $k_1 > 0$ *such that for each* $S \in C$, $|S| \leq k_1$.
- *There exists a constant* $k_2 > 0$ *such that for all* $u \in U$, $|\{S \in C : u \in S\}| \leq k_2$.
- *If* $S \in C$ *and* $S' \subset S$ *then* $S' \in C$.

Goal*: Find a sub-collection* $I \subseteq C$ *of minimum cardinality, such that* $\bigcup_{S \in I} S = U$, *and the sets in* I *are pairwise-disjoint.*

Note that the first and last conditions guarantee that a solution to REC always exists.

Lemma 2.6.1 *REC is MAX-SNP complete.*

Proof: By a simple L-reduction from a restriction of SET-COVER in which the size of every set is bounded and each element occurs in a bounded number of sets. This latter problem is known to be MAX-SNP complete [154]. ■

Corollary 2.6.2 *There exists some constant* $\delta_{REC} > 0$ *such that it is NP-hard to approximate REC to within a factor of* $1 + \delta_{REC}$.

A gap preserving reduction is defined as follows (cf. [105]): Let Π and Π' be two minimization problems. A *gap preserving reduction* from Π to Π' with parameters $(c, \rho), (c', \rho')$ is a polynomial time algorithm f. For each instance I of Π, algorithm f produces an instance $I' = f(I)$ of Π'. The optima of I and I', denoted by $opt(I)$ and $opt(I')$ respectively, satisfy the following properties:

$$opt(I) \leq c \quad \Rightarrow \quad opt(I') \leq c' \tag{2.1}$$

$$opt(I) > \rho c \quad \Rightarrow \quad opt(I') > \rho' c' \tag{2.2}$$

Here c, ρ are functions of $|I|$, c', ρ' are functions of $|I'|$, and $\rho, \rho' \geq 1$.

A gap preserving reduction can be used to prove inapproximability results as follows (cf. [105]): Suppose that it is NP-hard to approximate Π to within a factor of ρ. Then the reduction shows that it is NP-hard to approximate Π' to within a factor of ρ'.

Theorem 2.6.3 *There exists some constant $\epsilon > 0$ such that it is NP-hard to approximate Cluster Deletion to within a factor of $1+\epsilon$.*

Proof: By a gap preserving reduction from REC with the parameters $(c, 1+\delta_{REC})$,$(c', 1+\epsilon)$: Let $I_{REC} =< U, C >$ be an instance of REC, and let $|U| = t$. Suppose that each set in C has size at most k_1, and each element occurs in at most k_2 sets. Let $m = \frac{k_1^2 k_2}{\delta_{REC}}$ and let $q = \sum_{S \in C} |S|$. We build an instance $I_{CD} =< G = (V, E) >$ of Cluster Deletion as follows:

$$\begin{aligned}
V &= \bigcup_{S \in C} \{v_1(S), \ldots, v_m(S), w(S)\} \cup U \ , \\
E &= E_1 \cup E_2 \cup E_3 \cup E_4 \ , \\
E_1 &= \{(v_i(S), u) : S \in C, 1 \leq i \leq m, u \in S\} \ , \\
E_2 &= \{(v_i(S), v_j(S)) : S \in C, 1 \leq i < j \leq m\} \ , \\
E_3 &= \{(u, u') : \exists S \in C \text{ s.t. } u, u' \in S\} \ , \\
E_4 &= \{(v_i(S), w(S)) : S \in C, 1 \leq i \leq m\} \ .
\end{aligned}$$

In words, for each $S \in C$ we form a clique on S and a set of m new vertices $V_S = \{v_1(S), \ldots, v_m(S)\}$, and also connect all the new vertices to a single extra vertex $w(S)$. Note that $|E_3| \leq (k_1 - 1)k_2 t/2 < k_1 k_2 t/2$ and $q \leq k_2 t$. Clearly, $t/k_1 \leq opt(I_{REC}) \leq t$. Let c be any constant such that $t/k_1 \leq c \leq t$. Define $c' \equiv (q - t + c)m + |E_3|$ and $\epsilon \equiv \frac{\delta_{REC}}{2k_1 k_2 + \delta_{REC}}$. We prove that this reduction is gap preserving:

1. Suppose that $opt(I_{REC}) \leq c$. Let $I \subseteq C$ be an exact cover of U, $|I| \leq c$. Let $\bar{I} = C \setminus I$. For $u \in U$ denote by I_u the set in I which contains u.

   To obtain a cluster subgraph G' of G we delete the following edges:

   (a) For all $S \in \bar{I}, u \in S$ delete all the edges in $E_{V_S, \{u\}}$.

   (b) For all $S \in I$ delete all the edges in $E_{V_S, \{w(S)\}}$.

   (c) For all $u \in U, u' \in U \setminus I_u$ delete the edge (u, u') if it exists.

   Clearly, G' is a cluster graph and, therefore, $opt(I_{CD}) \leq (q - t + c)m + |E_3| = c'$.

2. Suppose that $opt(I_{REC}) > (1+\delta_{REC})c$. We can make the following observations with respect to $opt(I_{CD})$:

(a) In any cluster subgraph of G, every $u \in U$ is adjacent to vertices in V_S for at most one set $S \in C$. Therefore, $opt(I_{CD}) \geq (q-t)m$.

(b) There exists an optimum solution F of I_{CD} for which: If a vertex $u \in U$ is adjacent to a vertex of V_S in $(V, E \setminus F)$, for some $S \in C$, then F contains all the edges in $E_{V_S,\{w(S)\}}$. Indeed, if F' is a cluster deletion set such that $u_1, \ldots, u_r$ $(1 \leq r \leq k_1)$ are adjacent to a vertex of V_S in $(V, E \setminus F')$, then $F'' = (F' \cup E_{V_S,\{w(S)\}}) \setminus (\bigcup_{i=1}^{r} E_{V_S,\{u_i\}} \cup \{v_i(S), v_j(S) : i \neq j\})$ is also such a cluster deletion set, and $|F''| \leq |F'|$. Examine now F. For each $u \in U$, either $E_{V \setminus U,\{u\}} \subseteq F$ or there exists a single set $S \in C$ such that $E_{V_S,\{u\}} \not\subseteq F$ and $E_{V_S,\{w(S)\}} \subseteq F$. Let k be the number of vertices $u \in U$ for which the latter case applies, and let $\mathcal{T}$ be the collection of all sets S such that $(v_i(S), u) \in E \setminus F$ for some $u \in U, i$. It follows that $|F| \geq (q-k+|\mathcal{T}|)m$. The sets in $\mathcal{T}$ cover k elements of U, so $|\mathcal{T}| \geq opt(I_{REC}) - (t-k)$. Thus, we have $opt(I_{CD}) \geq (q - t + opt(I_{REC}))m > (q - t + (1+\delta_{REC})c)m$.

We conclude that

$$\begin{aligned} opt(I_{CD}) &> (q - t + (1+\delta_{REC})c)m = c' + (\delta_{REC}cm - |E_3|) \\ &> c'(1 + \frac{\delta_{REC}cm - |E_3|}{qm + |E_3|}) > c'(1 + \frac{\delta_{REC}(t/k_1)m - k_1k_2t/2}{k_2tm + k_1k_2t/2}) \\ &= c'(1 + \frac{2\delta_{REC}m/k_1 - k_1k_2}{2k_2m + k_1k_2}) = c'(1 + \frac{\delta_{REC}}{2k_1k_2 + \delta_{REC}}) \\ &= c'(1+\epsilon)\ . \end{aligned}$$

■

Chapter 3

Approximating the Minimum Fill-In

In this chapter we study the minimum fill-in problem, which calls for finding a minimum triangulation of a given graph. The problem has important applications in numerical algebra and has been studied intensively since the 1970s. We give the first polynomial approximation algorithm for the problem. Our algorithm constructs a triangulation whose size is at most eight times the optimum size squared. The algorithm builds on the recent parameterized algorithm of Kaplan, Shamir and Tarjan for the same problem. For bounded degree graphs we give a polynomial approximation algorithm with a polylogarithmic approximation ratio. Furthermore, we improve the parameterized algorithm. We also derive an approximation algorithm for Chain Completion.

This study was published in [149].

3.1 Introduction

For a non-chordal graph G, a chordal completion set F is called a *fill-in* or a *triangulation* of G. If $|F| \leq k$ then F is called a *k-triangulation* of G. We denote by $\Phi(G)$ the size of the smallest fill-in of G.

In the *minimum fill-in* problem one has to find a minimum triangulation of a given graph. The importance of the problem stems from its applications to numerical

algebra. In many fields, including VLSI simulation, solution of linear programs, signal processing and others (cf. [55]), one has to perform a Gaussian elimination on a sparse symmetric positive-definite matrix. During the elimination process zero entries may become non-zeroes. Different elimination orders may introduce different sets of new non-zero elements into the matrix. The time of the computation and its storage needs depend on the sparseness of the matrix. It is therefore desirable to find an elimination order such that a minimum number of zero entries is filled-in with non-zeroes (even temporarily). Rose [164] proved that the problem of finding an elimination order for a symmetric positive-definite matrix M, such that fewest new non-zero elements are introduced, is equivalent to the minimum fill-in problem on a graph whose vertices correspond to the rows of M and in which (i, j) is an edge if and only if $M_{i,j} \neq 0$.

Due to its importance the problem has been studied intensively [27, 73, 74, 169], and many heuristics have been developed for it [43, 75, 153, 164]. None of those gives a performance guarantee with respect to the size of the fill-in introduced. Note that in contrast, the *minimal* fill-in problem (finding a a triangulation of G which is minimal with respect to inclusion) is polynomial [152].

Approximation attempts succeeded only for the related *minimum triangulated supergraph* problem (MTS). In MTS the goal is to add edges to the input graph in order to obtain a chordal graph with minimum *total* number of edges. While as optimization problems MTS and minimum fill-in are equivalent, they may differ drastically as approximation problems. For example, if the input graph has $\Omega(n^2)$ edges and fill-in of size $o(n^2)$ then one can trivially achieve a constant approximation ratio for MTS by making the graph an n-clique (a complete graph), while no such approximation guarantee exists for the minimum fill-in problem. The approximation results regarding MTS use the nested dissection heuristic, which was first proposed by George [72] (see [74] for details). Gilbert [78] showed that for a graph with maximum degree d there exists a balanced separator decomposition such that a nested dissection ordering based on that decomposition yields a chordal supergraph, in which the number of edges is within a factor of $O(d \log n)$ of optimal. The result was not constructive as one has yet to find such a decomposition. Leighton and Rao [127] gave a polynomial approximation algorithm for finding a balanced separator in a graph of size within a factor of $O(\log n)$ of optimal. Agrawal, Klein and Ravi [5], using Gilbert's ideas and the result of [127], obtained a polynomial approximation algorithm with ratio $O(\sqrt{d} \log^4 n)$ for MTS on graphs with maximum

degree d. They also gave a polynomial approximation algorithm for MTS on general graphs, which generates for an input graph G a chordal supergraph with total number of edges $O((m + \Phi(G))^{3/4}\sqrt{m}\log^{3.5} n)$.

In the *parametric* fill-in problem the input is a graph G and a parameter k. The goal is to find a k-triangulation of G, or to determine that none exists. Kaplan, Shamir and Tarjan [118] and later independently Cai [28] proved that the minimum fill-in problem is fixed parameter tractable, by giving an algorithm of complexity $2^{O(k)}m$ for the problem. Both used the same algorithm, with the time bound in [28] being slightly tighter. Kaplan, Shamir and Tarjan also gave a more efficient $2^{O(k)} + O(k^2nm)$-time algorithm (henceforth, KST algorithm) for the problem.

Here we give the first polynomial approximation algorithm for the minimum fill-in problem. Our algorithm builds on ideas from [118]. For an input graph G with minimum fill-in of size k, our algorithm produces a triangulation of size at most $8k^2$ – within a factor of $8k$ of optimal. The approximation is achieved by identifying in G a *kernel* set of vertices A of size at most $4k$, such that one can triangulate G by adding edges only between vertices of A. Our algorithm produces the triangulation without prior knowledge of k. Let $M(n)$ denote the number of operations needed to multiply two integer matrices of order $n \times n$ (the current upper bound on $M(n)$ is $O(n^{2.376})$ [36]). The algorithm works in time $O(knm + \min\{n^2M(k)/k, nM(n)\})$, which makes it potentially suitable for practical use.

Our algorithm is particularly attractive for small fill-in values. Note that if $k = \Omega(n)$ then our algorithm guarantees only the trivial bound of fill-in size – $O(n^2)$, but if for example the fill-in size is constant then the approximation guarantee is a constant.

We also obtain better approximation results for bounded degree graphs. For graphs with maximum degree d we give a polynomial algorithm which achieves an approximation ratio of $O(d^{2.5}\log^4(kd))$. Since $k = O(n^2)$ this approximation ratio is polylogarithmic in the input size.

In order to compare our results to the approximation results regarding MTS, we translate the latter to approximation ratios in terms of the fill-in obtained. We assume throughout that $m > n$. For general graphs the algorithm in [5] guarantees that the number of edges in the chordal supergraph obtained is $O((m + k)^{3/4}\sqrt{m}\log^{3.5} n)$. In terms of the fill-in size, the approximation ratio achieved is $O(m^{1.25}\log^{3.5} n/k + \sqrt{m}\log^{3.5} n/k^{1/4})$. We obtain a better approximation ratio when-

ever $k = O(m^{5/8} \log^{1.75} n)$. For graphs with maximum degree d, the algorithm in [5] achieves an approximation ratio of $O(((nd + k)\sqrt{d} \log^4 n)/k)$. We provide a better ratio when $k = O(n/d)$. When any of these upper bounds on k is satisfied, our algorithm also achieves a better approximation ratio than [5] for the MTS problem.

Kaplan, Shamir and Tarjan posed in [118] an open problem of obtaining an algorithm for the parametric fill-in problem with time $2^{O(k)} + O(km)$. The motivation is to match the performance of the $2^{O(k)}m$ algorithm for all k. We make some progress towards solving that problem by providing a faster $2^{O(k)} + O(knm + \min\{n^2 M(k)/k, nM(n)\})$-time implementation of their algorithm. We also give a variant of the algorithm which produces a smaller kernel. Finally, we apply our approximation algorithm to Chain Completion and obtain an approximation ratio of $8k$, where k denotes the size of an optimum solution.

The chapter is organized as follows: Section 3.2 contains a description of KST algorithm and some background. Section 3.3 improves the complexity of KST algorithm and reduces the size of the kernel produced. Section 3.4 describes our approximation algorithm for the minimum fill-in problem on general graphs. Section 3.5 gives an approximation algorithm for graphs with bounded degree. Section 3.6 gives further reduction of the kernel size, and Section 3.7 gives an approximation algorithm for Chain Completion.

3.2 Preliminaries

Our polynomial approximation algorithm for the minimum fill-in problem builds on KST algorithm [118]. In the following we describe this algorithm. Our presentation generalizes that in [118], in order to allow succinct description of the approximation algorithm in Section 3.4.

Fact 3.2.1 *A minimal triangulation of a chordless l-cycle consists of $l-3$ edges.*

Lemma 3.2.2 *[118, Lemma 2.5] Let C be a chordless cycle and let p be an l-path on C, $1 \leq l \leq |C|-2$. If $l = |C|-2$ then in every minimal triangulation of C there are at least $l-1$ chords incident to vertices of p. If $l < |C|-2$ then in every minimal triangulation of C there are at least l chords incident to vertices of p.*

Let $< G = (V, E), k >$ be an input for the parametric fill-in problem. The algorithm has two main stages. In the first stage, which is polynomial in n, m and k, the algorithm produces a partition (A, B) of V and a set F of non-edges in G_A, such that $|A| = O(k^3)$ and no chordless cycle in $G' = (V, E \cup F)$ intersects B. We shall call this stage the *partition algorithm*.

In the second stage, which is exponential in k, an exhaustive search is applied to find a minimum triangulation F' of G'_A. $F \cup F'$ is then proved to be a minimum triangulation of G. The search procedure can be viewed as traversing part of a search tree T, which is defined as follows: Each tree node v corresponds to a supergraph $G(v)$ of G. For the root r, $G(r) = G$. Each leaf of T corresponds to a chordal supergraph of G. At an internal node v, a chordless cycle C in $G(v)$ is identified. For each minimal triangulation F_C of C, a node u is added as a child of v, and its corresponding graph $G(u)$ is obtained by adding F_C to $G(v)$. The algorithm visits only nodes v of T for which $|E(G(v)) \setminus E| \leq k$. If such a node is a leaf, then the search terminates successfully. Otherwise, no k-triangulation exists for G.

The partition algorithm applies sequentially the following three procedures. All three maintain a partition (A, B) of V and a lower bound cc on the minimum number of edges needed to triangulate G. Initially $A{=}\emptyset, B{=}V$ and $cc{=}0$.

- Procedure $P_1(k)$: *Extracting independent chordless cycles.* Search repeatedly for chordless cycles in G_B and move their vertices from B to A. For each chordless l-cycle found, increment cc by $l - 3$. If at any time $cc > k$, stop and declare that the graph admits no k-triangulation.

- Procedure $P_2(k)$: *Extracting related chordless cycles with independent paths.* Search repeatedly for chordless cycles in G containing at least two consecutive vertices from B. Let C be such a cycle, $|C| = l$. If $l > k + 3$ stop with a negative answer. Otherwise, suppose that C contains $j \geq 1$ disjoint maximal sub-paths in G_B, each of length at least 1. Move the vertices of those sub-paths from B to A. Denote their lengths in non-increasing order by $l_1, \ldots, l_j$. If $j = 1$ we increase cc either by $l_1{-}1$ if $l_1{=}l{-}2$, or by l_1 if $l_1{<}l{-}2$. Otherwise, cc is increased by $\max\{\frac{1}{2}\sum_{i=1}^{j} l_i, l_1\}$. If at any time $cc > k$, stop and declare that the graph admits no k-triangulation.

Definition 3.2.3 *For every $x, y \in A$ such that $(x, y) \notin E$, denote by $A_{x,y}$ the set of all vertices $b \in B$ such that x, b, y occur consecutively on some chordless cycle in G.*

If $|A_{x,y}| > 2k$ *then* (x, y) *is called a* k-essential edge.

- Procedure $P_3(k)$: *Adding* k*-essential edges in* G_A. For every $x, y \in A$ such that $(x, y) \notin E$ compute the set $A_{x,y}$. If (x, y) is k-essential, then add it to G. Otherwise, move all vertices in $A_{x,y}$ from B to A.

Denote by A^i, B^i the partition obtained after procedure P_i is completed, for $i = 1, 2, 3$. We shall omit the index i when it is clear from the context. Denote by cc_i the value of cc after procedure P_i is completed, for $i = 1, 2$. The size of A^2 is at most $4k$ since $k \geq cc_2 = cc_1 + (cc_2 - cc_1) \geq \frac{1}{4}|A^1| + \frac{1}{2}|A^2 \setminus A^1| \geq \frac{1}{4}|A^2|$. The size of A^3 is $O(k^3)$ since there are $O(k^2)$ non-edges in G_{A^2} and the number of vertices moved to A due to any such non-edge is at most $2k$.

Execute procedure $P_1(k)$.
Execute procedure $P_2(k)$.
Execute procedure $P_3(k)$.

Figure 3.1: KST partition algorithm.

The partition algorithm is summarized in Figure 3.1. Let G' denote the graph obtained after the execution of procedure P_3. Kaplan, Shamir and Tarjan prove that every k-essential edge must appear in any k-triangulation of G [118, Lemma 2.7], and that in G' no chordless cycle intersects B [118, Theorem 2.10]. The following theorem shows that it suffices to search for a minimum triangulation of G'_A.

Theorem 3.2.4 *[118, Theorem 2.13] Let* A, B *be a partition of the vertex set of a graph* G*, such that the vertices of every chordless cycle in* G *are contained in* A*. A set of edges* F *is a minimal triangulation of* G *if and only if* F *is a minimal triangulation of* G_A*.*

The complexity of the partition algorithm is $O(k^2nm)$ [118]. The complexity of finding a minimum triangulation of a given graph is $O(\frac{4^k}{(k+1)^{3/2}}m)$ [28]. Since G'_A contains $O(k^6)$ edges, a minimum triangulation of G'_A can be found in $O(k^{4.5}4^k)$ time. Hence, the complexity of KST algorithm is $O(k^2nm + k^{4.5}4^k)$.

3.3 Improvements to the Partition Algorithm

In this section we show some improvements to KST partition algorithm. We assume throughout that the input is $< G = (V, E), k >$. We first show how to implement procedure P_3 in $O(nm+\min\{n^2M(k)/k, nM(n)\})$-time. We then prove that the size of A^3 is only $O(k^2)$. These results imply that KST algorithm can be implemented in $O(knm + \min\{n^2M(k)/k, nM(n)\} + k^{2.5}4^k)$-time.

Lemma 3.3.1 *Procedure P_3 can be implemented in $O(nm+\min\{n^2M(k)/k, nM(n)\})$ time.*

Proof: Let $S = \{(x, y) \notin E : x, y \in A^2\}$. The bottleneck in the complexity of P_3 is computing the sets $A_{x,y}$ for every $(x, y) \in S$. To this end, we find for every $b \in B$ all pairs $(x, y) \in S$ such that $b \in A_{x,y}$. We then construct the sets $A_{x,y}$. This is done as follows:

Fix $b \in B$. Compute the connected components of $G^b = G \setminus N[b]$. This takes $O(m)$ time. Denote the connected components of G^b by $C^b_1, \ldots, C^b_l$. For each $x \in A^2 \cap N(b)$ compute a binary vector $\vec{v}_x = (v^x_1, \ldots, v^x_l)$ such that $v^x_j = 1$ if and only if C^b_j contains a neighbor of x, $1 \leq j \leq l$. Each vector can be computed in $O(n)$ time. Let $k' = |A^2 \cap N(b)|$. Number the vertices in $A^2 \cap N(b)$ arbitrarily according to some 1-1 mapping $\sigma : \{1, \ldots, k'\} \to A^2 \cap N(b)$. Define a $k' \times l$ boolean matrix M whose i-th row is the vector $\vec{v}_{\sigma(i)}$, $1 \leq i \leq k'$. Note that $k' = O(\min\{k, n\})$ and $l \leq n$. Let $M^* = MM^T$. It can be seen that for every pair (i, j) such that $1 \leq i < j \leq k'$ and $(\sigma(i), \sigma(j)) \in S$, $M^*_{i,j} \geq 1$ if and only if $b \in A_{\sigma(i),\sigma(j)}$. Since $k', l \leq n$ we can compute M^* in $O(M(n))$ time. If $k = o(n)$ then we can compute M^* in $O(nM(k)/k)$ time by partitioning M and M^T into $\lceil n/k' \rceil$ submatrices of order at most $k' \times k'$, multiplying corresponding pairs of sub-matrices, and summing the results. Hence, the computation of M^* takes $O(\min\{nM(k)/k, M(n)\})$ time.

After the above calculations are performed for every $b \in B$, it remains to compute the sets $A_{x,y}$. We can do that in $O(\min\{k^2n, n^3\})$ time. The total time is therefore $O(nm + \min\{n^2M(k)/k, nM(n)\})$. ■

Observation 3.3.2 *Let $x, y \in A^2, (x, y) \notin E$. If $A_{x,y} \neq \emptyset$ then for any triangulation F of G, either $(x, y) \in F$ or for every $b \in A_{x,y}$, F contains an edge incident to b.*

Lemma 3.3.3 *Assume that G admits a k-triangulation. If in procedure P_3 all sets $A_{x,y}$ that are moved into A have size at most d, then $|A^3 \setminus A^2| \leq Mk$, where $M = \max\{d, 2\}$.*

Proof: Let the non-edges in G_{A^2} be $(x_1, y_1), \ldots, (x_l, y_l)$. We process the sets $A_{x_1,y_1}, \ldots, A_{x_l,y_l}$ in this order. Let $A^{(0)} = A^2$. Let $A^{(i)}$ be the set A right after A_{x_i,y_i} was processed, and let $\Delta_i = A_{x_i,y_i} \setminus A^{(i-1)}$, for $1 \leq i \leq l$.

Let t be a lower bound on the minimum number of edges needed to triangulate G. Initially P_3 starts with $t = 0$. Let t_i be the value of t right after A_{x_i,y_i} was processed ($t_0 = 0$). If $\Delta_i \neq \emptyset$ then by Observation 3.3.2 t should increase by $\min\{1, |\Delta_i|/2\}$. We must maintain $t \leq k$. If $t_i - t_{i-1} = 0$ then $|\Delta_i| = 0$. If $t_i - t_{i-1} = 1/2$ then $|\Delta_i| = 1$. If $t_i - t_{i-1} \geq 1$ then $|\Delta_i| \leq d$. Therefore for all $1 \leq i \leq l$, $|\Delta_i| \leq M(t_i - t_{i-1})$. Now,

$$\begin{aligned} |A^3 \setminus A^2| &= |A^{(l)} \setminus A^{(0)}| = \sum_{i=1}^{l} |A^{(i)} \setminus A^{(i-1)}| = \sum_{i=1}^{l} |\Delta_i| \leq M \sum_{i=1}^{l} (t_i - t_{i-1}) \\ &= M(t - t_0) \leq Mk \ . \end{aligned}$$

■

Corollary 3.3.4 *If G admits a k-triangulation, then the partition algorithm terminates with $|A| \leq 2k(k+2)$.*

Proof: Let us assume that all k-essential edges were added to G, and denote the new set of edges of G by E'. For all $x, y \in A^2, (x, y) \notin E'$ we have $|A_{x,y}| \leq 2k$. By Lemma 3.3.3, $|A^3 \setminus A^2| \leq 2k^2$. Since $|A^2| \leq 4k$ the corollary follows. ■

Theorem 3.3.5 *KST algorithm takes $O(knm + \min\{n^2 M(k)/k, nM(n)\} + k^{2.5} 4^k)$ time.*

Proof: By the analysis in [118] P_1 takes $O(km)$ time and P_2 takes $O(knm)$ time. By Lemma 3.3.1 the complexity of P_3 is $O(nm + \min\{n^2 M(k)/k, nM(n)\})$. By Corollary 3.3.4, if G admits a k-triangulation then the size of A^3 is $O(k^2)$. Hence, a minimum triangulation of G'_A can be found in $O(k^{2.5} 4^k)$ time [28]. The complexity follows. ■

3.4 The Approximation Algorithm

Let $G = (V, E)$ be the input graph. Let $k_{opt} = \Phi(G)$. The key idea in our approximation algorithm is to find a set of vertices $A \subseteq V$, such that $|A| = O(k_{opt})$ and one can triangulate G by adding edges only between vertices of A. Since there are $O(k_{opt}^2)$ non-edges in G_A, we achieve an approximation ratio of $O(k_{opt})$.

In order to find such a set A we use ideas from the partition algorithm. If we knew k_{opt} we could execute the partition algorithm and obtain a set A, with $|A| = O(k_{opt}^2)$ (by Corollary 3.3.4), such that G can be triangulated by adding edges only between vertices of A. This would already give an $O(k_{opt}^3)$ approximation ratio.

Before describing our algorithm we analyze the role of the parameter k given to the partition algorithm. If $k < k_{opt}$ then the algorithm might stop during P_1 or P_2 and declare that no k-triangulation exists. Moreover, k-essential edges are not necessarily k_{opt}-essential. If $k > k_{opt}$ then the size of A may be $\omega(k_{opt}^2)$. The approximation algorithm is shown in Figure 3.2.

Algorithm APPROX
  Procedure P_1': Execute $P_1(\infty)$.
  Procedure P_2': Execute $P_2(\infty)$.
  Procedure P_3': Execute $P_3(0)$.
  Let G' be the resulting graph.
  Procedure P_4': Find a minimal triangulation of G'_A.

Figure 3.2: The approximation algorithm.

Procedures P_1' and P_2' execute P_1 and P_2 respectively, without bounding the size of the triangulation implied. Procedure P_3' takes advantage of the fact that we no longer seek a minimum triangulation, but rather a minimal one. In order to obtain our approximation result we want to keep A as small as possible. Hence, instead of moving new vertices to A we add new 0-essential edges accommodating for those vertices. By the same arguments as in [118] and Section 3.2, the size of A after the execution of P_2' is at most $4k_{opt}$. Since P_3' does not add new vertices to A, its size remains at most $4k_{opt}$ throughout. The size of the triangulation found by the algorithm is therefore at most $8k_{opt}^2 - 2k_{opt}$. The correctness of algorithm APPROX is established in the sequel. We need the following lemma which is implied by the

proof of [118, Lemma 2.9]. The subsequent theorem is a generalization of [118, Theorem 2.10].

Lemma 3.4.1 *Let $G = (V, E)$ be a graph and let $v \in V$. Let F be a set of non-edges in $G \setminus \{v\}$, such that each $e = (x, y) \in F$ is a chord in a chordless cycle $C_e = (x, z_e, y, \ldots, x)$ in G, where z_e is not an endpoint of any edge in F. Let $G' = (V, E \cup F)$. If there exists a chordless cycle C in G' with v_1, v, v_2 occurring consecutively on C, for some $v_1, v_2 \in N(v)$, then either there exists a chordless cycle in G on which v_1, v, v_2 occur consecutively, or there exists a chordless cycle in G on which v and z_e occur consecutively, for some $e \in F$.*

Theorem 3.4.2 *Let $G = (V, E)$ be a graph. Let A, B be a partition of V such that no chordless cycle in G contains two consecutive vertices from B. Let $S = \{(x, y) \notin E : x, y \in A, A_{x,y} \neq \emptyset\}$. Then for any choice of $F \subseteq S$ no chordless cycle in $G' = (V, E \cup F)$ intersects $B' = B \setminus (\bigcup_{(x,y)\in S\setminus F} A_{x,y})$.*

Proof: Suppose to the contrary that C is a chordless cycle in G' intersecting B'. Let $v \in C \cap B'$. Let v_1 and v_2 be the neighbors of v on C. Since $v \in B'$, it is not an endpoint of any edge in F. Every edge $e = (x, y) \in F$ is a chord in a chordless cycle $C_e = (x, z_e, y, \ldots, x)$ of G, where $z_e \in B$. Applying Lemma 3.4.1 we find that two cases are possible:

1. There exists a chordless cycle in G on which v_1, v, v_2 occur consecutively. If $v_1 \in B$ or $v_2 \in B$ we arrive at a contradiction. Hence, $v_1, v_2 \in A$ and $v \in A_{v_1,v_2}$. We conclude that either $(v_1, v_2) \in F$ or $v \notin B'$, a contradiction.

2. There exists a chordless cycle in G on which v and z_e occur consecutively (for some $e \in F$), a contradiction. ■

Theorem 3.4.3 *Let G be a graph and let $k_{opt} = \Phi(G)$. The algorithm finds a triangulation of G of size at most $8k_{opt}^2 - 2k_{opt}$, and can be implemented to run in time $O(k_{opt}nm + \min\{n^2M(k_{opt})/k_{opt}, nM(n)\})$.*

Proof: Correctness: By Theorems 3.4.2 and 3.2.4 a minimal triangulation of G'_A is a minimal triangulation of G'. Therefore, at the end of the algorithm G is

triangulated. Throughout the algorithm the only edges added to G are between vertices of A. Since $|A| \leq 4k_{opt}$ the size of the triangulation is at most $8k_{opt}^2 - 2k_{opt}$.

Complexity: The complexity analysis of procedures P_1 and P_2 in [118] implies that P_1' and P_2' can be performed in $O(k_{opt}nm)$ time. By Lemma 3.3.1 the complexity of P_3' is $O(nm + \min\{n^2M(k_{opt})/k_{opt}, nM(n)\})$. Procedure P_4' requires finding a minimal triangulation of G'_A. Since $|A| = O(\min\{k_{opt}, n\})$ and $|E(G'_A)| = O(\min\{k_{opt}^2, n^2\})$, this requires $O(\min\{k_{opt}^3, n^3\})$ time [152]. Hence, the total running time is $O(k_{opt}nm + \min\{n^2M(k_{opt})/k_{opt}, nM(n)\})$. ■

Note that although our analysis uses an upper bound of $\binom{t}{2}$ for the triangulation size of a t-vertex graph, replacing G'_A by the complete graph is not guaranteed to produce a triangulation of G.

3.5 Bounded Degree Graphs

In order to improve the approximation ratio for bounded degree graphs, we improve P_4'. Instead of simply finding a minimal triangulation of G'_A, we use the triangulation algorithm of Agrawal, Klein and Ravi [5]. This alone does not suffice to prove a better approximation ratio, since adding 0-essential edges (in P_3') might not be optimal. In other words, if we denote by F the set of 0-essential edges added to G by P_3', then it might be that $|F| + \Phi(G') > \Phi(G)$. To overcome this difficulty we use KST partition algorithm with $k = \infty$ as its input parameter, which implies that no new edge will be added to G_A by P_3. The approximation algorithm is as follows:

1. Execute KST partition algorithm with parameter $k = \infty$.
2. Find a minimal triangulation of G_A using the algorithm in [5].

Assume that the input graph G has maximum degree d, and let $k = \Phi(G)$. We will show that the algorithm achieves an approximation ratio of $O(d^{2.5}\log^4(kd))$. Since $k = O(n^2)$, this is in fact a polylogarithmic approximation ratio. It improves over the $O(k)$ approximation ratio obtained in the previous section, when $k/\log^4 k = \Omega(d^{2.5})$.

Theorem 3.5.1 *The algorithm finds a triangulation of G whose size is within a factor of $O(d^{2.5}\log^4(kd))$ of optimal.*

Proof: **Correctness:** By the correctness of KST partition algorithm, we obtain a partition (A, B) of $V(G)$ for which no chordless cycle in G intersects B. By Theorem 3.2.4 a minimal triangulation of G_A is a minimal triangulation of G. Therefore, the algorithm correctly computes a minimal triangulation of G.

Approximation Ratio: When executing P_3 the size of each set $A_{x,y}$ is at most d. By Lemma 3.3.3 $|A^3 \setminus A^2| = O(kd)$. Since $|A^2| = O(k)$, the size of A when the partition algorithm terminates is $O(kd)$. Setting the parameter value to ∞ in P_3 guarantees that no new edge is added to G_A and, therefore its maximum degree is at most d and $|E(G_A)| = O(kd^2)$. Using the algorithm in [5] we can produce a chordal supergraph of G_A with $O((kd^2 + k)\sqrt{d}\log^4(kd))$ edges. Hence, the size of the fill-in obtained is within a factor of $O(d^{2.5}\log^4(kd))$ of optimal. ■

3.6 Reducing the Kernel Size

We now return to the parametric fill-in problem. Let $< G = (V, E), k >$ be the input instance. By modifying procedure P_3 in KST partition algorithm we shall obtain a partition (A, B) of V and a set of non-edges F, such that no chordless cycle in $G' = (V, E \cup F)$ intersects B and $|A| = O(k)$. In fact we shall obtain at most 2^k such pairs (A, F), and prove that if G has a k-triangulation, then for at least one of those pairs G'_A admits a $(k - |F|)$-triangulation. Reducing the size of A results in improving the complexity of finding a minimum triangulation of G'_A to $O(\sqrt{k}4^k)$, although the total time of the algorithm increases, since we have to handle up to 2^k pairs. We include this result, since it gives further insight on the problem and presents ideas that may help resolve the open problem posed in [118].

As in the original algorithm we start by executing procedures $P_1(k)$ and $P_2(k)$. We also compute the sets $A_{x,y}$ for all $x, y \in A^2, (x, y) \notin E$. If (x, y) is k-essential, we add it to G. Otherwise, we do nothing. Let $\hat{E}$ be the set of k-essential edges, and let $e = |\hat{E}|$. Define $P \equiv \{(x, y) \notin E \cup \hat{E} : x, y \in A^2, A_{x,y} \neq \emptyset\}$, and let $p = |P|$.

The algorithm now enumerates subsets $F \subseteq P$. For a given set F, every $(x, y) \in F$ is added as an edge in the triangulation, and for every $(x, y) \in P \setminus F$ the vertices in $A_{x,y}$ are moved from B to A (which was initialized to A^2). Instead of directly enumerating each set F we branch and bound: We construct these sets incrementally, and stop when a lower bound for the size of the triangulation implied by F exceeds

k.

Specifically, pairs in P are considered in an arbitrary order (x_0, y_0),. . .,(x_{p-1}, y_{p-1}). For the current pair (x_i, y_i) the algorithm distinguishes between three cases as follows. Let $t = |A_{x_i,y_i} \setminus A|$ with respect to the current A. Let cc denote a lower bound for the size of the triangulation implied by the set F constructed so far (cc is initialized to e). If $t = 0$ then the algorithm does nothing. If $t = 1$, it updates A to $A \cup A_{x_i,y_i}$ and increases cc by 1/2. Finally, if $t \geq 2$ then the algorithm branches into two cases. In the first case, (x_i, y_i) is added to the triangulation and cc is increased by 1. In the second case, the vertices of A_{x_i,y_i} are moved from B to A, and cc is increased by $t/2$. The algorithm is implemented by the recursive procedure shown in Figure 3.3, and is invoked by calling BRANCH$(e, \emptyset, 0, A^2)$.

```
Procedure BRANCH(cc, F, r, A)
  If cc > k then return.
  If r = p then save the pair (A, F ∪ Ê) and return.
  Let t = |A_{x_r,y_r} \ A|.
  If t = 0 then
    Call BRANCH(cc, F, r + 1, A).
  Else if t = 1 then
    Call BRANCH(cc + 1/2, F, r + 1, A ∪ A_{x_r,y_r}).
  Else /* t ≥ 2 */
    Call BRANCH(cc + 1, F ∪ {(x_r, y_r)}, r + 1, A).
    Call BRANCH(cc + t/2, F, r + 1, A ∪ A_{x_r,y_r}).
```

Figure 3.3: Algorithm BRANCH.

Lemma 3.6.1 *The algorithm terminates after at most $p2^{k+1} + 1$ calls to procedure BRANCH.*

Proof: Denote by $T(i, j)$ the number of recursive calls invoked by BRANCH when called with parameters $cc = i, r = j$ (including this first call). Since always $i \geq 0$ and $0 \leq j \leq p$ in the following we consider these ranges only. Clearly, $T(i, j) \leq 1 + \max\{T(i, j + 1), T(i + 1/2, j + 1), 2T(i + 1, j + 1)\}$, for all $j < p, i$.

Also, $T(i,j) = 1$ for all $i > k, j$; and $T(i,p) = 1$ for all i. Since $T(0,0) \geq T(i,j)$ for all i,j, it suffices to compute an upper bound for $T(0,0)$.

We prove that $T(i,j) \leq (p-j)2^{k+1-i}+1$ by induction on i,j. For $i > k$ or $j = p$ the claim is true. Suppose the claim holds for all i,j where $i' \leq i \leq k$, $j' < j \leq p$. Then for $i = i'$ and $j = j'$ we have

$$\begin{aligned} T(i,j) &\leq 1 + \max\{T(i,j+1), T(i+1/2,j+1), 2T(i+1,j+1)\} \\ &\leq 2 + \max\{(p-j-1)2^{k+1-i}, (p-j-1)2^{k+\frac{1}{2}-i}, (p-j-1)2^{k+1-i}+1\} \\ &\leq 3 + (p-j-1)2^{k+1-i} \leq (p-j)2^{k+1-i}+1 \ . \end{aligned}$$

It follows that $T(0,0) \leq p2^{k+1}+1$. ■

Lemma 3.6.2 *The number of pairs saved by the algorithm is at most* 2^k.

Proof: The proof is analogous to that of Lemma 3.6.1: Denote by $N(i,j)$ the number of pairs saved by procedure BRANCH, when invoked with parameters $cc = i, r = j$. Since always $i \geq 0$ and $0 \leq j \leq p$, in the following we consider these ranges only. Clearly, $N(i,j) \leq \max\{N(i,j+1), N(i+1/2,j+1), 2N(i+1,j+1)\}$, for all $j < p, i$. Also, $N(i,j) = 0$ for all $i > k, j$; $N(k,j) \leq 1$ for all j; and $N(i,p) \leq 1$ for all i. Since $N(0,0) \geq N(i,j)$ for all i,j, it suffices to compute an upper bound for $N(0,0)$.

We prove that $N(i,j) \leq 2^{k-i}$ by induction on i,j. If $i \geq k$ or $j = p$ then the claim is true. Suppose the claim holds for all i,j where $i' \leq i < k$ and $j' < j \leq p$. Then for $i = i'$ and $j = j'$ we have

$$\begin{aligned} N(i,j) &\leq \max\{N(i,j+1), N(i+1/2,j+1), 2N(i+1,j+1)\} \\ &\leq \max\{2^{k-i}, 2^{k-\frac{1}{2}-i}, 2^{k-i}\} \\ &= 2^{k-i} \ . \end{aligned}$$

It follows that $N(0,0) \leq 2^k$. ■

As usual, for a set $A \subseteq V$ saved by the algorithm, B denotes $V \setminus A$. The following two claims establish the correctness of our partition algorithm.

Lemma 3.6.3 *For every pair* (A,F) *saved by the algorithm,* $|A| \leq 6k$*, and no chordless cycle in* $G' = (V, E \cup F)$ *intersects* B.

Proof: Whenever a partition is saved $cc \leq k$. By definition of BRANCH, $\frac{1}{2}|A \setminus A^2| \leq cc$. Hence, at most $2k$ new vertices are added to A^2 in any partition obtained. Since $|A^2| \leq 4k$ we conclude that $|A| \leq 6k$. By Theorem 3.4.2 no chordless cycle in G' intersects B. ■

Definition 3.6.4 *A pair (A, F) saved by BRANCH is called* good *if $\Phi(G) = \Phi(G') + |F|$, where $G' = (V, E \cup F)$.*

Proposition 3.6.5 *If $\Phi(G) \leq k$ then at least one good pair is saved by the algorithm.*

Proof: Let T be the tree of recursive calls of BRANCH. The nodes of T corresponds to invocations of BRANCH. The root of T corresponds to the first invocation of BRANCH. The leaves of T correspond to invocations of BRANCH in which either a pair was saved, or cc was found to exceed k. In nodes at level i of T, $0 \leq i < p$, the pair $(x_i, y_i) \in P$ is processed. Let cc_v, F_v, r_v and A_v denote the parameters of the invocation of BRANCH which corresponds to node v of T.

Let F^* denote a minimum triangulation of G. The proof will identify a root-leaf path in T which corresponds to F^*, and trace the changes to cc, A and F along that path. We use the following notation:

$$\begin{aligned} P_v &\equiv \{(x_0, y_0), \ldots, (x_{r_v-1}, y_{r_v-1})\} , \\ F_v^* &\equiv P_v \cap F^* , \\ A_v^* &\equiv A^2 \cup \bigcup_{(x,y) \in P_v \setminus F_v^*} A_{x,y} , \\ cc_v^* &\equiv e + |F_v^*| + \frac{1}{2}|A_v^* \setminus A^2| . \end{aligned}$$

Lemma 3.6.6 *For every node v of T, $cc_v^* \leq k$.*

Proof: Let v be any node of T. Let $cc^* = e + |P \cap F^*| + \frac{1}{2}|\bigcup_{(x,y) \in P \setminus F^*} A_{x,y}|$. Since $P_v \subseteq P$, it follows that $cc_v^* \leq cc^*$. By Observation 3.3.2, for every pair $(x, y) \in P$, either $(x, y) \in F^*$, or F^* contains edges incident to every $b \in A_{x,y}$. Hence, $cc^* \leq |F^*| \leq k$ where the last inequality follows from the fact that F^* is a k-triangulation. ■

We now return to the proof of Proposition 3.6.5. We shall prove that T has a leaf in which a good pair is saved. To this end, we show that for every $0 \leq i \leq p$, T contains some vertex v at level i, for which $F_v \subseteq F_v^*$ and $cc_v \leq cc_v^*$. In particular, this claim implies that T has a leaf z at level p, for which $F_z \subseteq F_z^*$ and $cc_z \leq cc_z^*$. By Lemma 3.6.6, $cc_z \leq cc_z^* \leq k$. Hence, a pair $(A_z, F_z \cup \hat{E})$ is saved at z. By [118, Lemma 2.7], $\hat{E} \subseteq F^*$. In addition, $F_z \subseteq F_z^* \subseteq F^*$. Therefore $(A_z, F_z \cup \hat{E})$ is a good pair, since $F_z \cup \hat{E} \subseteq F^*$ and by definition $F^* \setminus (F_z \cup \hat{E})$ triangulates $G' = (V, E \cup F_z \cup \hat{E})$.

We prove the claim by induction on i. The base of the induction is obvious, as for the root r at level 0, $F_r = \emptyset$ and $cc_r = e$. We assume that the claim is true for level $i-1$ ($i > 0$) and prove its correctness for level i. By the induction hypothesis T contains a node v at level $i - 1 < p$, for which $F_v \subseteq F_v^*$ and $cc_v \leq cc_v^*$. By Lemma 3.6.6 $cc_v \leq cc_v^* \leq k$ and, therefore, v is not a leaf. Thus, v has either one or two children in T. There are two cases to examine:

1. Suppose that $(x_{i-1}, y_{i-1}) \in F^*$. Then for any child w of v, $cc_w^* = cc_v^* + 1 \geq cc_v + 1$. If v has a single child w then $F_w = F_v \subseteq F_v^* \subset F_w^*$ and $cc_w \leq cc_v + 1/2 < cc_w^*$. Otherwise, let w be the child of v for which $(x_{i-1}, y_{i-1}) \in F_w$. Then clearly $F_w \subseteq F_w^*$ and $cc_w = cc_v + 1 \leq cc_w^*$.

2. Suppose that $(x_{i-1}, y_{i-1}) \notin F^*$. Since $F_v \subseteq F_v^*$, it follows that $A_v^* \subseteq A_v$. Let w be the child of v for which $(x_{i-1}, y_{i-1}) \notin F_w$. Then $F_w = F_v \subseteq F_v^* = F_w^*$ and

$$cc_w = cc_v + \frac{1}{2}|A_{x_{i-1},y_{i-1}} \setminus A_v| \leq cc_v^* + \frac{1}{2}|A_{x_{i-1},y_{i-1}} \setminus A_v^*| = cc_w^* \ .$$

■

Theorem 3.6.7 *If $\Phi(G) \leq k$ then the new partition algorithm produces at least one pair (A, F) for which $|A| \leq 6k$ and $\Phi(G) = \Phi(G'_A) + |F|$, where $G' = (V, E \cup F)$. The complexity of the algorithm is $O(knm + \min\{n^2 M(k)/k, nM(n)\} + k^3 2^k)$.*

Proof: Correctness: By Lemma 3.6.3, for each pair (A, F) saved by the algorithm $|A| \leq 6k$ and no chordless cycle in G' intersects B. Therefore, by Theorem 3.2.4 for each such pair $\Phi(G') = \Phi(G'_A)$. Since $\Phi(G) \leq k$, by Proposition 3.6.5 the algorithm saves some pair (A, F) for which $\Phi(G) = \Phi(G') + |F|$. Correctness follows.

Complexity: By the complexity analysis in [118] P_1 and P_2 take $O(knm)$ time. By Lemma 3.3.1 the complexity of computing the sets $A_{x,y}$ for all $x, y \in A^2, (x, y) \notin E$ is $O(nm + \min\{n^2M(k)/k, nM(n)\})$. By Lemma 3.6.1 and the fact that $|P| = O(k^2)$, the number of calls to BRANCH is $O(k^2 2^k)$. By Lemma 3.6.3 and since $\Phi(G) \leq k$, the parameters A and F to each invocation of BRANCH satisfy $|A| = O(k)$ and $|F| \leq k$. Also, for all $(x, y) \in P$, $|A_{x,y}| \leq 2k$. Thus, each call can be carried out in $O(k)$ time. The total work done by BRANCH is therefore $O(k^3 2^k)$. ■

3.7 An Approximation Algorithm for Chain Completion

In this section we show a direct application of the Chordal Completion approximation result to approximate Chain Completion.

Theorem 3.7.1 *There exists a polynomial approximation algorithm for Chain Completion with an approximation ratio of $8k$, where k denotes the size of a minimum chain completion set. The complexity of the algorithm is $O(kn^3)$.*

Proof: Let $G = (U, V, E)$ be an input bipartite graph, and let k be the size of a minimum chain completion set for G. We apply the following reduction given by Yannakakis [194] from Chain Completion to Chordal Completion: Build a graph $G' = (U \cup V, E')$, where $E' = E \cup (U \otimes U) \cup (V \otimes V)$. Observe that G is a chain graph if and only if G' is chordal. Hence, a set of edges F triangulates G' if and only if $(U, V, E \cup F)$ is a chain graph.

Approximation Ratio: By the above argument k equals $\Phi(G')$. Using our approximation algorithm for the minimum fill-in problem, we can find a triangulation of G' of size at most $8k^2$. Adding these edges to G produces a chain graph. The number of new edges is within a factor of $8k$ of optimal.

Complexity: G' can be computed in $O(n^2)$ time. Due to the reduction $|E(G')| = \Theta(n^2)$. Therefore the complexity of the approximation algorithm is $O(kn^3)$. ■

Chapter 4

Dynamic Recognition Algorithms

In this chapter we study dynamic recognition problems on certain graph classes. These problems call for maintaining a representation of a graph throughout a series of on-line modifications (insertions or deletions of a vertex or an edge), as long as the graph satisfies some property, and detecting when it ceases to satisfy the property. This chapter contains two parts. In the first part we present a fully dynamic algorithm for proper interval graph recognition and representation. The algorithm handles an operation involving d edges in time $O(d + \log n)$. (In case of an edge modification $d = 1$, and in case of a vertex modification d equals its degree.) We also prove a close lower bound of $\Omega(\log n/(\log\log n + \log b))$ for an edge operation in the cell probe model of computation with word-size b. In addition, we give algorithms requiring $O(d)$ time per operation for variants of the problem where either only addition operations are allowed, or only deletion operations are allowed. The latter algorithms are optimal with respect to all operations, with the possible exception of vertex deletion. This study was published in [99].

In the second part we provide a fully dynamic algorithm for cograph recognition, which works in $O(d)$ time per operation involving d edges. The algorithm maintains utilizes a modular decomposition tree of the dynamic graph. We derive from this result fully dynamic algorithms for threshold recognition and for trivially perfect graph recognition. These algorithms are optimal with respect to all operations, with the possible exception of vertex deletion.

4.1 Background

In a *dynamic graph problem* one has to maintain a graph throughout a series of online modifications (insertion or deletion of a vertex or an edge) and answer queries regarding certain properties of the dynamic graph. For example, in *dynamic connectivity* one has to maintain the connected components of a graph during a series of modifications and answer queries of the form "are vertices u and v connected?". Dynamic algorithms for such a problem may be of several types depending on the modification operations they support. A *vertex-incremental* (*vertex-decremental*) algorithm supports only vertex insertions (deletions). An *edge-incremental* (*edge-decremental*) algorithm supports only edge additions (deletions). An *incremental* (*decremental*) algorithm support both edge and vertex additions (deletions). An *edges-only fully dynamic* algorithm supports both edge additions and edge deletions but no vertex modifications. A *fully dynamic* algorithm supports all kinds of modifications, namely, insertions and deletions of vertices and edges.

Here we investigate *dynamic recognition problems* in which the queries are of the form: "Does the graph belong to a certain class Π?". An algorithm for the problem is required to maintain a representation of the dynamic graph as long as it belongs to Π, and to detect when it ceases to belong to Π.

A *fully dynamic algorithm* for Π-recognition maintains a data structure of the current graph $G = (V, E)$ and supports the following operations:

- **Edge Insertion**: Given a non-edge $(u, v) \notin E$, update the data structure if $G \cup \{(u, v)\} \in \Pi$, or output *False* and halt otherwise.
- **Edge Deletion**: Given an edge $(u, v) \in E$, update the data structure if $G \setminus \{(u, v)\} \in \Pi$, or output *False* and halt otherwise.
- **Vertex Insertion**: Given a new vertex $v \notin V$ and a set of edges between v and vertices of G, update the data structure if $G \cup v \in \Pi$, or output *False* and halt otherwise.
- **Vertex Deletion**: Given a vertex $v \in V$, update the data structure if $G \setminus v \in \Pi$, or output *False* and halt otherwise.

Whenever the current graph ceases to satisfy Π, the algorithm should recognize this and halt.

Traditionally, fully dynamic algorithms handle only edge modifications, since vertex modifications can be performed by a series of edge modifications. (For example, in dynamic graph connectivity adding a vertex of degree d is equivalent to adding an isolated vertex, and then adding its edges one by one.) However, in our context we have to be more careful, since we may not be able to add or delete one edge at a time without ceasing to satisfy property Π (and even if there is a way to do that, it might be non-trivial to find it). In other words, adding or deleting a vertex can preserve the property, but adding or removing one edge at a time might fail to do so. Hence, vertex operations must be handled separately by the dynamic algorithm.

Several authors have studied the problem of dynamically recognizing and representing various graph families. Corneil, Perl and Stewart [41] have given a linear-time vertex-incremental algorithm for recognizing cographs. Hsu [108] has given an $O(m + n \log n)$-time vertex-incremental algorithm for recognizing interval graphs. Deng, Hell and Huang [46] have given a linear-time vertex-incremental algorithm for recognizing and representing connected proper interval graphs. The latter algorithm requires that the graph remains connected throughout the modifications. Ibarra [110] has given an edges-only fully dynamic algorithm for recognizing chordal graphs, which handles each edge operation in $O(n)$ time, and an edges-only fully dynamic algorithm for split graph recognition, which handles each operation in constant time. Recently, Ibarra devised an edges-only fully dynamic algorithm for interval graph recognition, which handles each edge operation in $O(n \log n)$ time [111].

4.2 Proper Interval Graph Recognition

4.2.1 Introduction

This section deals with the problem of recognizing and representing dynamically changing proper interval graphs. Proper interval graphs have been studied extensively in the literature (cf. [82, 163]), and several linear time algorithms are known for their recognition and realization [39, 46].

The motivation for the problem of dynamically recognizing proper interval graphs comes from its application to *physical mapping* of DNA [30]. Physical mapping is the process of reconstructing the relative position of DNA fragments, called *clones*,

along the target DNA molecule, prior to their sequencing, based on information about their pairwise overlaps. In some biological frameworks the set of clones is virtually inclusion-free – for example when all clones have similar lengths (this is the case for instance for cosmid clones). In this case, the physical mapping problem can be modeled using proper interval graphs as follows. A graph G is built according to the biological data. Each clone is represented by a vertex and two vertices are adjacent if and only if their corresponding clones overlap. The physical mapping problem then translates to the problem of finding a realization of G, or determining that none exists.

Had the overlap information been accurate, the two problems would have been equivalent. However, some biological techniques may occasionally lead to an incorrect conclusion about whether two clones intersect, and additional experiments may change the status of an intersection between two clones. The resulting changes to the corresponding graph are the deletion of an edge, or the addition of an edge. The set of clones is also subject to changes, such as adding new clones or deleting 'bad' clones (such as chimerics [189]). These translate into addition or deletion of vertices in the corresponding graph. Thus, we would like to be able to dynamically change our graph, so as to reflect the changes in the biological data, as long as they allow us to construct a map, i.e., as long as the graph remains a proper interval graph.

Our results are as follows: For the general problem of recognizing and representing proper interval graphs we give a fully dynamic algorithm which handles each operation in time $O(d + \log n)$, where d denotes the number of edges involved in the operation. Thus, in case a vertex is added or deleted, d equals its degree, and in case an edge is added or deleted, $d = 1$. Our algorithm builds on the representation of proper interval graphs given in [46]. We prove a close lower bound of $\Omega(\log n/(\log\log n+\log b))$ amortized time per edge operation in the cell probe model of computation with word-size b [196]. It follows that our algorithm is nearly optimal (up to a factor of $O(\log\log n)$). We also give a fast $O(n)$ time algorithm for computing a realization of a proper interval graph given its representation, improving the $O(m + n)$ bound of [46].

For the incremental version of the problem we give an optimal algorithm (up to a constant factor) which handles each operation in time $O(d)$. This generalizes the result of [46] to arbitrary instances. The same bound is achieved for the decremental problem.

As part of our general algorithm we give a fully dynamic procedure for maintaining connectivity in proper interval graphs. The procedure receives as input a sequence of operations each of which is a vertex addition or deletion, an edge addition or deletion, or a query whether two vertices are in the same connected component. It is assumed that the graph remains proper interval throughout the modifications, since otherwise our recognition algorithm detects that the graph is no longer a proper interval graph and halts. We show how to implement this procedure in $O(d + \log n)$ worst-case time per operation involving d edges. In comparison, the best known algorithms for fully dynamic connectivity in general graphs require $O(\log n(\log\log n)^3)$ expected amortized time per edge operation [185], or $O(\log^2 n)$ amortized time per edge operation [107], or $O(\sqrt{n})$ worst-case time per edge operation [60]. Furthermore, we show that the lower bound of Fredman and Henzinger [100] of $\Omega(\log n/(\log\log n + \log b))$ amortized time per edge operation (in the cell probe model with word-size b) for fully dynamic connectivity in general graphs, applies also to the problem of maintaining connectivity in proper interval graphs.

This part is organized as follows: In Section 4.2.2 we give the basic background and describe our representation of proper interval graphs and the realization it defines. In Section 4.2.3 we describe the data structure used by the algorithm. In Sections 4.2.4 and 4.2.5 we present the incremental algorithm. In Section 4.2.6 we extend the incremental algorithm to a fully dynamic algorithm for proper interval graph recognition and representation. We also derive the decremental algorithm. In Section 4.2.7 we give a fully dynamic algorithm for maintaining connectivity in proper interval graphs. Finally, in Section 4.2.8 we prove lower bounds on the amortized time per edge operation of fully dynamic algorithms for recognizing proper interval graphs, and for maintaining connectivity in proper interval graphs.

4.2.2 Preliminaries

Let $G = (V, E)$ be a graph. Let R be an equivalence relation on V defined by uRv if and only if $N[u] = N[v]$. Each equivalence class of R is called a *block* of G. Note that every block of G is a complete subgraph of G. The *size* of a block is the number of vertices in it. Two blocks A and B are *adjacent*, or *neighbors*, in G, if some (and hence all) vertices $a \in A, b \in B$, are adjacent in G. A *straight enumeration* of G is a linear ordering Φ of the blocks in G, such that for every block, the block and its neighboring blocks are consecutive in Φ.

A *contig* of a connected proper interval graph G is a straight enumeration of G. The first and last blocks of a contig are called *end-blocks*, and their vertices are called *end-vertices*. The rest of the blocks are called *inner-blocks*.

Let $\Phi = B_1 < \ldots < B_l$ be a linear ordering of the blocks of G. For any $1 \leq i < j \leq l$, we say that B_i is ordered *to the left of* B_j, and that B_j is ordered *to the right of* B_i in Φ. The *out-degree* of a block B with respect to Φ, denoted by $o(B)$, is the number of neighbors of B which are ordered to its right in Φ.

We now quote some well-known properties of proper interval graphs that will be used in the sequel.

Theorem 4.2.1 *([128]) An interval graph (and in particular a proper interval graph) contains no chordless cycle.*

Theorem 4.2.2 *([190]) A graph is a proper interval graph if and only if it is an interval graph and is claw-free.*

Theorem 4.2.3 *([46]) A graph is a proper interval graph if and only if it has a straight enumeration.*

Lemma 4.2.4 ("The umbrella property") *([133]) Let Φ be a straight enumeration of a connected proper interval graph G. If A, B and C are blocks of G such that $A < B < C$ in Φ and A is adjacent to C, then B is adjacent to A and to C (see Figure 4.1).*

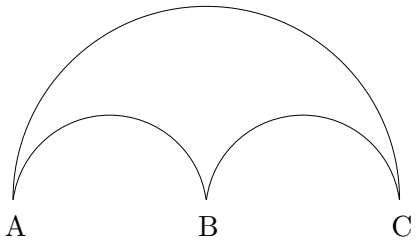


Figure 4.1: The umbrella property.

It is shown in [46] that a connected proper interval graph has a unique straight enumeration up to full reversal. This motivates our representation of proper interval graphs: For each connected component of the dynamic graph we maintain a straight

enumeration (in fact, for technical reasons we shall maintain both the enumeration and its reversal). The details of the data structure containing this information will be described in Section 4.2.3.

This information implicitly defines a realization of the dynamic graph (cf. [46]) as follows: Assign to each vertex in block B_i the interval $[i, i + o(B_i) + 1 - \frac{1}{i}]$. We show in Section 4.2.3 how to compute a realization of the dynamic graph from our data structure in time $O(n)$.

4.2.3 The Data Structure

As mentioned above, each connected component of the dynamic graph has exactly two contigs (which are reversals of each other) and both are maintained by the algorithm. Each operation involves updating the representation. In the sequel we concentrate on describing only one of the two contigs for each component. The second contig is updated in a similar way.

We now describe the details of how we keep our representation. The following data is kept and updated by the algorithm:

1. For each vertex v we keep pointers to the two blocks containing it (one in each of the two contigs that contain v).

2. For each block we keep the following:

   (a) The *size* of the block.

   (b) Left and right *near pointers*, pointing to nearest neighbor blocks on the left and on the right respectively.

   (c) Left and right *far pointers*, pointing to farthest neighbor blocks on the left and on the right respectively.

   (d) Left and right *self pointers*, pointing to the block itself.

   (e) An *end pointer* which is null if the block is not an end-block of its contig and, otherwise, points to the other end-block of that contig.

   (f) A *counter* initialized to 0.

In the following we shall omit details about the obvious updates to the pointers to the blocks containing each of the vertices (item 1), and to the block sizes (item 2a).

We introduce self pointers due to the possible need in the course of the algorithm to update many far pointers pointing to a certain block, so that they point to another block. In order to be able to do that in $O(1)$ time we use the technique of *nested pointers*: We make the far pointers point to a *location* whose content is the address of the block to which the far pointers should point. The role of this special location will be served by our self-pointers. The value of the left and right self-pointers of a block B is always the address of B. When we say that a certain left (right) far pointer points to B we mean that it points to a left (right) self-pointer of B. Let A and B be blocks. In order to change all left (right) far pointers pointing to A so that they point to B, we require that no left (right) far pointer points to B. If this is the case, we simply *exchange* the left (right) self-pointer of A with the left (right) self-pointer of B. This means that: (1) The previous left (right) self-pointer of A is made to point to B, and the algorithm records it as the new left (right) self-pointer of B; and (2) the previous left (right) self-pointer of B is made to point to A, and the algorithm records it as the new left (right) self-pointer of A.

We shall use the following notation: For a block B we denote its address in the memory by $\&B$. $\&\emptyset$ denotes the null pointer. When we set a far pointer to point to a left or to a right self-pointer of B we shall abbreviate and set it to $\&B$. We denote the left and right near pointers of B by $N_l(B)$ and $N_r(B)$ respectively. We denote the left and right far pointers of B by $F_l(B)$ and $F_r(B)$ respectively. We denote its end pointer by $E(B)$. In the sequel we often refer to blocks by their addresses. For example, if A and B are blocks and $N_r(A) = \&B$, we sometimes refer to B by $N_r(A)$. We define $N_r(\emptyset) = N_l(\emptyset) = F_r(\emptyset) = F_l(\emptyset) = \&\emptyset$. When it is clear from the context, we also use a name of a block to denote any vertex in that block. Given a contig Φ we denote its reversal by Φ^R. In general when performing an operation, we denote the graph before the operation is carried out by G, and the graph after the operation is carried out by G'.

Given this data structure we can compute a realization of a contig C of G as follows: We first rank the blocks of C, starting with the leftmost block. This is done by choosing an arbitrary block of C, and marching up the enumeration of blocks of C using left near pointers, until we reach an end-block. We then set the rank of this

block to 1, and march down the enumeration of blocks using right near pointers, until we reach the other end-block. We rank all the blocks of C along the way. Let us denote by $r(B)$ the rank of a block B. Then the out-degree of B is simply $o(B) = r(F_r(B)) - r(B)$, and the interval that we assign to the vertices of B is $[r(B), r(F_r(B)) + 1 - 1/r(B)]$. We conclude:

Theorem 4.2.5 *A realization of a proper interval graph which is represented using the data structure described above, can be computed in time $O(n)$.*

In the following two sections we describe an optimal incremental algorithm for recognizing and representing proper interval graphs. The algorithm receives as input a series of addition operations to be performed on a graph. Upon each operation the algorithm updates its representation of the graph and halts if the current graph is no longer a proper interval graph. The algorithm handles each operation in time $O(d)$, where d denotes the number of edges involved in the operation. (Thus, $d = 1$ in case of an edge addition, and d is the degree in case of a vertex addition.) It is assumed that initially the graph is empty or, alternatively, that the representation of the initial graph is known. We also show how to compute in $O(n)$ time a realization of a graph given its representation.

4.2.4 A Vertex-Incremental Algorithm

In this section we describe the updates to the representation of the graph in case G' is formed from G by the addition of a new vertex v of degree d. We also give some necessary and some sufficient conditions for deciding whether G' is a proper interval graph.

Let B be a block of G. We say that v is *adjacent* to B if v is adjacent to some vertex in B. We say that v is *fully adjacent* to B if v is adjacent to *every* vertex in B. We say that v is *partially adjacent* to B if v is adjacent to B but not fully adjacent to B.

The following lemmas characterize the adjacencies of the new vertex, assuming that G' is a proper interval graph.

Lemma 4.2.6 *If G' is a proper interval graph then v can have neighbors in at most two connected components of G.*

Proof: Suppose to the contrary that x, y and z are neighbors of v in three distinct components of G. Then v, x, y and z induce a claw in G', a contradiction. ■

Lemma 4.2.7 *[46] Let C be a connected component of G containing neighbors of v. Let $B_1 < \ldots < B_k$ be a contig of C. Suppose that G' is a proper interval graph and let $1 \leq a < b < c \leq k$. Then the following properties are satisfied:*

1. *If v is adjacent to B_a and to B_c, then v is fully adjacent to B_b.*
2. *If v is adjacent to B_b and not fully adjacent to B_a and to B_c, then B_a is not adjacent to B_c.*
3. *If $b = a + 1, c = b + 1$ and v is adjacent to B_b, then v is fully adjacent to B_a or to B_c.*

One can view a contig Φ of a connected proper interval graph C as a weak linear order $<_\Phi$ on the vertices of C, where $x <_\Phi y$ if and only if the block containing x is ordered in Φ to the left of the block containing y. We say that Φ' is a *refinement* of Φ if either for every $x, y \in V(C)$, $x <_\Phi y$ implies $x <_{\Phi'} y$; or for every $x, y \in V(C)$, $x >_\Phi y$ implies $x <_{\Phi'} y$.

Lemma 4.2.8 *If H is a connected induced subgraph of a proper interval graph H', Φ is a contig of H, and Φ' is a straight enumeration of H', then Φ' is a refinement of Φ.*

Proof: By induction on the number of additional vertices in H': If $H' = H$ then the claim is obvious. Let $k = |V(H') \setminus V(H)|$. By the induction hypothesis, for a proper interval graph H'' which contains H (as an induced subgraph) and is contained in H', and for which $|V(H'') \setminus V(H)| = k - 1$, every straight enumeration is a refinement of Φ. Let C be the connected component of H'' which contains the vertices of H, and let Φ''_C be a contig of C. Let C' be the connected component of H' which contains $V(H)$ (and, therefore, $V(C') \supseteq V(C)$), and let Φ'_C be a contig of C'. In [46] it is constructively shown how Φ'_C is obtained as a refinement of Φ''_C (see also Section 4.2.4). Since Φ''_C is a refinement of Φ, the claim follows. ■

Note that whenever v is partially adjacent to a block B in G, then the addition of v will cause B to split into two blocks of G', namely $B \setminus N(v)$ and $B \cap N(v)$.

Otherwise, if B is a block of G to which v is either fully adjacent or not adjacent, then one of B or $B \cup \{v\}$ is a block of G'.

Corollary 4.2.9 *If B is a block of G to which v is partially adjacent, then $B \setminus N(v)$ and $B \cap N(v)$ occur consecutively in a straight enumeration of G'.*

Lemma 4.2.10 *Let C be a connected component of G, which contains neighbors of v. Let $\{B_1, \ldots, B_k\}$ denote the set of blocks in C which are adjacent to v, such that in a contig of C, $B_1 < \ldots < B_k$. If G' is a proper interval graph then the following properties are satisfied:*

1. *$B_1, \ldots, B_k$ are consecutive in a contig of C.*
2. *If $k \geq 3$ then v is fully adjacent to $B_2, \ldots, B_{k-1}$.*
3. *If v is adjacent to a single block B_1 in C, then B_1 is an end-block.*
4. *If v is adjacent to more than one block in C and has neighbors in another component, then B_1 is adjacent to B_k, and one of B_1 or B_k is an end-block to which v is fully adjacent, while the other is an inner-block.*

Proof: Claims 1 and 2 follow directly from part 1 of Lemma 4.2.7. Claim 3 follows from part 3 of Lemma 4.2.7. To prove the last part of the lemma let us denote the other component containing neighbors of v by D. Examine the induced connected subgraph H of G' whose set of vertices is $V(H) = \{v\} \cup V(C) \cup V(D)$. H is a proper interval graph as an induced subgraph of G'. It is composed of three types of blocks: Blocks whose vertices are from $V(C)$, which we will henceforth call C-blocks; blocks whose vertices are from $V(D)$, which we will henceforth call D-blocks; and $\{v\}$, which is a block of H, since $H \setminus \{v\}$ is not connected. All blocks of C remain intact in H, except B_1 and B_k, each of which may split into $B_j \setminus N(v)$ and $B_j \cap N(v)$, $j = 1, k$.

Surely, in a contig of H all C-blocks must be ordered completely before or completely after all D-blocks. Let Φ denote a contig of H, in which C-blocks are ordered before D-blocks. Let X denote the rightmost C-block in Φ. By the umbrella property, $X < \{v\}$ and, moreover, X is adjacent to v. By Lemma 4.2.8, Φ is a refinement of a contig of C. Hence, $X \subseteq B_1$ or $X \subseteq B_k$ (more precisely, $X = B_1 \cap N(v)$ or $X = B_k \cap N(v)$). Therefore, one of B_1 or B_k is an end-block.

Without loss of generality, $X \subseteq B_k$. Suppose to the contrary that v is not fully adjacent to B_k. Then by Lemma 4.2.8 we have $B_{k-1} \cap N(v) < B_k \setminus N(v) < \{v\}$ in Φ (note that these blocks are not consecutive), contradicting the umbrella property. We conclude that v is fully adjacent to B_k. Furthermore, B_1 must be adjacent to B_k, or else G' contains a claw consisting of v and one vertex from each of B_1, B_k, and $V(D) \cap N(v)$. It remains to show that B_1 is an inner-block in C. Suppose it is an end block. Since B_1 and B_k are adjacent, C consists of a single block, a contradiction. Thus, claim 4 is proved. ■

The DHH Algorithm

In our algorithm we rely on the vertex-incremental algorithm of Deng, Hell and Huang [46]. This algorithm handles the insertion of a new vertex into a connected proper interval graph in $O(d)$ time, changing its straight enumeration appropriately, or determining that the new graph is not a proper interval graph. We describe it briefly below. For simplicity, we assume throughout that the modified graph is a proper interval graph.

Let H be a connected proper interval graph, and let v be a vertex to be added, which is adjacent to d vertices in H. Let $\Phi = B_1 < \ldots < B_p$ denote a contig of H. By Lemma 4.2.10, the blocks to which v is fully adjacent occur consecutively along Φ. Assume that v is fully adjacent to $B_l, \ldots, B_r$, and for clarity we shall consider only the case where $1 < l < r < p$. Let $a = l - 1$ and $c = r + 1$. By Lemma 4.2.7(2) B_a and B_c are non-adjacent. Let $b > a$ be the largest index such that B_b is adjacent to B_a, and let $d < c$ be the smallest integer such that B_d is adjacent to B_c. It is shown in [46] that $a < b < d < c$.

In order to construct a straight enumeration of the new graph we have to distinguish between two cases:

1. If v is adjacent either to B_a or to B_c, then a straight enumeration of the new graph can be obtained as follows: If v is adjacent to B_a, we split B_a into $B_a \setminus N(v), B_a \cap N(v)$, list them in this order, and add $\{v\}$ as a block just after B_b. If v is adjacent to B_c, we split B_c into $B_c \cap N(v), B_c \setminus N(v)$ in this order, and add $\{v\}$ as a block just before B_d. In case v is adjacent to both B_a and B_c these two instructions coincide, as shown in [46].

2. If v is adjacent neither to B_a nor to B_c then there are two options: If there exists a block B_j, $b < j < d$, such that B_j is adjacent to both B_l and B_r, then a straight enumeration is obtained by adding v to B_j. Otherwise, let $u > b$ be the smallest integer such that B_u is adjacent to B_r. Then a straight enumeration is obtained by inserting a new block $\{v\}$ just before B_u.

Below we show how to find the sequence of blocks $B_l, \ldots, B_r$ from our data structure in $O(d)$ time. Using near and far pointers we can identify in $O(1)$ time the blocks $B_a = N_l(B_l), B_c = N_r(B_r), B_b = F_r(B_a)$, and $B_d = F_l(B_c)$. If v is adjacent to B_a or to B_c then updating the straight enumeration can be done in $O(1)$ time. Otherwise, finding B_j (if such exists) can be done in $O(d)$ time and, alternatively, finding $B_u = F_l(B_r)$ can be done in $O(1)$ time. Again in this case we can update the straight enumeration in $O(1)$ time. Hence, our data structure supports the insertion of a vertex of degree d in $O(d)$ time, when all its neighbors are in the same connected component.

Our Algorithm

We perform the following upon a request for adding a new vertex v: We iterate over the neighbors of v. For each neighbor u of v we increment the counter of the block containing u. We call a block *full* if its counter equals its size, *empty* if its counter equals zero, and *partial* otherwise. In order to find a set of consecutive blocks that contain neighbors of v, we pick arbitrarily a neighbor of v and march up the enumeration of blocks to the left using the left near pointers. We continue till we hit an empty block or till we reach the end of the contig. We do the same to the right and this way we discover a maximal sequence of nonempty blocks in that component that contain neighbors of v. We call this maximal sequence a *segment*. Only the two extreme blocks of the segment are allowed to be partial, or else we fail (by Lemma 4.2.10(2)).

If the segment we found contains all the neighbors of v then we use the DHH algorithm in order to insert v into G, updating our internal data structure accordingly. Otherwise, by Lemmas 4.2.6 and 4.2.10(1) there could be only one more segment (in another contig) which contains neighbors of v. In that case, exactly one extreme block in each segment is an end-block to which v is fully adjacent (if the segment contains more than one block), and the two extreme blocks in each segment are

adjacent, or else we fail (by Lemma 4.2.10(3,4)).

We proceed as above to find a second segment containing neighbors of v. We can make sure that the two segments are from two different contigs by checking that their end-blocks do not point to each other. We also check that conditions 3 and 4 in Lemma 4.2.10 are satisfied for both segments. If the two segments do not cover all neighbors of v, we fail.

If v is adjacent to vertices in two distinct components C and D, then we should merge their contigs. Let $\Phi = B_1 < \ldots < B_k$ and Φ^R be the two contigs of C. Let $\Psi = B'_1 < \ldots < B'_l$ and Ψ^R be the two contigs of D. The way in which the segments are merged depends on the identity of the end-blocks to which v is adjacent in each segment. If v is adjacent to B_k and B'_1 then by the umbrella property the two new contigs (up to refinements described below) are $\Phi < \{v\} < \Psi$ and $\Psi^R < \{v\} < \Phi^R$. In the following we describe the updates to our internal data structure in case these are the new contigs. The other three cases (e.g., v is adjacent to B_1 and B'_1, etc.) are handled similarly.

- Block enumeration: We merge the two enumerations of blocks and put a new block $\{v\}$ in-between the two contigs. Let the leftmost block which is adjacent to v in the new ordering $\Phi < \{v\} < \Psi$ be B_i, and let the rightmost block adjacent to v be B'_j. If B_i is partial we split it into two blocks $\hat{B}_i = B_i \setminus N(v)$ and $B_i = B_i \cap N(v)$ and list them in this order. If B'_j is partial we split it into two blocks $B'_j = B'_j \cap N(v)$ and $\hat{B}'_j = B'_j \setminus N(v)$ in this order.

- End pointers: We set $E(B_1) = E(B'_1)$ and $E(B'_l) = E(B_k)$. We then nullify the end pointers of B_k and B'_1.

- Near pointers: We update $N_l(\{v\}) = \&B_k, N_r(\{v\}) = \&B'_1, N_r(B_k) = \&\{v\}$ and $N_l(B'_1) = \&\{v\}$. Let $B_0 = \emptyset$. If B_i was split we set $N_r(\hat{B}_i) = \&B_i, N_l(B_i) = \&\hat{B}_i, N_l(\hat{B}_i) = \&B_{i-1}$ and $N_r(B_{i-1}) = \&\hat{B}_i$. Analogous updates are made to the near pointers of $B'_j, \hat{B}'_j$ and B'_{j+1}, in case B'_j was split.

- Far pointers: If B_i was split we set $F_l(\hat{B}_i) = F_l(B_i), F_r(\hat{B}_i) = \&B_k$, and exchange the left self-pointer of B_i with the left self-pointer of $\hat{B}_i$. If B'_j was split we set $F_r(\hat{B}'_j) = F_r(B'_j), F_l(\hat{B}'_j) = \&B'_1$ and exchange the right self-pointer of B'_j with the right self-pointer of $\hat{B}'_j$. In addition, we set all right far pointers of $B_i, \ldots, B_k$ and all left far pointers of $B'_1, \ldots, B'_j$ to $\&\{v\}$ (in $O(d)$ time). Finally, we set $F_l(\{v\}) = \&B_i$ and $F_r(\{v\}) = \&B'_j$.

The algorithm is summarized in Figure 4.2. When the addition procedure terminates we reset the counters of all blocks adjacent to v to 0.

Input: A representation of the current graph G and a list of neighbors in G of a new vertex v.
Output: A representation of $G \cup v$ or a *False* value indicating that $G \cup v$ is not a proper interval graph.

Find all segments of blocks which are adjacent to v, and let their number be s.
If $s \geq 3$ **then** return *False*.
Else if $s = 1$ **then** apply the DHH algorithm.
Else /* $s = 2$ */
Check that exactly one extreme block in each segment is an end-block to which v is fully adjacent, and the two extreme blocks in each segment are adjacent. Otherwise, return *False*.
Check if the two segments are in distinct contigs. Otherwise, return *False*.
Update the representation of the graph as described above.

Figure 4.2: A vertex-incremental algorithm for proper interval graph representation.

4.2.5 An Edge-Incremental Algorithm

In this section we show how to handle the addition of a new edge (u, v) in constant time. We characterize the cases for which $G' = G \cup \{(u, v)\}$ is a proper interval graph and show how to efficiently detect them, and how to update our representation of the graph.

Lemma 4.2.11 *If u and v are in distinct connected components in G, then G' is a proper interval graph if and only if u and v are end-vertices in a straight enumeration of G.*

Proof: To prove the 'only if' part let us examine the graph $H = G' \setminus \{u\} = G \setminus \{u\}$. H is a proper interval graph as it is an induced subgraph of G. If G' is also a proper interval graph, then by Lemma 4.2.10(3) v must be an end-vertex in a straight enumeration of G, since u is not adjacent to any other vertex in the component containing v. The same argument applies to u.

To prove the 'if' part we give a straight enumeration of the new connected component containing u and v in G'. Denote by C and D the components containing u and v, respectively. Let $B_1 < \ldots < B_k$ be a contig of C, such that $u \in B_k$. Let $B'_1 < \ldots < B'_l$ be a contig of D, such that $v \in B'_1$. Then $B_1 < \ldots < B_{k-1} < B_k \setminus \{u\} < \{u\} < \{v\} < B'_1 \setminus \{v\} < B'_2 < \ldots < B'_l$ is the required straight enumeration. ■

By the previous lemma if u and v are in distinct components in G, and G' is a proper interval graph, then they must reside in end-blocks of distinct contigs. We can check that in $O(1)$ time. In case u and v are end-vertices of two distinct contigs, we update our internal data structure as follows:

- Block enumeration: Given in the proof of Lemma 4.2.11.
- End pointers: We set $E(B_1) = E(B'_1)$ and $E(B'_l) = E(B_k)$. We then nullify the end-pointers of B_k and B'_1.
- Notation: Let $B_0 = \emptyset$ and $B'_{l+1} = \emptyset$. Let $B_k = B_k \setminus \{u\}$ and $B'_1 = B'_1 \setminus \{v\}$. If $B_k \neq \emptyset$ let $x = k$, and otherwise, let $x = k - 1$. If $B'_1 \neq \emptyset$ let $y = 1$, and otherwise, let $y = 2$.
- Near pointers: We set $N_r(\{u\}) = \&\{v\}, N_l(\{u\}) = \&B_x, N_l(\{v\}) = \&\{u\}$, and $N_r(\{v\}) = \&B'_y$. We also update $N_r(B_x) = \&\{u\}$ and $N_l(B'_y) = \&\{v\}$.
- Far pointers: We set $F_l(\{u\}) = F_l(B_k)$ and $F_r(\{v\}) = F_r(B'_1)$. We exchange the right self-pointer of B_k with the right self-pointer of $\{u\}$, and the left self-pointer of B'_1 with the left self-pointer of $\{v\}$. Finally, we set $F_r(\{u\}) = \&\{v\}$ and $F_l(\{v\}) = \&\{u\}$.

It remains to handle the case where u and v are in the same connected component C in G. If $N(u) = N(v)$ then by the umbrella property C contains only three blocks which are merged into a single block in G'. In this case G' is a proper interval graph and updates to the internal data structure are trivial. The remaining case is analyzed in the following lemma.

Lemma 4.2.12 *Let $B_1 < \ldots < B_k$ be a contig of C, such that $u \in B_i$ and $v \in B_j$ for some $1 \leq i < j \leq k$. Assume that $N(u) \neq N(v)$. Then G' is a proper interval graph if and only if $F_r(B_i) = B_{j-1}$ and $F_l(B_j) = B_{i+1}$ in G.*

Proof: Let G' be a proper interval graph. Since B_i and B_j are non-adjacent, $F_r(B_i) \leq B_{j-1}$ and $F_l(B_j) \geq B_{i+1}$. Suppose to the contrary that $F_r(B_i) < B_{j-1}$. Let z be a vertex of B_{j-1}. If in addition $F_l(B_j) = B_{i+1}$, then by the umbrella property $N[v] \supset N[z]$ (this is a strict containment). As v and z are in distinct blocks, there exists a vertex $b \in N[v] \setminus N[z]$. But then, $\{v, b, z, u\}$ induce a claw in G', a contradiction. Hence, $F_l(B_j) > B_{i+1}$ and ,therefore, $F_r(B_{i+1}) < B_j$. Let $x \in B_{i+1}$ and let $y \in F_r(B_{i+1})$. As u and x are in distinct blocks, either $(u, y) \notin E(G)$, or there exists a vertex $a \in N[u] \setminus N[x]$ (or both). In the first case, v, u, x, y, and the vertices on a shortest path from y to v, induce a chordless cycle in G'. In the second case $\{u, a, x, v\}$ induce a claw in G'. Thus, in both cases we arrive at a contradiction. By a symmetric argument we conclude that $F_l(B_j) = B_{i+1}$.

To prove the 'if' part we provide a straight enumeration of $C \cup \{(u, v)\}$. If $B_i = \{u\}$, $F_r(B_{j-1}) = F_r(B_j)$ and $F_l(B_{j-1}) = B_i$ (i.e., $N[v] = N[B_{j-1}]$ in G'), we move v from B_j to B_{j-1}. Similarly, if B_j contained only v, $F_l(B_{i+1}) = F_l(B_i)$ and $F_r(B_{i+1}) = B_j$ (i.e., $N[u] = N[B_{i+1}]$ in G'), we move u from B_i to B_{i+1}. If u was not moved and B_i contained vertices other than u, we split B_i into $B_i = B_i \setminus \{u\}, \{u\}$ in this order. If v was not moved and B_j contained vertices other than v, we split B_j into $\{v\}, B_j = B_j \setminus \{v\}$ in this order. It is easy to see that the result is a straight enumeration of $C \cup \{(u, v)\}$. ■

If u and v are neither end-vertices of distinct contigs, nor end-vertices of a three-block contig, then assuming that G' is a proper interval graph, the condition of Lemma 4.2.12 must hold. We can verify that in constant time, and if this is the case, change our data structure so as to reflect the new straight enumeration of blocks given in the proof of Lemma 4.2.12. We describe below the updates to our data structure.

- Block enumeration: Given in the proof of Lemma 4.2.12.
- Near pointers: If u was moved into B_{i+1} then no change is necessary with respect to u. Otherwise, if $|B_i| > 1$ then u forms a new block and we set $N_l(\{u\}) = \&B_i, N_r(B_i) = \&\{u\}, N_r(\{u\}) = \&B_{i+1}$, and $N_l(B_{i+1}) = \&\{u\}$. Analogous updates are made with respect to v.
- Far pointers: If u was moved into B_{i+1}, then no change is necessary with respect to u. Otherwise, if $|B_i| > 1$ we exchange the right self-pointer of B_i with

the right self-pointer of (the new block) $\{u\}$. Let B denote the block containing v in G'. We also set $F_l(\{u\}) = F_l(B_i)$ and $F_r(\{u\}) = \&B$. Analogous updates are made with respect to v.

The following theorem summarizes the results of Sections 4.2.4 and 4.2.5.

Theorem 4.2.13 *There is an optimal incremental algorithm for proper interval graph representation which handles an addition operation involving d edges in $O(d)$ time.*

4.2.6 A Fully Dynamic Algorithm

In this section we give a fully dynamic algorithm for recognizing and representing proper interval graphs. The algorithm performs a modification involving d edges in $O(d + \log n)$ time. It supports all types of operations: Adding a vertex, adding an edge, deleting a vertex, and deleting an edge. It is based on the incremental algorithm. The main difficulty in extending the incremental algorithm to handle all types of operations, is updating the end pointers of blocks when both insertions and deletions are allowed. To bypass this problem we (implicitly) keep the identity of each block as an end/inner block, but do not keep end pointers at all. Instead, we maintain the connected components of G, and use this information in our algorithm. In the next section we provide a fully dynamic algorithm for maintaining the connected components of a proper interval graph. This algorithm handles a modification request involving d edges in $O(d + \log n)$ time, and determines for any two blocks whether they are in the same connected component in $O(\log n)$ time. We now describe how each operation is handled by the fully dynamic proper interval graph representation algorithm.

The Addition of a Vertex

This operation is handled in essentially the same way as above. However, in order to check if the end-blocks of two distinct segments are in distinct components, we query our data structure of connected components (in $O(\log n)$ time), rather than checking if the end pointers of these blocks do not point to each other.

The Addition of an Edge

Again, handling this operation is similar to its handling by the incremental algorithm, with the exception that in order to check if the endpoints of an edge are in distinct components, we query our data structure of connected components (in $O(\log n)$ time).

The Deletion of a Vertex

We next show how to update the contigs of G after deleting a vertex v of degree d. Note, that in this case G' is an induced subgraph of G and, thus, a proper interval graph.

Denote by X the block containing v. If X contains vertices other than v then the data structure is simply updated by deleting v. Hence, we concentrate on the case that $X = \{v\}$. In time $O(d)$ we can find the segment of blocks which includes X and all its neighbors. Let the contig containing X be $B_1 < \ldots < B_k$, and let the blocks of the segment be $B_i < \ldots < B_j$, where $X = B_l$ for some $1 \leq i \leq l \leq j \leq k$. The following updates should be performed:

- Block enumeration: If $1 < i < l$, we check whether B_i can be merged with B_{i-1}. If $F_l(B_i) = F_l(B_{i-1})$, $F_r(B_i) = B_l$, and $F_r(B_{i-1}) = B_{l-1}$, we merge these blocks by moving all vertices from B_i to B_{i-1} (in $O(d)$ time) and deleting B_i. If $l < j < k$ we deal similarly with B_j and B_{j+1}.

  Finally, we delete B_l. If $1 < l < k$ and B_{l-1}, B_{l+1} are non-adjacent, then by the umbrella property they are no longer in the same connected component, and the contig should be split into two contigs, one ending at B_{l-1} and the other beginning at B_{l+1}.

- Near pointers: Let $B_0 = \emptyset, B_{k+1} = \emptyset$. If B_i and B_{i-1} were merged, we update $N_r(B_{i-1}) = \&B_{i+1}$ and $N_l(B_{i+1}) = \&B_{i-1}$. Similar updates are made with respect to B_{j-1} and B_{j+1} in case B_j and B_{j+1} were merged. If the contig is split, we nullify $N_r(B_{l-1})$ and $N_l(B_{l+1})$. Otherwise, we update $N_r(B_{l-1}) = \&B_{l+1}$ and $N_l(B_{l+1}) = \&B_{l-1}$.

- Far pointers: If B_i and B_{i-1} were merged, we exchange the right self-pointer of (the previous) B_i with the right self-pointer of B_{i-1}. Similar changes should be

made with respect to B_j and B_{j+1}. We also set all right far pointers, previously pointing to B_l, to $\&B_{l-1}$; and all left far pointers, previously pointing to B_l, to $\&B_{l+1}$ (in $O(d)$ time).

Note, that these updates take $O(d)$ time and require no knowledge about the connected components of G. Since we are dealing with an hereditary property, the trivial lower bound for handling a vertex deletion is $O(1)$ time, so it is not clear whether the above algorithm is optimal.

The Deletion of an Edge

Let (u, v) be an edge of G to be deleted. Let C be the connected component of G containing u and v. Let B_i and B_j be the blocks containing u and v, respectively, in a contig $B_1 < \ldots < B_k$ of C. If $i = j = k = 1$ then B_1 is split into $\{u\}, B_1 \setminus \{u, v\}$ and $\{v\}$, in this order, resulting in a straight enumeration of G'. Updates are trivial in this case. Henceforth we assume that $k > 1$. We first observe that $i \neq j$, i.e., $N[u] \neq N[v]$:

Lemma 4.2.14 *If $N[u] = N[v]$ then G' is a proper interval graph if and only if C is a clique.*

Proof: To prove the 'only if' part, we first show that every vertex $x \in C \setminus \{u, v\}$ is adjacent to both u and v. Suppose to the contrary that there exists a vertex $x \in C \setminus \{u, v\}$ which is not adjacent to u. Let $x = x_1, \ldots, x_k = u$ be a shortest path in C from x to u, where $k > 2$. By definition, x_{k-1} is the first vertex on the path which is adjacent to u (and, therefore, also to v). Hence, $\{x_{k-2}, x_{k-1}, u, v\}$ induce a claw in G', a contradiction. Finally, if a and b are two non-adjacent vertices in $C \setminus \{u, v\}$ then $\{a, u, b, v\}$ induce a chordless cycle in G', a contradiction.

To prove the 'if' part, notice that since C is a clique, it is a block in G. Therefore, $\{u\}, C \setminus \{u, v\}, \{v\}$ is a straight enumeration of $C \setminus \{(u, v)\}$. ■

Since by our assumptions $k > 1$, we conclude that $N[u] \neq N[v]$ and, therefore, $N(u) \neq N(v)$. Without loss of generality, $i < j$. The updates to the straight enumeration of $C \setminus \{(u, v)\}$ are derived from the following lemma.

Lemma 4.2.15 *Let $B_1 < \ldots < B_k$ be a contig of C, such that $u \in B_i$ and $v \in B_j$ for some $1 \leq i < j \leq k$. Then G' is a proper interval graph if and only if $F_r(B_i) = B_j$ and $F_l(B_j) = B_i$ in G.*

Proof: Suppose that G' is a proper interval graph. We prove that $F_r(B_i) = B_j$. A symmetric argument shows that $F_l(B_j) = B_i$. Since B_i and B_j are adjacent in G, $F_r(B_i) \geq B_j$. Suppose to the contrary that $F_r(B_i) > B_j$. Let $x \in F_r(B_i)$. By the umbrella property $(x, v) \in E(G)$. Since x and v are in distinct blocks in G, either there exists a vertex $a \in N[v] \setminus N[x]$ or there exists a vertex $b \in N[x] \setminus N[v]$ (or both). In the first case, by the umbrella property $(a, u) \in E(G)$. Therefore, $\{u, x, v, a\}$ induce a chordless cycle in G'. In the second case, $\{x, b, u, v\}$ induce a claw in G'. Hence in both cases we arrive at a contradiction.

To prove the converse implication we give a straight enumeration of $C \setminus \{(u, v)\}$. If $B_i = \{u\}$, $B_j = \{v\}$ and $j = i + 1$, we have to split the contig into two contigs, one ending at B_i and the other beginning at B_j. If $B_j = \{v\}$, $F_l(B_{i-1}) = F_l(B_i)$ and $F_r(B_{i-1}) = B_{j-1}$ (i.e., $N[u] = N[B_{i-1}]$ in G'), we move u into B_{i-1}. If B_i contained only u, $F_r(B_{j+1}) = F_r(B_j)$ and $F_l(B_{j+1}) = B_{i+1}$ (i.e., $N[v] = N[B_{j+1}]$ in G'), we move v into B_{j+1}. If u was not moved and B_i contains vertices other than u, then B_i is split into $\{u\}, B_i = B_i \setminus \{u\}$ in this order. If v was not moved and B_j contains vertices other than v, then B_j is split into $B_j = B_j \setminus \{v\}, \{v\}$ in this order. The result is a straight enumeration of $C \setminus \{(u, v)\}$. ■

If the conditions of Lemma 4.2.15 are fulfilled, then the following updates should be made:

- Block enumeration: Given in the proof of Lemma 4.2.15.
- Near pointers: Let $B_0 = \emptyset, B_{k+1} = \emptyset$. If $B_i = \{u\}$, $B_j = \{v\}$ and $j = i + 1$, we nullify $N_r(u)$. If B_i was split, we set $N_r(\{u\}) = \&B_i$, $N_l(B_i) = \&\{u\}$, $N_l(\{u\}) = \&B_{i-1}$, and $N_r(B_{i-1}) = \&\{u\}$. If B_i contained only u, and u was moved into B_{i-1}, we update $N_r(B_{i-1}) = \&B_{i+1}$ and $N_l(B_{i+1}) = \&B_{i-1}$. Analogous updates are made with respect to v.
- Far pointers: If $B_i = \{u\}$, $B_j = \{v\}$ and $j = i + 1$, we nullify $F_r(u)$. If B_i was split, we exchange the left self-pointer of B_i with the left self-pointer of $\{u\}$. We also set $F_l(\{u\}) = F_l(B_i)$ and $F_r(\{u\}) = \&B_y$, where $y = j$ in case v is no

longer in B_j (that is, v was moved into B_{j+1}, or B_j was split) and, otherwise, $y = j - 1$. If B_i contained only u, and u was moved into B_{i-1}, we exchange the right self-pointer of B_i with the right self-pointers of B_{i-1}, and delete B_i. Analogous updates are made with respect to v.

Note that these updates take $O(1)$ time and require no knowledge about the connected components of G. The following theorem summarizes our results.

Theorem 4.2.16 *There is a decremental algorithm for proper interval graph representation which handles a deletion operation involving d edges in $O(d)$ time.*

4.2.7 Maintaining the Connected Components

In this section we describe a fully dynamic algorithm for maintaining the connected components of a proper interval graph G in $O(d+\log n)$ time per operation involving d edges. In Section 4.2.8 we shall establish a lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per edge operation (in the cell probe model of computation with word-size b) for this problem.

The algorithm receives as input a series of operations to be performed on a graph, which can be any of the following: Adding a vertex, adding an edge, deleting a vertex, deleting an edge, or querying if two vertices are in the same connected component. It operates on the blocks of the graph rather than on its vertices. The algorithm depends on a data structure which includes the blocks and the contigs of the graph. Hence, it interacts with the proper interval graph representation algorithm. In response to an update request, changes are made to the representation of the graph based on the structure of its connected components prior to the update. Only then are the connected components of the graph updated. We provide a data structure of connected components which performs each operation in $O(\log n)$ time.

Let us denote by $B(G)$ the *block graph* of G, that is, a graph in which each vertex corresponds to a block of G and two vertices are adjacent if and only if their corresponding blocks are adjacent in G. The algorithm maintains a spanning forest F of $B(G)$. When a modification in the graph occurs, the spanning forest is updated accordingly. In order to decide if two blocks are in the same connected component, the algorithm checks if they belong to the same tree in F.

The key idea is to design F so that it can be efficiently updated upon a modification in G. We define the edges of F as follows: For every two vertices u and v in $B(G)$, $(u, v) \in E(F)$ if and only if their corresponding blocks are consecutive in a contig of G (or equivalently, if the near pointers of these blocks point to each other in our representation). Consequently, each tree in F is a path representing a contig. The crucial observation about F is that an addition or a deletion of a vertex or an edge in G induces a constant number of modifications to the vertices and edges of F. This can be seen by noting that each modification of G induces a constant number of updates to near pointers in our representation of G.

It remains to describe a data structure for storing F that allows to query for each vertex to which path it belongs, and that enables adding a vertex, deleting a vertex, splitting a path upon a deletion of an edge in F, and joining two paths upon an addition of an edge to F. If we store the vertices of each path of F in a balanced tree, then each of these operations can be supported in $O(\log n)$ time (cf. [38]).

We are now ready to state our main result:

Theorem 4.2.17 *The fully dynamic proper interval graph representation problem is solvable in $O(d + \log n)$ worst-case time per modification involving d edges.*

We note that the performance of our representation algorithm depends on the performance of a data structure of connected components for a graph, which is a union of disjoint paths, that supports the following operations: Joining two paths, splitting a path, and querying if two vertices belong to the same path. Given such a data structure which supports each operation in $O(f(n))$ time, for some function f, our representation algorithm can be implemented to run in $O(d + f(n))$ time per modification involving d edges.

4.2.8 The Lower Bounds

In this section we prove a lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per edge operation for fully dynamic proper interval graph recognition in the cell probe model of computation with word-size b (see [196] for details about the model). Furthermore, we prove the same lower bound also for the problem of fully dynamic connectivity maintenance of a proper interval graph.

Fredman and Saks [66] have shown a lower bound of $\Omega(\log n/(\log\log n + \log b))$ amortized time per operation for the following *parity prefix sum* (PPS) problem: Given an array of integers $A[1], \ldots, A[n]$ with initial value zero, execute an arbitrary sequence of Add(t) and Sum(t) operations, where an Add(t) increases $A[t]$ by 1, and Sum(t) returns $(\sum_{i=1}^{t} A[i])$ *mod* 2. Fredman and Henzinger [100] and independently Miltersen et al. [145] have proven that the same lower bound applies to the problem of maintaining connectivity in general graphs, by reduction from PPS. We use similar constructions in our lower bound proofs.

Theorem 4.2.18 *There is a fully dynamic algorithm for proper interval graph recognition which takes* $\Omega(\log n/(\log\log n + \log b))$ *amortized time per edge operation in the cell probe model of computation with word-size* b.

Proof: Given an instance of the PPS problem (i.e., a sequence of Add and Sum operations) we construct an instance of the dynamic proper interval graph recognition problem, such that each Add operation corresponds to $O(1)$ edge modifications in the dynamic proper interval graph instance, and each Sum query corresponds to a constant number of temporary edge modifications to the dynamic graph: The answer to the query is determined by checking if the modified graph is proper interval and the modifications are reversed. Thus, a sequence of m operations for the PPS problem translates to $O(m)$ edge modifications, and the lower bound for the PPS problem implies that there exists a sequence of m operations for the dynamic proper interval recognition problem that takes $\Omega(m \log n/(\log\log n + \log b))$ time in the cell probe model of computation with word-size b.

Given an instance of the PPS problem, define $S_t = (\sum_{i=1}^{t} A[i])$ *mod* 2 for $1 \leq t \leq n$. The reduction is as follows: We construct a graph $G = (V, E)$ with $2(n+1)$ vertices labeled $0, \overline{0}, 1, \overline{1}, \ldots, n, \overline{n}$. For every $1 \leq t \leq n$ we add two edges depending on S_t. If $S_t = 0$, we add the edges $\{(t-1, t), (\overline{t-1}, \overline{t})\}$. Otherwise, we add the edges $\{(t-1, \overline{t}), (\overline{t-1}, t)\}$. We define a partial order on the vertices of G as follows: $0, \overline{0} < 1, \overline{1} < \ldots < n, \overline{n}$.

To answer a Sum(t) query ($1 \leq t \leq n$) we act according to one of the following cases:

1. $t = 1$: If $(\overline{0}, 1) \in E$ we output 1, otherwise we output 0.
2. $t = 2$: If $(\overline{0}, t'), (t', 2) \in E$ for $t' \in \{1, \overline{1}\}$ we output 1, otherwise we output 0.

3. $t \geq 3$: If $t < n$ let $t' > t$ be a vertex adjacent to t and define $H \equiv G \setminus \{(t,t')\} \cup \{(\overline{0},t)\}$. If $t = n$, define $H \equiv G \cup \{(\overline{0},t)\}$. If Sum($t$)= 1 then this modification forms a chordless cycle (in H). Otherwise, the new graph is a union of two disjoint paths. Hence, H is a proper interval graph if and only if Sum(t)= 0. Correspondingly, if H is a proper interval graph we output 0, otherwise we output 1. After producing the reply, the modification is undone and G is restored.

To perform an Add(t) operation we do the following:

1. Let $a, a' \in \{t-1, \overline{t-1}\}$ be the vertices adjacent to $t, \overline{t}$, respectively.

2. Delete from G the edges (a,t) and $(a',\overline{t})$.

3. Add to G the edges $(a,\overline{t})$ and (a',t).

This completes the reduction. ■

Note that since the key to the reduction above is the ability to detect cycles, similar arguments can be used to show that the same lower bound applies also to recognizing other graph classes, e.g., interval graphs and chordal graphs.

Theorem 4.2.19 *There is a lower bound of* $\Omega(\log n/(\log\log n + \log b))$ *amortized time per edge operation in the cell probe model of computation with word-size* b *for fully dynamic connectivity maintenance in a proper interval graph.*

Proof: We use the same reduction as in the proof of Theorem 4.2.18, with the exception that in order to answer a $Sum(t)$ query we check whether vertices $\overline{0}$ and t are connected. If the answer is positive we output 1, otherwise we output 0. The reduction is valid since the graph G, which is constructed in the reduction, is a union of two disjoint paths and, therefore, is a proper interval graph. ■

Note that both theorems above apply even if the only modifications allowed in the graph are edge insertions and edge deletions.

4.3 Cograph Recognition

4.3.1 Introduction

A very useful representation of a graph is its modular decomposition tree (defined below). The problem of generating the modular decomposition tree of a graph was studied by many authors and several linear-time algorithms were developed for it [140, 42, 44]. For the problem of dynamically maintaining the modular decomposition tree of a graph only two partial results are known. Muller and Spinrad [147] have given a vertex-incremental algorithm for modular decomposition, which handles each vertex insertion in $O(n)$ time. Corneil, Perl and Stewart [41] have given an optimal vertex-incremental algorithm for the recognition and modular decomposition of cographs, which handles the insertion of a vertex of degree d in $O(d)$ time.

Here we give the first fully dynamic algorithm for maintaining the modular decomposition tree of a cograph. Our algorithm builds on ideas and observations made in the pioneering work on cographs by Corneil, Perl and Stewart [41]. For handling edge operations the algorithm exploits the restricted structure of a cograph that remains such after an edge modification. Vertex operations are handled using ideas from [41]. Our algorithm works in $O(d)$ time per operation involving d edges. Based on this algorithm we develop fully dynamic algorithms for the recognition of cographs, threshold graphs and trivially perfect graphs. All these algorithms handle a modification involving d edges in $O(d)$ time. This is optimal with respect to all operations, with the possible exception of vertex deletion.

This part is organized as follows: Section 4.3.2 contains definitions and terminology. Section 4.3.3 describes some observations on modular decompositions of graphs and their complements. Section 4.3.4 presents the fully dynamic algorithm for recognizing cographs and maintaining their modular decomposition tree. Sections 4.3.5 and 4.3.6 describe the recognition algorithms for threshold graphs and trivially perfect graphs.

4.3.2 Preliminaries

Let G be a graph. The *complement-connected components* of G are the connected components of its complement graph $\overline{G}$. A *module* M in G is a set of vertices $M \subseteq V$ such that every vertex in $V \setminus M$ is either adjacent to every vertex in M, or non-adjacent to every vertex in M. A module M is called *trivial* if $M = V$ or M contains a single vertex. M is called *connected* if G_M is a connected subgraph. M is called *complement-connected* if $\overline{G_M}$ is a connected graph. We shall often refer to a module as though it was the subgraph induced by its vertices. (For example, we shall talk about the connected components of a module.) A disconnected module is called *parallel*. A complement-disconnected module is called *series*. A module which is both connected and complement-connected is called a *neighborhood* module. Note that every module is exactly one of the three types: Series, parallel or neighborhood.

A module M is *strong* if for any module N with $N \cap M \neq \emptyset$, we have $N \subseteq M$ or $M \subseteq N$. A strong module M is a *maximal submodule* of a module $N \supset M$, if no strong submodule of N properly contains M and is properly contained in N. It has been shown (cf. [178]) that every vertex of a non-trivial module M belongs to a unique maximal submodule of M. Clearly, the maximal submodules of a parallel module are its connected components, and the maximal submodules of a series module are its complement-connected components. Hence, the structure of the modules of a graph G can be captured by the following *modular decomposition tree* T_G: The nodes of T_G correspond to strong modules of G. The root node is V, and the set of leaves of T_G consists of all the vertices of G. The children of every internal node M of T_G are the maximal submodules of M. Each internal node in T_G is labeled 'series', 'parallel', or 'neighborhood', depending on the type of its corresponding module. Note that the modular decomposition tree of a given graph is unique.

In the sequel we denote the modular decomposition tree of a graph G by T_G. We refer to a node M of T_G by the set of vertices it represents, that is, the set of vertices in the leaves of the subtree rooted at M. For two vertices $u, v \in V$, we denote by M_{uv} the least common ancestor of $\{u\}$ and $\{v\}$ in T_G.

Let Π be some graph class. Π is called *complement-invariant* if $G \in \Pi$ implies $\overline{G} \in \Pi$. Examples for complement-invariant classes include perfect graphs, cographs, split graphs, threshold graphs and permutation graphs.

4.3.3 A Reduction

We say that a dynamic algorithm *Alg* for recognizing some graph property is *based on modular decomposition* if: (1) *Alg* maintains the modular decomposition tree of the dynamic graph; and (2) the only operations that *Alg* makes are updates to the tree, or queries regarding the tree.

Observation 4.3.1 *The modular decomposition trees of a graph and its complement are identical up to exchanging the labels 'series' and 'parallel'.*

Note that this observation relates to the modular decomposition tree structure only. If the tree contains only parallel and series nodes, this structure suffices to reconstruct the graph. However, if there are also neighborhood modules then additional information on the relations between the maximal submodules of each neighborhood module is needed.

Theorem 4.3.2 *Let* Π *be a complement-invariant graph property. Let Alg be a dynamic algorithm for* Π *recognition, which supports either edge insertions only or edge deletions only, and is based on modular decomposition. Then Alg can be extended to support both operations with the same time complexity.*

Proof: Suppose that *Alg* is an edge-incremental algorithm. The proof for the case that *Alg* is an edge-decremental algorithm is analogous. Let $G = (V, E)$ be the current graph. In order to delete an edge $(u, v) \in E$ we perform an insert operation on $\overline{G}$, by treating each parallel node in T_G as a series node and vice-versa. By Observation 4.3.1, the modular decomposition tree of $\overline{G}$ is identical to T_G up to exchanging the labels 'series' and 'parallel'. Since $\overline{\overline{G} \cup \{(u, v)\}} = G \setminus \{(u, v)\}$, the algorithm performs the update successfully if and only if $G \setminus \{(u, v)\} \in \Pi$. ■

4.3.4 Cographs

In this section we give a fully dynamic algorithm for recognizing cographs and maintaining their modular decomposition tree. The algorithm works in $O(d)$ time per operation involving d edges. It is based on the following fundamental characterization of cographs:

Theorem 4.3.3 ([40]) *A graph is a cograph if and only if its modular decomposition tree contains only parallel and series nodes.*

Another viewpoint on the modular decomposition tree of a cograph is as a method to build the graph: Going recursively up the tree, the subgraph of a parallel node is formed by taking the union of its children's subgraphs. For a series node, all edges between vertices in distinct child modules are added to that graph.

Theorem 4.3.3 implies that a cograph is connected or complement-connected, but not both. It also implies that in a modular decomposition tree of a cograph parallel and series nodes alternate along any path starting from the root. We use these facts often in the sequel. We also rely on the following observation:

Observation 4.3.4 *Let G be a cograph. If u and v are adjacent vertices in G then M_{uv} is a series module in T_G. If u and v are non-adjacent then M_{uv} is a parallel module.*

The Data Structure

Let $G = (V, E)$ be the input graph. We maintain the modular decomposition tree T_G of G as follows: For each vertex of G we keep a pointer to its corresponding leaf-node in T_G. For each node M of T_G we keep its type, which can be 'series' or 'parallel', and its number of children. We also keep pointers from M to its parent and to its children. The parent pointer of the root node points to itself. In detail, each node M has an associated doubly linked list L. Each element of L corresponds to a child N of M, and consists of two pointers, one pointing to N and the other to M. The parent pointer of N points to its corresponding element in L. This data structure allows detaching a child from its parent in constant time. Note that a node in T_G has no explicit record of the vertices it contains as a module.

Initially T_G is calculated in linear time, e.g., using the algorithm of [41]. If G is discovered to contain an induced P_4 then our algorithm outputs *False* and halts. In the description below we assume that G is a cograph.

Adding an Edge

Let (u, v) be the edge to be added, and let $G' = G \cup \{(u, v)\}$. By Observation 4.3.4 M_{uv} is a parallel module. Let C_u and C_v denote the maximal submodules (equiva-

lently, connected components) of M_{uv} which contain u and v, respectively. Without loss of generality, $|C_u| \leq |C_v|$. Our edge-incremental algorithm is based on the following lemma:

Lemma 4.3.5 *G' is a cograph if and only if $|C_u| = 1$ and v is adjacent to every other vertex in C_v.*

Proof:

$\Rightarrow$ Suppose that $|C_u| > 1$. Then C_u contains some vertex a which is adjacent to u, and C_v contains some vertex b which is adjacent to v. Hence, $\{a, u, v, b\}$ induce a P_4 in G', so G' is not a cograph.

Suppose that $w \in C_v \setminus \{v\}$ is not adjacent to v. Let $v, x_1, \ldots, x_k = w$ be a shortest path from v to w in C_v, $k \geq 2$. Then $\{u, v, x_1, x_2\}$ induce a P_4 in G', so G' is not a cograph.

$\Leftarrow$ Suppose that G' contains an induced P_4. Since G is a cograph, an induced P_4 in G' must contain the edge (u, v). Suppose that $\{u, v, x, y\}$ induce a P_4 in G' (not necessarily in this order). One of x and y is therefore adjacent to exactly one of u and v. Without loss of generality, let x be adjacent to exactly one of u and v. Since every vertex in $V \setminus M_{uv}$ is either adjacent to both u and v, or non-adjacent to both of them, we have $x \in M_{uv}$ and, therefore, $x \in C_u$ or $x \in C_v$. If $x \in C_u$, $|C_u| > 1$ and we are done. If $x \in C_v$, then x is adjacent to v and not to u. As $\{u, v, x, y\}$ induce a P_4, y is adjacent either to u only (out of u, v and x), or to x only. In the first case we have $y \in C_u$, implying that $|C_u| > 1$. In the latter case, we conclude that $y \in C_v$. But $(v, y) \notin E(G')$. ■

Note that the lemma implies that $\{v\}$ is a child of C_v in T_G, since otherwise the path from C_v to $\{v\}$ in T_G would contain a parallel node, and v would not be adjacent to all the vertices of C_v.

Let us assume for now that G' is a cograph and that we have already identified M_{uv}, C_u, and C_v. We show below how to update T_G in this case. Later, we shall show how to check the conditions of Lemma 4.3.5 and how to find each of M_{uv}, C_u and C_v.

Let r be the number of children of M_{uv} in T_G. If both C_u and C_v contain a single vertex, we update T_G as follows: If $r = 2$, then the updates depend on the position of M_{uv} in T_G. If M_{uv} lies at the root of T_G, we change its label to 'series'. Otherwise, we connect $\{u\}$ and $\{v\}$ as children of the parent P of M_{uv} (which is a series module), and delete M_{uv}. If $r > 2$, we make $\{u\}$ and $\{v\}$ the children of a new series node $\{u, v\}$, and connect this node as a child of M_{uv}.

Suppose now that $|C_v| > 1$. By Lemma 4.3.5 (since G' is a cograph) $|C_u| = 1$ and v is adjacent to every vertex in $C_v \setminus \{v\}$. We update T_G by first detaching $\{u\}, \{v\}$ and C_v from their parents and forming a new parallel node $K = \{u\} \cup (C_v \setminus \{v\})$. We continue according to one of the following cases:

1. $r > 2$: We add a new series node $\{u\} \cup C_v$ as a child of M_{uv}. We then make $\{v\}$ and K the children of $\{u\} \cup C_v$. This case is illustrated in Figure 4.3.

2. $r = 2$: We connect $\{v\}$ and K to the parent node of M_{uv} (which might be M_{uv} itself if it is the root). We then delete M_{uv}, unless it lies at the root of T_G, in which case we change its label to 'series'.

It remains to describe the subtree of $T_{G'}$ rooted at the new parallel node K. Let $K_1, \ldots, K_l, \{v\}$ be the complement-connected components of C_v. There are two cases to consider:

1. $l > 1$: In this case $C_v \setminus \{v\}$ is necessarily connected. Hence, we need to make $\{u\}$ and $C_v \setminus \{v\}$ the children of K, and connect $K_1, \ldots, K_l$ to $C_v \setminus \{v\}$ as its children (see Figure 4.3). In order to carry out these changes efficiently, we do not introduce a new node $C_v \setminus \{v\}$. Instead, we make C_v a child of K. Since a node has no record of its corresponding vertex set, this alternative update is equivalent to the requested one. Correspondingly, we shall now refer to the former node C_v as $C_v \setminus \{v\}$.

2. $l = 1$: If $K_1 = C_v \setminus \{v\}$ contains a single vertex w, we make $\{u\}$ and $\{w\}$ the children of K. Otherwise, K_1 is complement-connected and, therefore, it is disconnected. Let $J_1, \ldots, J_p$ be the connected components of K_1, $p \geq 2$. Then we need to make $\{u\}$ and $J_1, \ldots, J_p$ the children of K. Instead of introducing the new node K, we make (the former node) K_1 a child of $\{u\} \cup C_v$ (in addition to $\{v\}$), and attach $\{u\}$ as an additional child of K_1. Finally, we delete C_v.

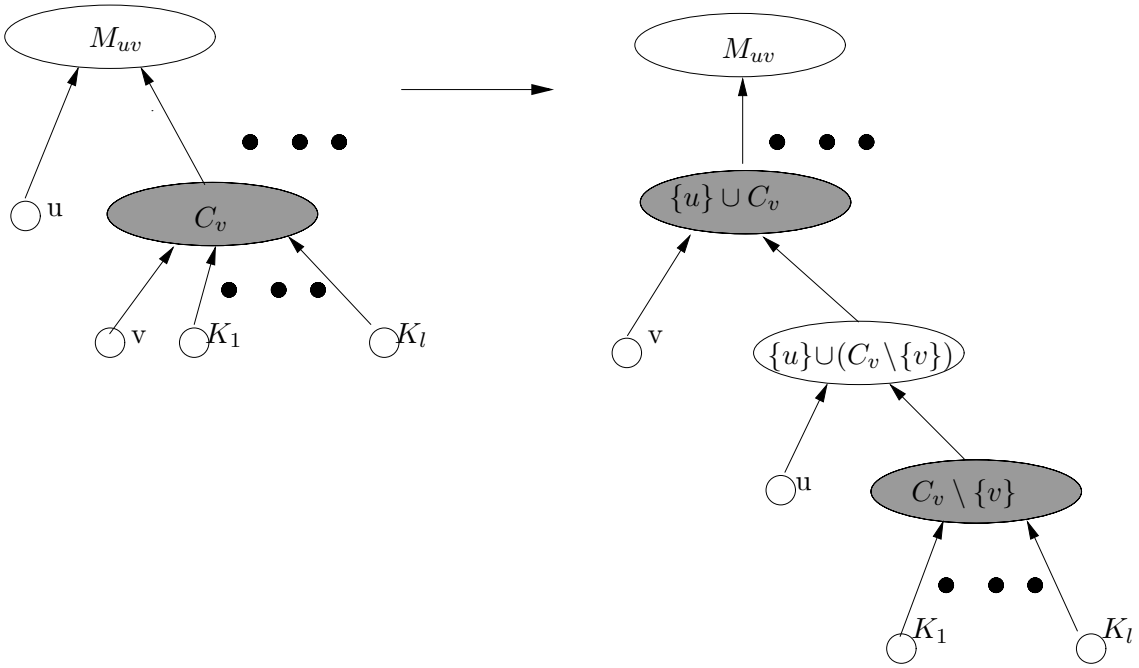


Figure 4.3: The updates to the modular decomposition tree in case M_{uv} and C_v have more than two children each, and $|C_u| = 1$. Series nodes are drawn shaded.

Obviously, all the above updates to T_G can be carried out in constant time. Updating the number of children at each node can be also supported in constant time. It remains to show how to find M_{uv}, C_u and C_v efficiently, and how to verify the conditions of Lemma 4.3.5. In other words, we have to check if one of $\{u\}$ and $\{v\}$ is a child of M_{uv}, and the other is connected to every vertex in its connected component in $G(M_{uv})$. It is straightforward to see that this is the case if and only if M_{uv} is parallel and is either the parent of $\{u\}$ and the grandparent of $\{v\}$, or vice versa (assuming that $|C_u| > 1$ or $|C_v| > 1$). One can determine if such a configuration exists in constant time, by checking if the parent of $\{u\}$ ($\{v\}$) is parallel, and coincides with the grandparent of $\{v\}$ ($\{u\}$). If such a configuration exists, then it immediately identifies M_{uv}, C_u and C_v, and we update T_G accordingly. Otherwise, the algorithm outputs *False* and halts.

The following theorem and corollary summarize our results:

Theorem 4.3.6 *There is an optimal edge-incremental algorithm for recognizing cographs and maintaining their modular decomposition tree, which handles each edge*

insertion in constant time.

Corollary 4.3.7 *There is an optimal edges-only fully dynamic algorithm for recognizing cographs and maintaining their modular decomposition tree, which handles each operation in constant time.*

Vertex Modifications

We shall generalize our algorithm to handle vertex insertions and deletions as well. Supporting vertex insertions is based on the vertex-incremental algorithm for cograph recognition of Corneil et al. [41]. This algorithm handles the insertion of a vertex of degree d in $O(d)$ time, updating the modular decomposition tree accordingly, and can be supported by our data structure with some trivial extensions.

It remains to show how to handle the deletion of a vertex u of degree d from G. Let $G' = G \setminus u$. G' is a cograph as an induced subgraph of G. Hence, we concentrate on updating T_G. Let P be the parent node of $\{u\}$ in T_G. There are four cases to consider:

1. If T_G contains $\{u\}$ only, then $T_{G'}$ is empty.
2. If P has at least three children then $T_{G'}$ is obtained from T_G by deleting $\{u\}$.
3. If P has only two children that are both leaves, $\{u\}$ and $\{v\}$, then $T_{G'}$ is obtained from T_G by deleting $\{u\}$ and replacing P with $\{v\}$.
4. If P has only two children $\{u\}$ and M, where M is an internal node of T_G, then two cases are possible:
   (a) If P lies at the root of T_G, then $T_{G'}$ is the subtree of T_G which is rooted at M.
   (b) Otherwise, let F be the parent of P. Then $T_{G'}$ is formed from T_G by connecting the children of M to F, and deleting $\{u\}, P$ and M.

Proposition 4.3.8 *The deletion of a vertex u of degree d can be handled in $O(d)$ time.*

Proof: All cases except 4b can be handled in constant time. Consider this last case. If P is a series module, then u is adjacent to all the vertices of M, and $T_{G'}$ can be constructed in $O(d)$ time. If P is a parallel module, then instead of deleting M we replace F with M, attaching the former children of F (except P) as children of M. Since u is adjacent to all the vertices of these children modules, this takes $O(d)$ time. ■

We are now ready to state our main result:

Theorem 4.3.9 *There is a fully dynamic algorithm for recognizing cographs and maintaining their modular decomposition tree, which handles insertions and deletions of vertices and edges, and works in $O(d)$ time per modification involving d edges.*

4.3.5 Threshold Graphs

In this section we show a simple extension of our cograph recognition algorithm to dynamically recognize threshold graphs. We use the following characterization of threshold graphs:

Theorem 4.3.10 (cf. [25]) *A graph is a threshold graph if and only if it is both a cograph and a split graph.*

We also use the split recognition algorithm of Ibarra [110], which handles insertions and deletions of edges in constant time. Ibarra's algorithm builds on a characterization of split graphs by their degree sequence [91]. Upon each modification it updates the degree sequence of the dynamic graph. A query is handled by checking if the degree sequence of the graph satisfies the split graph characterization. Notably, this algorithm does not require the graph to be a split graph throughout its modifications. Hence, it can be used to also support vertex modifications in $O(d)$ time per d-degree vertex, by modifying (adding or deleting) the edges incident to the vertex one by one.

Theorem 4.3.11 *There is a fully dynamic algorithm for threshold recognition, which works in $O(d)$ time per operation involving d edges.*

Proof: By Theorem 4.3.9 there exists a fully dynamic algorithm A_1 for cograph recognition, which works in $O(d)$ time per modification involving d edges. Ibarra's work [110] implies a fully dynamic algorithm A_2 for split recognition, achieving the same time bounds. Our algorithm for threshold recognition executes A_1 and A_2 in parallel, and upon a modification outputs *False* and halts if and only if any of these algorithms outputs *False*. ■

4.3.6 Trivially Perfect Graphs

In this section we present a fully dynamic algorithm for trivially perfect graph recognition. Note that this class of graphs is not complement-invariant (C_4 is a counter example). Our algorithm is an extension of the cograph recognition algorithm, which after each operation checks whether the current graph contains an induced C_4. It works in $O(d)$ time per modification involving d edges. Note that trivially perfect graphs are exactly the class of chordal cographs (cf. [25]). Hence, one could use our cograph recognition algorithm in conjunction with Ibarra's chordal recognition algorithm [110] to recognize this class. However, such an algorithm would require $O(n)$ time per edge modification and would not support vertex modifications.

Suppose that $G = (V, E)$ is trivially perfect. If we delete a vertex from G then the resulting graph is clearly trivially perfect. If we add an edge to G and the new graph is a cograph, then it is also a trivially perfect graph. This follows by noting that if an induced C_4 is created, then G must have contained an induced P_4. Hence, it suffices to show how to check for the existence of an induced C_4 after edge deletions and vertex insertions. We assume in the following that the current graph G is trivially perfect, and the modified graph G' is a cograph as, otherwise, the cograph recognition algorithm outputs *False* and we are done.

Adding a Vertex

Let z be a new vertex of degree d to be added, and let $G' = G \cup z$. Clearly, if G' contains an induced C_4, it is of the form $\{a, b, c, z\}$ for some vertices $a, b, c \in V$. If z connects two or more connected components of G then it must be adjacent to every vertex in these components, or else G' would contain an induced P_4. Therefore, in this case G' is trivially perfect. If z is adjacent to all vertices of a single component

then again G' is trivially perfect. One of these cases applies if and only if $\{z\}$ is either a child of a series root module (if G' contains a single connected component), or a grandchild of a parallel root module (if G' contains more than one component). We can check for such configurations in constant time. The remaining case is when z is adjacent to some but not all vertices of a single connected component C of G. We handle this case below.

Lemma 4.3.12 *A cograph contains an induced C_4 if and only if its modular decomposition tree has a series node with at least two non-trivial children.*

Proof: If H is a cograph and $\{a, b, c, d\}$ induce a C_4 in H, then the least common ancestor of $\{a\}$,$\{b\}$,$\{c\}$, and $\{d\}$ in T_H is a series module with at least two non-trivial maximal submodules (one containing a, c and the other containing b, d).

Conversely, if the modular decomposition tree of a cograph H contains a series node with two non-trivial children M_1 and M_2, then any two vertices from M_1 together with any two vertices from M_2 induce a C_4 in H. ■

Lemma 4.3.12 implies that in order to check whether a C_4 is formed in G' it suffices to check if the updates to the modular decomposition tree produce any series node with more than one non-leaf child. In order to verify that efficiently, we introduce at each internal node N of T_G a counter, which stores the number of children of N which are not leaves. These counters can be easily maintained and checked by our dynamic modular decomposition algorithm with no increase to its time complexity. Hence, a d-degree vertex insertion can be supported in $O(d)$ time.

Deleting an Edge

Let $(a, c) \in E$ be an edge to be deleted, and let $G' = G \setminus \{(a, c)\}$. Clearly, any induced C_4 in G' is of the form $\{a, b, c, d\}$ for some vertices $b, d \in V$. By the previous discussion, in order to check whether G' contains an induced C_4, it suffices to check whether the updates to the modular decomposition tree produce any series node with a counter greater than one. By examining the updates to the tree it can be seen that the only series node whose counter might exceed one is M_{ac}, the least common ancestor of $\{a\}$ and $\{c\}$ in T_G. (Using the notation of Section 4.3.4 this happens when $|C_a| = |C_c| = 1$ and $r > 2$.) We provide below a direct proof for that.

Lemma 4.3.13 *If $\{a, b, c, d\}$ induce a C_4 in G' then $N[a] = N[c]$ in G.*

Proof: By our assumption $(a, c) \in E$. Suppose to the contrary that $v \in V$ is adjacent to only one of a and c. Without loss of generality, suppose v is adjacent to a. Hence, v must be adjacent to both b and d or, else,G' contains an induced P_4. But then $\{d, v, b, c\}$ induce a C_4 in G, a contradiction. ■

Lemma 4.3.14 *If $\{a, b, c, d\}$ induce a C_4 in G', and $v \in V$ is adjacent to b or d, then v is adjacent to both a and c in G.*

Proof: By Lemma 4.3.13, $N[a] = N[c]$ in G. Hence, it suffices to prove that v is adjacent to a. Suppose to the contrary that $(v, a) \notin E$. Then $\{d, a, b, v\}$ induce a forbidden subgraph in G (either a P_4 or a C_4), a contradiction. ■

Let M'_{ac} be the least common ancestor of $\{a\}$ and $\{c\}$ in $T_{G'}$. By Observation 4.3.4 M'_{ac} is parallel. If M'_{ac} lies at the root of $T_{G'}$ then G' is a trivially perfect graph, since a and c are in different connected components (and, therefore, cannot be part of the same induced C_4). We assume in the sequel that this is not the case.

Theorem 4.3.15 *Let P be the parent of M'_{ac} in $T_{G'}$. Then G' is a trivially perfect graph if and only if M'_{ac} is the only non-trivial maximal submodule of P.*

Proof: Suppose to the contrary that G' is not a trivially perfect graph. Then there exist two vertices $b, d \in V$ such that $\{a, b, c, d\}$ induce a C_4 in G'. By Lemma 4.3.13, $N(a) = N(c)$ in G'. Hence, M'_{ac} is the parent of both $\{a\}$ and $\{c\}$. We claim that $M'_{ac} = \{a, c\}$. Suppose to the contrary that $v \in M'_{ac} \setminus \{a, c\}$, then v is non-adjacent to a and c (since M'_{ac} is parallel). By Lemma 4.3.14, v is non-adjacent to b and d. However, both a and c are adjacent to b and d. Hence, b must be a vertex of M'_{ac}, implying that a and c are in the same connected component in $G'(M'_{ac})$, a contradiction.

Let M'_{abcd} be the least common ancestor of $M'_{ac}, \{b\}$ and $\{d\}$ in $T_{G'}$. We now prove that $M'_{abcd} = P$. Let S_1 be a maximal submodule of M'_{abcd} that contains M'_{ac}. Since a is adjacent to both b and d, M'_{abcd} must be a series module. Hence, any vertex $v \in S_1 \setminus \{a, c\}$ is adjacent to b or d. By Lemma 4.3.14, v is also adjacent to a and c. Since this holds for all $v \in S_1 \setminus \{a, c\}$, and since M'_{abcd} is a series module,

$S_1 = \{a, c\} = M'_{ac}$, implying that $M'_{abcd} = P$. Finally, since P is a series module, its maximal submodule that contains both b and d is non-trivial and different from M'_{ac}, a contradiction.

Conversely, suppose that P contains a non-trivial maximal submodule $L \neq M'_{ac}$. Since M'_{ac} is a parallel module, P is a series module. Let b and d be two non-adjacent vertices of L. Then $\{a, b, c, d\}$ induce a C_4 in G', a contradiction. ■

Consider the updates to T_G after deleting the edge (a, c). If G' is not trivially perfect then Lemma 4.3.13 implies that M_{ac} was the parent of both $\{a\}$ and $\{c\}$ in T_G. Due to the update a new node $M'_{ac} = \{a, c\}$ is created and attached as a child of M_{ac}. Hence, $P = M_{ac}$ is the parent of M'_{ac} in $T_{G'}$, and in order to determine if G' is trivially perfect, it suffices to check the counter of M_{ac} after the update. We conclude:

Theorem 4.3.16 *There is a fully dynamic algorithm for trivially perfect graph recognition which works in $O(d)$ time per modification involving d edges.*

Chapter 5

Incomplete Directed Perfect Phylogeny

In this chapter we study the problem of reconstructing evolutionary history based on incomplete data. In the perfect phylogeny model for studying evolution every species has an associated vector of characters, each having one of several states. The goal is to reconstruct a tree in which the species are at the leaves and each internal node is associated with a character vector representing an ancestral species, such that the set of all species having the same state in any character induces a connected subtree.

We study the following variant of perfect phylogeny: The input is a species-characters matrix. The characters are binary and directed, i.e., a species can only gain characters. The difference from standard perfect phylogeny is that for some species the state of some characters is unknown. The question is whether one can complete the missing states in a way admitting a perfect phylogeny. The problem arises in classical phylogenetic studies, when some states are missing or undetermined. Quite recently, studies that infer phylogenies using inserted repeat elements in DNA gave rise to the same problem. Extant solutions for it take time $O(n^2m)$ for n species and m characters. We provide a graph theoretic formulation of the problem as a graph sandwich problem, and give near-optimal $\tilde{O}(nm)$-time algorithms for the problem. We also study the problem of finding a single, general solution tree, from which any other solution can be obtained by node-splitting. We provide an algorithm to construct such a tree, or determine that none exists.

Most of the results in this chapter were published in [155] and [156].

5.1 Introduction

When studying evolution, the divergence patterns leading from a single ancestor species to its contemporary descendants are usually modeled by a tree structure, called *phylogenetic tree*, or *phylogeny*. Extant species correspond to the tree leaves, while their common progenitor corresponds to the root. Internal nodes correspond to hypothetical ancestral species, which putatively split up and evolved into distinct species. Tree branches model changes through time of the hypothetical ancestor species. The common case is that one has information regarding the leaves, from which the phylogenetic tree is to be inferred. This task, called *phylogenetic reconstruction* (cf. [63]), was one of the first algorithmic challenges posed by biology, and the computational community has been dealing with problems of this flavor for over three decades (see, e.g., [88]).

The character-based approach to tree reconstruction describes contemporary species by their attributes or *characters*. Each character takes on one of several possible *states*. The input is represented by a matrix $\mathcal{A}$ where a_{ij} is the state of character j in species i, and the i-th row is the *character vector* of species i. The output sought is a hypothesis regarding evolution, i.e., a phylogenetic tree along with the suggested character vectors of the internal nodes. This output must satisfy properties specified by the problem variant.

One important class of phylogenetic reconstruction problems concerns finding a *perfect phylogeny*. The property required from such a phylogeny is that for each possible character state, the set of all nodes that have that state induces a connected subtree. The general perfect phylogeny problem is NP-hard [23, 179]. When considering the number of possible states per character as a parameter, the problem is fixed parameter tractable [4, 116]. For *binary characters*, having only two possible states, perfect phylogeny is linear-time solvable [87].

When no perfect phylogeny is possible, one option is to seek a largest subset of characters which admits a perfect phylogeny. Characters in such a subset are said to be *compatible*. Compatibility problems have been studied extensively (see, e.g., [142]).

Another common optimization approach to phylogenetic reconstruction is the *parsimony* criterion. It calls for a solution with fewest state changes altogether, counting a change whenever the state of a character changes between a species and its ancestor species. This problem is known to be NP-hard [65]. A variant introduced by Camin and Sokal [29] assumes that characters are binary and *directed*, namely, only $0 \rightarrow 1$ changes may occur on any path from the root to a leaf. Denoting by 1 and 0 the presence and absence, respectively, of the character, this means that characters can only be gained throughout evolution. Another related binary variant is Dollo parsimony [47, 158], which assumes that a $0 \rightarrow 1$ change may happen only once, i.e., a character can be gained once, but it can be lost several times. Both of these variants are polynomially solvable (cf. [63]).

In this chapter, we discuss a variant of binary perfect phylogeny which combines the assumptions of both Camin-Sokal parsimony and Dollo parsimony. The setup is as follows: The characters are binary, directed, and can be gained only once. As in perfect phylogeny, the input is a matrix of character vectors, with the difference that some character states are missing. The question is whether one can complete the missing states in a way admitting a perfect phylogeny. We call this problem *Incomplete Directed Perfect phylogeny (IDP)*.

The problem of handling incomplete phylogenetic data arises whenever some of the data are missing. It is also encountered in the context of morphological characters, where for some species it may be impossible to reliably assign a state to a character. The problem of determining whether a set of incomplete *undirected* characters is compatible was shown to be NP-complete, even in the case of binary characters [179]. Indeed, the popular PAUP software package [180] provides an exponential solution to the problem by exhaustively searching the space of missing states.

Quite recently, a novel kind of genomic data has given rise to the same problem: Nikaido et al. [151] use inserted repetitive genomic elements, particularly SINEs (Short Interspersed Nuclear Elements), as a source of evolutionary information. SINEs are short DNA sequences that were copied and randomly reinserted into various genomic loci during evolution. The distinct insertion loci are identifiable by the flanking sequences on both sides of the insertion site (see Figure 5.1). These insertions are assumed to be unique events in evolution, because the odds of having separate insertion events at the very same locus are negligible. Furthermore, a SINE

insertion is assumed to be irreversible, i.e., once a SINE sequence has been inserted somewhere along the genome, it is practically impossible for the exact, complete SINE to leave that specific locus. However, the inserted segment along with its flanking sequences may be lost when a large genomic region, which includes them, is deleted. In that case we do not know whether a SINE insertion had occurred in the missing site prior to its deletion. One can model such data by assigning to each locus a character, whose state is '1' if the SINE occurred in that locus, '0' if the locus is present but does not contain the SINE, and '?' if the locus is missing. The resulting reconstruction problem is precisely Incomplete Directed Perfect phylogeny.

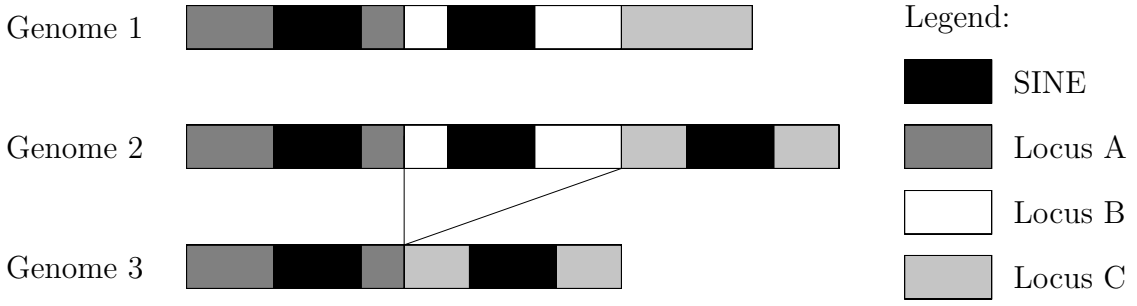


Figure 5.1: SINEs (black boxes) repeat in different loci (different shades of grey) across distinct genomes. A SINE insertion transformed Genome 1 into Genome 2. A deletion of a locus transformed Genome 2 into Genome 3. Given Genomes 1 and 3, we can identify that the SINE on locus C is not present in Genome 1, by its flanking sequence. However, locus B is missing in Genome 3.

The IDP problem becomes polynomial when the characters are directed: Benham et al. [18] studied the compatibility problem on generalized characters. Their work implies an $O(n^2m)$-time algorithm for IDP, where n and m denote the number of species and characters, respectively. Another problem related to IDP is the consensus tree problem [6, 101]. This problem calls for constructing a consensus tree from binary subtrees, and is solvable in polynomial time. One can reduce IDP to the latter problem, but the reduction itself takes $\Omega(n^2m)$ time.

Our approach to the IDP problem is graph theoretic. We first provide several graph and matrix characterizations for solvable instances of binary directed perfect phylogeny. We then reformulate IDP as a *graph sandwich* problem: The input data is recast as two nested graphs, and solving IDP is shown to be equivalent to finding a graph of a particular type "sandwiched" between them. This formulation allows

us to devise efficient algorithms for IDP.

We provide two algorithms for IDP, which we call Algorithms A and B. Algorithm A has two possible implementations: Deterministic and randomized. Its deterministic complexity is $O(nm + k\log^2(n+m))$, for an instance with k 1-entries in the species-characters matrix. The randomized version of Algorithm A takes $O(nm+k\log(l^2/k)+l(\log l)^3\log\log l)$ expected time, where $l = n+m$. Algorithm B is deterministic and takes $O(l^2\log l)$ time. For both algorithms, the improved complexity is obtained by using dynamic data structures for maintaining the connected components of a graph [101, 107, 184]. Since an $\Omega(nm)$ lower bound was shown by Gusfield for directed binary perfect phylogeny [87], our algorithms have near optimal time complexity. We implemented Algorithm A and used it to reanalyze the mammalian evolution data of Nikaido et al. [151].

We also study the issue of multiple solutions for IDP. Often there is more than one phylogeny that is consistent with the data. When the input matrix is complete and has a solution, there is always a tree $\mathcal{T}^*$ that is *general*, i.e., it is a solution, and every other tree consistent with the data can be obtained from $\mathcal{T}^*$ by node splitting. In other words, $\mathcal{T}^*$ describes all the definite information in the data, and ambiguities (nodes with three or more children) can be resolved by additional information. This is not always the case if the data matrix is incomplete: There may or may not be a general solution tree. In that case, using a particular solution and additional information, one can conclude that the data is inconsistent, even though the additional information may be consistent with another solution. It is thus desirable to know if a general solution exists and to generate such a solution if the answer is positive.

We provide answers to both questions. We prove that Algorithm A provides the general solution of a problem instance, if such exists. We also give an algorithm which determines if the solution $\mathcal{T}$ produced by Algorithm A is indeed general. The complexity of the latter algorithm is $O(nm + kd)$, where d denotes the maximum out-degree of $\mathcal{T}$.

The chapter is organized as follows: In Section 5.2 we provide some preliminaries, and formalize the IDP problem. In Section 5.3 we characterize binary matrices admitting a directed perfect phylogeny, and provide the graph sandwich formulation for IDP. In Section 5.4 we present algorithms for IDP. In Section 5.5 we analyze the generality of the solution produced by Algorithm A. Finally, in Section 5.6 we

describe an implementation of our algorithm for IDP, and a study of mammalian evolution using this implementation.

5.2 Preliminaries

We first specify some terminology and notation. We reserve the terms *nodes* and *branches* for trees, and use the terms *vertices* and *edges* for other graphs. Matrices are denoted by an upper-case letter, while their elements are denoted by the corresponding lower-case letter.

Let $\mathcal{T}$ be a rooted tree with leaf set S, where branches are directed from the root towards the leaves. The *out-degree* of a node x in $\mathcal{T}$ is its number of children, and is denoted by $d(x)$. For a node x in $\mathcal{T}$ we denote the leaf set of the subtree rooted at x by $L(x)$. $L(x)$ is called a *clade* of $\mathcal{T}$. For consistency, we consider $\emptyset$ to be a clade of $\mathcal{T}$ as well, and call it the *empty clade*. $S, \emptyset$ and all singletons are called *trivial clades*. We denote by $triv(S)$ the collection of all trivial clades. Two sets are said to be *compatible* if they are either disjoint, or one of them contains the other.

Observation 5.2.1 *(cf. [142]) A collection $\mathcal{S}$ of subsets of a set S is the set of clades of some tree over S if and only if $\mathcal{S}$ contains $triv(S)$ and its subsets are pairwise compatible.*

A tree $\mathcal{T}$ is uniquely characterized by its set of clades. The transformation between a branch-node representation of a tree and a list of its clades is straightforward. Thus, we hereafter identify a tree with the set of its clades. If $\hat{S}$ is a subset of the leaves of $\mathcal{T}$, then the subtree of $\mathcal{T}$ *induced* on $\hat{S}$ is the collection $\{\hat{S} \cap S' : S' \in \mathcal{T}\} \cup \{\hat{S}\}$ (which defines a tree).

Throughout the chapter we denote by $S = \{s_1, \ldots, s_n\}$ the set of all species and by $C = \{c_1, \ldots, c_m\}$ the set of all (binary) characters. For a graph K, we define $C(K) \equiv C \cap V(K)$ and $S(K) \equiv S \cap V(K)$. Let $\mathcal{B}_{n \times m}$ be a binary matrix whose rows correspond to species, each row being the character vector of its corresponding species. That is, $b_{ij} = 1$ if and only if the species s_i has the character c_j. A *phylogenetic tree for* $\mathcal{B}$ is a rooted tree $\mathcal{T}$ with n leaves corresponding to the n species of S, such that each character is associated with a clade S' of $\mathcal{T}$, and the following properties are satisfied:

(1) If c_j is associated with S' then $s_i \in S'$ if and only if $b_{ij} = 1$.
(2) Every non-trivial clade of $\mathcal{T}$ is associated with at least one character.
For a character c, the node x of $\mathcal{T}$ whose clade $L(x)$ is associated with c, is called the *origin* of c with respect to $\mathcal{T}$. Characters associated with $\emptyset$ have no origin.

A $\{0, 1, ?\}$ matrix is called *incomplete*. For convenience, we consider binary matrices as incomplete. Let $\mathcal{A}_{n\times m}$ be an incomplete matrix in which $a_{ij} = 1$ if s_i has c_j, $a_{ij} = 0$ if s_i lacks c_j, and $a_{ij} =?$ if it is not known whether s_i has c_j. For a character c_j and a state $x \in \{0, 1, ?\}$, the *x-set* of c_j in $\mathcal{A}$ is the set of species $\{s_i \in S : a_{ij} = x\}$. c_j is called a *null character* if its 1-set is empty. For subsets $\hat{S} \subseteq S$ and $\hat{C} \subseteq C$, define $\mathcal{A}|_{\hat{S},\hat{C}}$ to be the submatrix of $\mathcal{A}$ induced on $\hat{S} \cup \hat{C}$.

A binary matrix $\mathcal{B}$ is called a *completion* of $\mathcal{A}$ if $a_{ij} \in \{b_{ij}, ?\}$ for all i, j. Thus, a completion replaces all the ?-s in $\mathcal{A}$ by zeroes and ones. If $\mathcal{B}$ has a phylogenetic tree $\mathcal{T}$, we say that $\mathcal{T}$ is a *phylogenetic tree for* $\mathcal{A}$ as well. We also say that $\mathcal{T}$ *explains* $\mathcal{A}$ *via* $\mathcal{B}$, and that $\mathcal{A}$ is *explainable*. An example of these definitions is given in Figure 5.2.

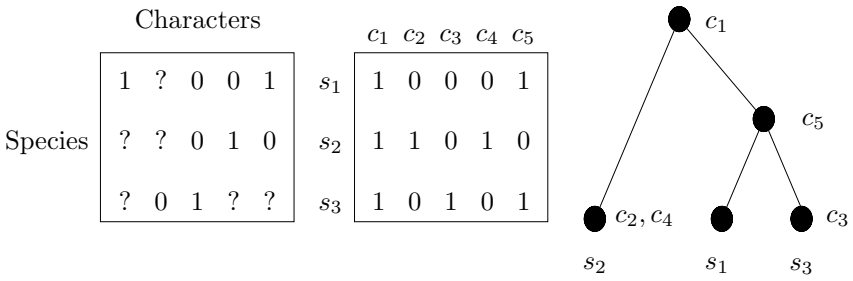


Figure 5.2: Left to right: An incomplete matrix $\mathcal{A}$, a completion $\mathcal{B}$ of $\mathcal{A}$, and a phylogenetic tree that explains $\mathcal{A}$ via $\mathcal{B}$. Each character is written to the right of its origin node.

The following lemma, closely related to Observation 5.2.1, has been proven independently by several authors:

Lemma 5.2.2 (cf. [142]) *A binary matrix $\mathcal{B}$ has a phylogenetic tree if and only if the 1-sets of every two characters are compatible.*

An analogous lemma holds for undirected characters (cf. [87]). In contrast, for incomplete matrices, even if every pair of columns has a phylogenetic tree, the full matrix might not have one. An example of such a matrix was provided in [63] for

incomplete undirected characters. We provide a simpler example for incomplete *directed* characters in Figure 5.3.

	Characters		
Species	1	0	0
	1	1	0
	?	1	1
	0	?	1

Figure 5.3: An incomplete matrix which has no phylogenetic tree although every pair of its columns has one.

We are now ready to state the IDP problem:

Problem 3 (Incomplete Directed Perfect Phylogeny (IDP))
Instance: *An incomplete matrix $\mathcal{A}$.*
Goal: *Find a phylogenetic tree for $\mathcal{A}$, or determine that no such tree exists.*

5.3 Characterizations of Explainable Binary Matrices

5.3.1 Forbidden Subgraph Characterization

Let $\mathcal{B}$ be a species-characters binary matrix of order $n \times m$. Construct the bipartite graph $G(\mathcal{B}) = (S, C, E)$ with $E = \{(s_i, c_j) : b_{ij} = 1\}$. An induced path of length four in $G(\mathcal{B})$ is called a Σ *subgraph* if it starts (and therefore ends) at a vertex corresponding to a species (see Figure 5.4). A bipartite graph with no induced Σ subgraph is called Σ-*free*.

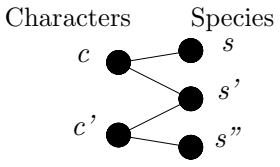


Figure 5.4: The Σ subgraph.

The following theorem restates Lemma 5.2.2 in terms of graph theory.

Theorem 5.3.1 *$\mathcal{B}$ has a phylogenetic tree if and only if $G(\mathcal{B})$ is Σ-free.*

Corollary 5.3.2 *Let $\hat{S} \subseteq S$ and $\hat{C} \subseteq C$ be subsets of the species and characters, respectively. If $\mathcal{A}$ is explainable then so is $\mathcal{A}|_{\hat{S},\hat{C}}$.*

Observation 5.3.3 *Let $\mathcal{A}$ be a matrix explained by a tree $\mathcal{T}$ and let $\hat{S} = L(x)$ be a clade in $\mathcal{T}$, where x is a node of $\mathcal{T}$. Then the submatrix $\mathcal{A}|_{\hat{S},C}$ is explained by the subtree of $\mathcal{T}$ rooted at x.*

For a subset $S' \subseteq S$ of species, we say that a character c is *S'-universal* in $\mathcal{B}$, if its 1-set (in $\mathcal{B}$) contains S'.

Proposition 5.3.4 *If $G(\mathcal{B})$ is connected and Σ-free, then there exists a character which is S-universal in $\mathcal{B}$.*

Proof: Suppose to the contrary that $\mathcal{B}$ has no S-universal character. Consider the collection of all 1-sets of characters in $\mathcal{B}$. Let c be a character whose 1-set is maximal with respect to inclusion in this collection. Let s'' be a species which lacks c. Since $G(\mathcal{B})$ is connected, there exists a path from s'' to c in $G(\mathcal{B})$. Consider a shortest such path P. Since $G(\mathcal{B})$ is bipartite, the length of P is odd. However, P cannot be of length 1, by the choice of s''. Furthermore, if P is of length greater than 3, then its first 5 vertices induce a Σ subgraph, a contradiction. Thus $P = (s'', c', s', c)$ must be of length 3. By maximality of the 1-set of c, it is not contained in the 1-set of c'. Hence, there exists a species s which has the character c but lacks c'. Together with s, the vertices of P induce a Σ subgraph, as depicted in Figure 5.4, a contradiction. ■

Let Ψ be a graph property. In the Ψ *sandwich problem* one is given a vertex set V and a partition of $V \otimes V$ into three disjoint subsets: E_0 - *forbidden edges*, E_1 - *mandatory edges*, and $E_?$ - *optional edges*. The objective is to find a supergraph of (V, E_1) which satisfies Ψ and contains no forbidden edges. Hence, the required graph (V, F) must be "sandwiched" between (V, E_1) and $(V, E_1 \cup E_?)$. The reader is referred to articles [83, 85] for a discussion of various sandwich problems.

For the property "containing no induced Σ subgraph" (a property of bipartite graphs) the sandwich problem is defined as follows:

Problem 4 (Σ-free sandwich)
Instance: *A vertex set $S \cup C$ with $S \cap C = \emptyset$, and a partition $(E_0, E_?, E_1)$ of $S \times C$.*
Goal: *Find a set of edges F such that $F \supseteq E_1$, $F \cap E_0 = \emptyset$, and the graph (S, C, F) is Σ-free, or determine that no such set exists.*

Theorem 5.3.1 motivates considering the IDP problem on input $\mathcal{A}$ as an instance $((S, C), E_0^{\mathcal{A}}, E_?^{\mathcal{A}}, E_1^{\mathcal{A}})$ of the Σ-free sandwich problem. Here, $E_x^{\mathcal{A}} = \{(s_i, c_j) : a_{ij} = x\}$, for $x = 0, ?, 1$. In the sequel, we omit the superscript $\mathcal{A}$ when it is clear from the context.

Proposition 5.3.5 *The Σ-free sandwich problem is equivalent to IDP.*

Note that there is an obvious 1-1 correspondence between completions of $\mathcal{A}$ and possible solutions of the corresponding sandwich instance $((S, C), E_0, E_?, E_1)$. Hence, in the sequel we refer to matrices and their corresponding sandwich instances interchangeably.

5.3.2 Forbidden Submatrix Characterizations

A binary matrix $\mathcal{B}$ is called *good* if it can be decomposed as follows:
(1) Its left $k_1 \geq 0$ columns are all ones.
(2) There exist good matrices $\mathcal{B}_1, \ldots, \mathcal{B}_l$, such that the rest (0 or more) of the columns of $\mathcal{B}$ form the block-structure illustrated in Figure 5.5.

A matrix $\mathcal{A}$ is *canonical* if $\mathcal{A} = [\mathcal{B}, \mathcal{C}]$ where $\mathcal{B}$ is a zero submatrix and $\mathcal{C}$ is good. We say that a matrix $\mathcal{B}$ *avoids* a matrix $\mathcal{X}$, if no submatrix of $\mathcal{B}$ is identical to $\mathcal{X}$.

Theorem 5.3.6 *Let $\mathcal{B}$ be a binary matrix. The following are equivalent:*

1. *$\mathcal{B}$ has a phylogenetic tree.*
2. *$G(\mathcal{B})$ is Σ-free.*
3. *Every matrix obtained by permuting the rows and columns of $\mathcal{B}$ avoids the following matrix:*

$$\mathcal{Z} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

4. *There exists an ordering of the rows and columns of $\mathcal{B}$ which yields a canonical matrix.*

5. *There exists an ordering of the rows and columns of $\mathcal{B}$ so that the resulting matrix avoids the following matrices:*

$$\mathcal{X}_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \mathcal{X}_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, \mathcal{X}_3 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \mathcal{X}_4 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Proof:

1⇔2 Theorem 5.3.1.

2⇔3 Trivial.

1⇒4 Suppose $\mathcal{T}$ is a tree that explains $\mathcal{B}$. Assign to each node of $\mathcal{T}$ an index which equals its position in a preorder visit of $\mathcal{T}$. Sort the characters according to the indices of their origin nodes, letting null characters come first. Sort the species according to the indices of their corresponding leaves in $\mathcal{T}$. The result is a canonical matrix.

4⇒5 It is easy to verify that canonical matrices avoid $\mathcal{X}_1, \ldots, \mathcal{X}_4$.

5⇒3 Suppose to the contrary that $\mathcal{B}$ has an ordering of its rows and columns, so that rows i_1, i_2, i_3 and columns j_1, j_2 of the resulting matrix form the submatrix $\mathcal{Z}$. Consider the permutations $\theta_{row}, \theta_{col}$ of the rows and columns of $\mathcal{B}$, respectively, which yield a matrix avoiding $\mathcal{X}_1, \ldots, \mathcal{X}_4$. In this ordering, row $\theta_{row}(i_1)$ necessarily lies between rows $\theta_{row}(i_2)$ and $\theta_{row}(i_3)$ or, else, the submatrix $\mathcal{X}_4$ occurs. Suppose that $\theta_{row}(i_2) < \theta_{row}(i_3)$ and $\theta_{col}(j_1) < \theta_{col}(j_2)$, then $\mathcal{X}_3$ occurs, a contradiction. The remaining cases are similar. ■

Note that a matrix which avoids $\mathcal{X}_4$ has the consecutive ones property in columns. Gusfield [87, Theorem 3] has proven that a matrix which has an *undirected* perfect phylogeny can be reordered so as to satisfy this property [87, Theorem 3]. In fact, for explainable binary matrices, the reordering used by Gusfield's proof essentially generates a canonical matrix. Note also that Σ-free graphs are bipartite convex as they avoid $\mathcal{X}_1$, $\mathcal{X}_2$ and $\mathcal{X}_3$ (see, e.g., [2]).

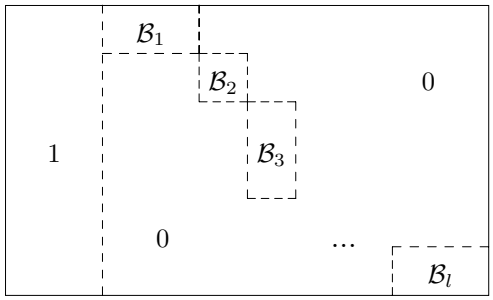


Figure 5.5: Construction of a good matrix. Each $\mathcal{B}_i$ is a good matrix. A canonical matrix is formed from it by appending columns of zeros on the left.

The reader is referred to the article [121] for other problems of permuting matrices to avoid forbidden submatrices.

5.4 Algorithms for Solving IDP

Let $\mathcal{A}$ be the input matrix, and define $G(\mathcal{A}) = (S, C, E_1^{\mathcal{A}})$. For a nonempty subset $S' \subseteq S$, we say that a character is *S'-semi-universal* in $\mathcal{A}$ if its 0-set does not intersect S'. The following lemmas motivate a divide and conquer approach to IDP, which is the basis of our algorithms for solving it.

Lemma 5.4.1 *Let $\mathcal{A}$ be an incomplete matrix with a Σ-free completion $\mathcal{B}$. Let c be S-semi-universal in $\mathcal{A}$. Let $\mathcal{B}'$ be the matrix obtained from $\mathcal{B}$ by setting all entries in the column of c to 1. Then $\mathcal{B}'$ is also a Σ-free completion of $\mathcal{A}$.*

Proof: Suppose to the contrary that $\{s_1, c_1, s_2, c_2, s_3\}$ induce a Σ subgraph in $G(\mathcal{B}')$. Since $G(\mathcal{B})$ is Σ-free, if follows that at least one of the Σ edges was added to $\mathcal{B}'$. But then one of c_1 and c_2 is c, a contradiction. ■

Lemma 5.4.2 *Let $\mathcal{A}$ be an incomplete matrix with a Σ-free completion $\mathcal{B}$. Let $(K_1, \ldots, K_r)$ be a partition of $S \cup C$ such that each K_i is a union of one or more connected components of $G(\mathcal{A})$. Let $\mathcal{B}'$ be the matrix obtained from $\mathcal{B}$ by setting all entries between vertices of K_i and K_j to 0, for all $i \neq j$. Then $\mathcal{B}'$ is also a Σ-free completion of $\mathcal{A}$.*

Proof: Suppose to the contrary that $\{s_1, c_1, s_2, c_2, s_3\}$ induce a Σ subgraph in $G(\mathcal{B}')$. Then one of the non-edges (s_1, c_2) or (c_1, s_3) contains one vertex from K_i and the other from K_j, for $i \neq j$. It follows that there is a path in $G(\mathcal{B}')$ between a vertex of K_i and a vertex of K_j, a contradiction. ■

We now describe two efficient $\tilde{O}(nm)$-time algorithms for solving IDP.

5.4.1 Algorithm A

Algorithm A is described in Figure 5.6. The algorithm outputs the set of non-empty clades of a tree explaining $\mathcal{A}$, or outputs *False* if no such tree exists. The algorithm is recursive and is initially called with Alg_A($\mathcal{A}$) .

Alg_A($\mathcal{A} = ((S, C), E_0, E_?, E_1)$)**:**

1. **If** $|S| > 1$ **then** do:
   (a) Remove all S-semi-universal characters and all null characters from $G(\mathcal{A})$.
   (b) **If** the resulting graph G' is connected **then** output *False* and **halt**.
   (c) Otherwise, let $K_1, \ldots, K_r$ be the connected components of G', and let $\mathcal{A}_1, \ldots, \mathcal{A}_r$ be the corresponding submatrices of $\mathcal{A}$.
   (d) **For** $i = 1, \ldots, r$ **do**: Alg_A($\mathcal{A}_i$) .
2. Output S.

Figure 5.6: Algorithm A for solving IDP.

Theorem 5.4.3 *Algorithm A correctly solves IDP.*

Proof: Suppose that the algorithm outputs *False*. Then there exists a recursive call Alg_A($\mathcal{A}'$) in which the graph $G' = (S', C', E')$ obtained in Step 1b was found to be connected. Suppose to the contrary that $\mathcal{A}$ has a phylogenetic tree. Then by Corollary 5.3.2 there exists some edge set F^*, which solves $\mathcal{A}'$. The graph $G^* = (S', C', F^*)$ is connected and by Theorem 5.3.1, it is also Σ-free. Therefore, by

Proposition 5.3.4 there exists an S'-universal character in G^*. That character must be S'-semi-universal in $\mathcal{A}'$. By Algorithm A this vertex should have been removed at step 1a, a contradiction.

To prove the other direction, we will show that if the algorithm outputs a collection $\mathcal{T}' = \{S_1, \ldots, S_l\}$ of sets, then $\mathcal{T} = \mathcal{T}' \cup \{\emptyset\}$ is a tree which explains $\mathcal{A}$. We first prove that the collection $\mathcal{T}$ of sets is pairwise compatible, implying by Observation 5.2.1 that $\mathcal{T}$ is a tree. Associate with each S_i the recursive call Alg_A($\mathcal{A}_i$) at which it was output. Observe that each such call makes recursive calls associated with disjoint subsets of S_i. By induction, it follows that $S_i \subseteq S_j$ if and only if the recursive call associated with S_i is nested within the one associated with S_j. Otherwise, $S_i \cap S_j = \emptyset$. Hence, $S_1, \ldots, S_l$ are pairwise compatible and, thus, $\mathcal{T}$ is a tree.

It remains to show that $\mathcal{T}$ is a phylogenetic tree for $\mathcal{A}$. Associate each null character with the empty clade. Each other character $\hat{c}$ is removed at Step 1a only once in the course of the algorithm, during some recursive call Alg_A($\hat{\mathcal{A}}$). Associate $\hat{c}$ with the clade $\hat{S}$ which was output at that recursive call. Observe that each non-trivial clade $\hat{S} \in \mathcal{T}$ is associated with at least one character. Finally, define a binary matrix $\mathcal{B}_{n \times m}$ with $b_{sc} = 1$ if and only if s belongs to the clade S_c associated with c. Since $a_{sc} \neq 1$ for all $s \notin S_c$ and $a_{sc} \neq 0$ for all $s \in S_c$, $\mathcal{B}$ is a completion of $\mathcal{A}$. The claim follows. ■

Let $h \leq \min\{m, n\}$ be the height of the reconstructed tree. Each recursive call increases the height of the output tree by at most one. The work at each level of the tree requires: (1) Finding semi-universal vertices; and (2) finding connected components in disjoint graphs whose total number of edges is at most mn. Hence, the total work is $O(mn)$ per level, and a naive implementation requires $O(hmn)$ time. We give a faster implementation below.

Theorem 5.4.4 *Algorithm A has an $O(nm + |E_1| \log^2(n+m))$-time deterministic implementation, and a randomized implementation taking $O(nm + |E_1| \log(l^2/|E_1|) + l(\log l)^3 \log\log l)$ expected time, where $l = n + m$.*

Proof: For the complexity proof we give an alternative, non-recursive implementation of Algorithm A, shown in Figure 5.7. This iterative version mimics the recursive one, but traverses the tree of recursive calls in a breadth first manner, rather than

a depth first manner. Consequently, the implementation deals with a single graph, rather than a different graph per each recursive call. The reduction in complexity is primarily due to the use of an efficient dynamic data structure for graph connectivity. The data structure maintains the connected components of the graph while edge deletions occur.

We now analyze the running time of this implementation. Step 1 takes $O(nm)$ time. Each iteration of the 'while' loop (Step 2) splits the (potential) clades added in the previous one. Thus, Algorithm A performs one iteration of this type per each level of the tree returned, and at most h iterations.

Step 2b requires explicitly computing the connected components of G. Both data structures that we use for storing the connected components of G (see below) maintain a spanning tree for each connected component of G, and allow computing the connected components in $O(n+m)$ time per iteration, or $O(h(m+n)) = O(nm)$ time in total.

The loop of Step 2c is performed at most $\min\{2n-1, m\}$ times altogether, as in each (successful) iteration at least one character is removed from G (Step 2(c)vii), and at least one clade is added to $\mathcal{T}$. Thus, Step 2(c)i takes $O(\min\{n, m\})$ time altogether, and Step 2(c)ii takes $O(nm)$ time in total. Step 2(c)iii takes $O(nm)$ time in total, as it considers each species-character pair only once throughout the execution of the algorithm.

In order to analyze the complexity of Step 2(c)iv, observe that the following invariants hold in this step for each character $c \in C(K_i)$:

- $d_c^? = |\{(s, c) \in E_? : s \in S(K_i)\}|$, as guaranteed by Step 2(c)iii.
- $d_c^1 = |\{(s, c) \in E_1 : s \in S(K_i)\}| = |\{(s, c) \in E_1 : s \in S\}|$, as initialized in Step 1b, since species are never removed, and each species adjacent to c must be in its connected component until c is removed.

Given $d_c^1, d_c^?$ and $|S(K_i)|$, one can check in $O(1)$ time whether c is $S(K_i)$-semi-universal, and thus Step 2(c)iv takes $O(|C(K_i)|)$ time, or $O(hm)$ time in total.

Since each set added to $\mathcal{T}$ in Step 2(c)vi corresponds to at least one character, and each character is associated with exactly one such set, updating $\mathcal{T}$ requires $O(nm)$ time in total. This also implies an $O(nm)$ bound on the size of the output produced in Step 3.

Alg_A_fast$(\mathcal{A} = ((S, C), E_0, E_?, E_1))$ **:**

1. Initialize:
   (a) Set $t \leftarrow 0$, $\mathcal{K}^0 \leftarrow \{S \cup C\}$, $G \leftarrow G(\mathcal{A})$, $\mathcal{T} \leftarrow triv(S)$.
   (b) **For** each character c, and $i \in \{1, ?\}$ **do:**
   Set $d_c^i \leftarrow |\{s \in S : (s, c) \in E_i\}|$.
   (c) Remove all S-semi-universal and all null characters from G.
   (d) Initialize a data structure for maintaining the connected components of G.
2. **While** $E(G) \neq \emptyset$ **do**:
   (a) Increment t.
   (b) Explicitly compute the set $\mathcal{K}^t$ of connected components $K_1, \ldots, K_r$ of G.
   (c) **For** each connected component $K_i \in \mathcal{K}^t$ such that $|E(K_i)| \geq 1$ **do**:
      i. Pick any character $c' \in C(K_i)$.
      ii. Compute $S' = S(K') \setminus S(K_i)$, where K' is the component in $\mathcal{K}^{t-1}$ which contains c'.
      iii. **For** each species-character pair $(s, c) \in S' \times C(K_i)$ **do**:
      If $(s, c) \in E_?$ **then** decrement $d_c^?$.
      iv. Compute the set U of all characters in K_i that are $S(K_i)$-semi-universal in $\mathcal{A}$.
      v. **If** $U = \emptyset$ **then** output *False* and **halt**.
      vi. Set $\mathcal{T} \leftarrow \mathcal{T} \cup \{S(K_i)\}$.
      vii. Remove U from G and update the data structure of connected components accordingly.
3. Output $\mathcal{T}$.

Figure 5.7: An iterative presentation of Algorithm A.

It remains to discuss the cost of the dynamic data structure, which is charged for Step 2(c)vii. Using the dynamic algorithm of [107], the connected components of G can be maintained during $|E_1|$ edge deletions at a total cost of $O(|E_1|\log^2(n+m))$ time spent in Step 2(c)vii. Alternatively, using the Las-Vegas type randomized algorithm of [184] for decremental dynamic connectivity, the edge deletions can be supported in $O(|E_1|\log(l^2/|E_1|)+l(\log l)^3\log\log l)$ expected time. The complexity follows. ■

5.4.2 Algorithm B

We now describe another deterministic algorithm for IDP, which is faster than Algorithm A whenever $|E_1| = \omega((n+m)^2/\log(n+m))$. Algorithm B uses the dynamic-connectivity data structure of [101], which supports deletion of batches of edges from a graph, while detecting after each batch one of the new connected components in the resulting graph (if new components were formed).

Algorithm B is described in Figure 5.8. For an instance $\mathcal{A}$ it outputs the non-empty clades of a tree explaining $\mathcal{A}$ (except possibly the root clade if it has no matching character), or *False* if no such tree exists. It is initially called with Alg_B($\mathcal{A}$).

Theorem 5.4.5 *Algorithm B correctly solves IDP in $O((n+m)^2\log(n+m))$ deterministic time.*

Proof: **Correctness:** We prove correctness by induction on the problem size. If G' is connected (at Step 2c), then by Proposition 5.3.4 $\mathcal{A}$ has no phylogenetic tree, and indeed the algorithm outputs *False*. Otherwise, let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the sub-instances induced on K and $K' = V(G') \setminus K$, respectively, as detected in Step 2b. If $\mathcal{A}$ has a phylogenetic tree then by Corollary 5.3.2 so do $\mathcal{A}_1$ and $\mathcal{A}_2$. On the other hand, let $\mathcal{T}_1, \mathcal{T}_2$ be phylogenetic trees for $\mathcal{A}_1, \mathcal{A}_2$, respectively. Note that by definition, $\mathcal{T}_2$ must contain the trivial clade $S(K')$, which is not necessarily a clade in a phylogenetic tree for $\mathcal{A}$ (if K' has no semi-universal character). To remedy that define $\mathcal{T}_2' = \mathcal{T}_2$ if the algorithm outputs $S(K')$, and $\mathcal{T}_2' = \mathcal{T}_2 \setminus \{S(K')\}$ otherwise. Then $\mathcal{T}_1 \cup \mathcal{T}_2' \cup \{S\}$ is a phylogenetic tree for $\mathcal{A}$.

Complexity: The data structure of [101] dynamically maintains a graph $H = (V, E)$ through batches of edge deletions, with each batch followed by a query for

Alg_B$(\mathcal{A} = ((S, C), E_0, E_?, E_1))$**:**

1. **If** $|S| = 1$ or $G(\mathcal{A})$ has an S-semi-universal vertex **then** output S.
2. **If** $|S| > 1$ **then** do:
   (a) Remove all S-semi-universal characters and all null characters from $G(\mathcal{A})$.
   (b) **If** the resulting graph G' contains a new connected component K **then** do:
      i. Let $\mathcal{A}_1, \mathcal{A}_2$ be the submatrices of $\mathcal{A}$ induced on $V(K)$ and $V(G') \setminus V(K)$, respectively.
      ii. **For** $i = 1, 2$ **do**: Alg_B$(\mathcal{A}_i)$.
   (c) **Else** output *False* and **halt**.

Figure 5.8: Algorithm B for solving IDP.

a newly created connected component in the resulting graph. If we denote by b_0 the number of batches which do not result in a new component, then as shown in [101], the total cost of answering the queries and performing the batch deletions, if eventually all edges are deleted, is $O(|V|^2 \log |V| + b_0 \min\{|V|^2, |E| \log |V|\})$.

We use this data structure to maintain $G(\mathcal{A})$ during all the recursive calls. As $b_0 = 1$ (since in case no new component is formed the algorithm outputs *False* and halts) and $|V| = n+m$, the total cost is $O((m+n)^2 \log(n+m))$ time. This expression dominates the complexity, as finding the semi-universal vertices at each recursive call costs in total only $O(nm)$ time (see proof of Theorem 5.4.4). ■

We remark, that an $\Omega(nm)$-time lower bound for (undirected) binary perfect phylogeny was proven by Gusfield [87]. A closer look at Gusfield's proof reveals that it applies, as is, also to the directed case. As IDP generalizes directed binary perfect phylogeny, any algorithm for this problem would require $\Omega(nm)$ time.

5.4.3 Greedy Approach Fails

We end the section by showing that a simple greedy approach to IDP fails. Let $\mathcal{A}$ be an incomplete matrix. We say that $a_{sc} =?$ is *forced* if there exists an assignment $x \in \{0,1\}$ such that completing a_{sc} to x results in an induced Σ in the graph $(S, C, E_1^{\mathcal{A}'} \cup E_?^{\mathcal{A}'})$ corresponding to the completed matrix $\mathcal{A}'$. $\mathcal{A}$ is called *forced* if it has some forced ?-entry.

A naive greedy algorithm for IDP is as follows: At each step complete one ?-entry in the matrix. If there are no forced entries, choose any ?-entry and complete it arbitrarily. Otherwise, try to complete a forced entry. If such completion is not possible (an induced Σ is formed) report *False*.

Figure 5.9A shows an explainable instance with no forced entries. Setting the bottom-left ?-entry to 0 results in an instance which cannot be explained. A solution matrix is shown in Figure 5.9B.

	Characters			
Species	1	?	0	0
	1	1	?	?
	?	1	1	?
	?	?	1	1
	?	0	?	1

A

	Characters			
Species	1	0	0	0
	1	1	1	1
	1	1	1	1
	1	1	1	1
	1	0	1	1

B

Figure 5.9: A counter-example to the greedy approach. A: The input matrix. B: A solution.

5.5 Determining the Generality of the Solution

A 'yes' instance of IDP may have several distinct phylogenetic trees as solutions. These trees may be related in the following way: We say that a tree $\mathcal{T}$ *generalizes* a tree $\mathcal{T}'$, and write $\mathcal{T} \subseteq \mathcal{T}'$, if every clade of $\mathcal{T}$ is a clade of $\mathcal{T}'$, i.e., the evolutionary scenario expressed by $\mathcal{T}'$ includes all the details of the scenario expressed by $\mathcal{T}$, and possibly more. Therefore, $\mathcal{T}'$ represents a more specific scenario, and $\mathcal{T}$ represents a more general one. We say that a tree $\mathcal{T}$ is the *general solution* of an instance $\mathcal{A}$, if $\mathcal{T}$ explains $\mathcal{A}$ and generalizes every other tree which explains $\mathcal{A}$. Figure 5.10

demonstrates the definitions and also gives an example of an instance that has no general solution.

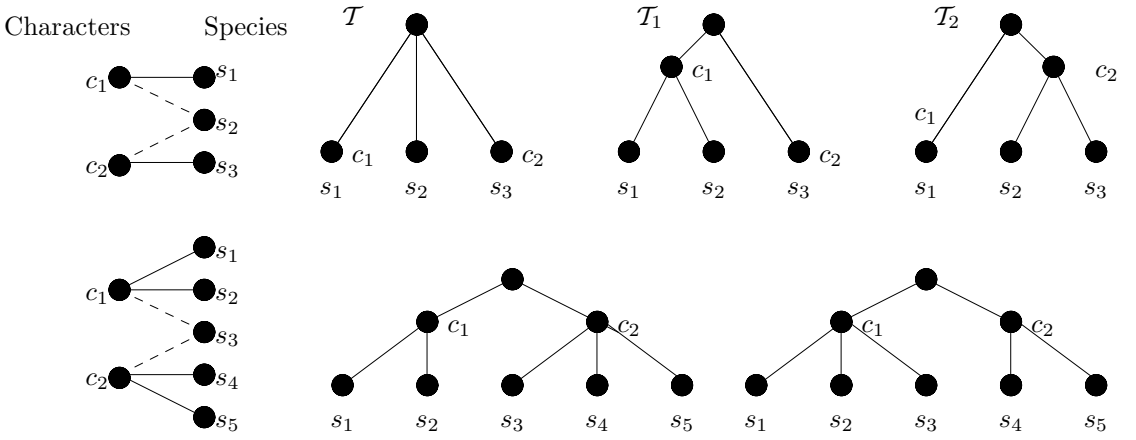


Figure 5.10: Top left: An IDP instance which has a general solution. Dashed lines denote $E_?$-edges, while solid lines denote E_1-edges. Top-right: $\mathcal{T}, \mathcal{T}_1$ and $\mathcal{T}_2$ are the possible solutions. $\mathcal{T}$ generalizes $\mathcal{T}_1$ and $\mathcal{T}_2$ (which are obtained by splitting the root node of $\mathcal{T}$), and is the general solution. Bottom left: An IDP instance which has no general solution. Bottom middle and bottom right: Two possible solutions. The only tree which generalizes both solutions is the tree composed of the trivial clades only, which is not a solution.

We give in this section a characterization of IDP instances that admit a general solution. We prove that whenever a general solution exists, Algorithm A finds it. We also provide an algorithm to determine whether the solution tree $\mathcal{T}$ returned by Algorithm A is general. The complexity of the latter algorithm is shown to be $O(mn + |E_1|d)$, where d is the maximum out-degree in $\mathcal{T}$.

The following notation is used in the sequel: Let $\mathcal{A}$ be an incomplete matrix and let $\hat{S} \subseteq S$. We denote by $W_{\mathcal{A}}(\hat{S})$ the set of $\hat{S}$-semi-universal characters in $\mathcal{A}$. Note that if $\mathcal{A}$ is binary, then $W_{\mathcal{A}}(\hat{S})$ is its set of $\hat{S}$-universal characters. We now define the operator $\tilde{}$ on incomplete matrices: We denote by $\widetilde{\mathcal{A}}$ the submatrix $\mathcal{A}|_{S,C\setminus W_{\mathcal{A}}(S)}$ of $\mathcal{A}$. In particular, $G(\widetilde{\mathcal{A}})$ is the graph produced from $G(\mathcal{A})$ by removing its set of S-semi-universal characters. A species set $\emptyset \neq S' \subseteq S$ is said to be *connected* in a graph G, if S' is contained in some connected component of G.

Lemma 5.5.1 *Let $\mathcal{T}$ be the general solution for an instance $\mathcal{A}$ of IDP. Let $S' =$*

$L(x)$ be a clade of $\mathcal{T}$, corresponding to some node x. Let $\mathcal{T}'$ be the subtree of $\mathcal{T}$ rooted at x, and let $\mathcal{A}'$ be the instance induced on $S' \cup C$. Then $\mathcal{T}'$ is the general solution for $\mathcal{A}'$.

Proof: By Observation 5.3.3, $\mathcal{T}'$ explains $\mathcal{A}'$. Suppose that $\mathcal{T}''$ also explains $\mathcal{A}'$ and $\mathcal{T}' \not\subseteq \mathcal{T}''$. Then $\hat{\mathcal{T}} = (\mathcal{T} \setminus \mathcal{T}') \cup \mathcal{T}''$ explains $\mathcal{A}$, and $\mathcal{T} \not\subseteq \hat{\mathcal{T}}$, a contradiction. ■

A non-empty clade of a tree is called *maximal* if the only clade that properly contains it is S.

Lemma 5.5.2 *Let $\mathcal{T}$ be a phylogenetic tree for a binary matrix $\mathcal{B}$. A non-empty clade S' of $\mathcal{T}$ is maximal if and only if S' is the species set of some connected component of $G(\widetilde{B})$.*

Proof: Suppose that S' is a maximal clade of $\mathcal{T}$. We first claim that S' is contained in some connected component K of $G(\widetilde{B})$. If $|S'| = 1$ this trivially holds. If $|S'| > 1$, let c be a character associated with S'. c is adjacent to all the vertices in S' and to no other vertex. Hence, c is not S-universal, implying that all the edges $\{(c, s) : s \in S'\}$ are present in $G(\widetilde{B})$. This proves the claim. It remains to show that $S(K) = S'$. Suppose $S(K) \supset S'$. In particular, $|S(K)| > 1$. By Proposition 5.3.4, there exists a character c' in $G(\widetilde{B})$ whose 1-set is $S(K)$. Hence, $S(K)$ must be a clade of $\mathcal{T}$ which is associated with c', contradicting the maximality of S'.

To prove the converse, let S' be the species set of some connected component K of $G(\widetilde{B})$. We first claim that S' is a clade. If $|S'| = 1$, S' is a trivial clade. Otherwise, by Proposition 5.3.4 there exists an S'-universal character c' in $G(\widetilde{B})$. Since K is a connected component, c' has no neighbors in $S \setminus S'$. Hence, S' must be a clade in $\mathcal{T}$. Suppose to the contrary that S' is not maximal, then it is properly contained in a maximal clade S'', which by the previous direction is the species set of K, a contradiction. ■

Theorem 5.5.3 *Algorithm A produces the general solution for every IDP instance that has one.*

Proof: Let $\mathcal{A}$ be an instance of IDP for which there exists a general solution $\mathcal{T}^*$. Let $\mathcal{T}_{alg}$ be the solution tree produced by Algorithm A. By definition $\mathcal{T}^* \subseteq \mathcal{T}_{alg}$.

Suppose to the contrary that $\mathcal{T}^* \neq \mathcal{T}_{alg}$. Let S' be the largest clade reported by Algorithm A, which is not a clade of $\mathcal{T}^*$ (S' must be non-trivial), and let S'' be the smallest clade in $\mathcal{T}_{alg}$ which properly contains S'. Let $\mathcal{A}'$ be the instance induced on $S'' \cup C$. By Observation 5.3.3, $\mathcal{A}'$ is explained by the corresponding subtrees $\mathcal{T}'_{alg}$ of $\mathcal{T}_{alg}$ and $\mathcal{T}'^*$ of $\mathcal{T}^*$. By Lemma 5.5.1, $\mathcal{T}'^*$ is the general solution of $\mathcal{A}'$. Due to the recursive nature of Algorithm A, it produces $\mathcal{T}'_{alg}$ when invoked with input $\mathcal{A}'$. Thus, without loss of generality, one can assume that $S'' = S$ and S' is a maximal clade of $\mathcal{T}_{alg}$.

Suppose that $\mathcal{T}^*$ explains $\mathcal{A}$ via a completion $\mathcal{B}^*$, and let $G^* = G(\mathcal{B}^*)$. Since S' is a maximal clade, it is reported during a second level call of Alg_A($\cdot$) (the call at the first level reports the trivial clade S). Hence, it must be the species set of some connected component K in $G(\widetilde{\mathcal{A}})$. Since every S-universal character in G^* is S-semi-universal in $\mathcal{A}$, S' is contained in some connected component K^* of $G(\widetilde{B^*})$. Denote $S^* \equiv S(K^*)$. By Lemma 5.5.2, S^* is a maximal clade of $\mathcal{T}^*$. Since $S' \notin \mathcal{T}^*$, we have $S' \neq S^*$, and therefore, $S^* \supset S'$. But $\mathcal{T}^* \subseteq \mathcal{T}_{alg}$, implying that S^* is also a non-trivial clade of $\mathcal{T}_{alg}$, in contradiction to the maximality of S'. ■

We now characterize IDP instances for which a general solution exists. Let $\mathcal{A}$ be a 'yes' instance of IDP. Consider a recursive call Alg_A($\mathcal{A}'$) nested within Alg_A($\mathcal{A}$), where $\mathcal{A}' = \mathcal{A}|_{C',S'}$. Let $K_1, \ldots, K_r$ be the connected components of $G(\widetilde{\mathcal{A}'})$, computed in Step 1c. Observe that $S(K_1), \ldots, S(K_r)$ are clades to be reported by recursive calls launched during Alg_A($\mathcal{A}'$). A set U of characters is said to be (K_i, K_j)-*critical* if:

- Characters in U are both $S(K_i)$-semi-universal and $S(K_j)$-semi-universal in $\mathcal{A}'$.
- Removing U from $G(\widetilde{\mathcal{A}'})$ disconnects $S(K_i)$.

Note that by definition of U, $U \subseteq W_{\mathcal{A}'}(S(K_i))$, and $a'_{sc} =?$ for all $c \in U, s \in S(K_j)$. A clade $S(K_i)$ is called *optional* (with respect to $\mathcal{A}'$), if $r \geq 3$ and there exists a (K_i, K_j)-critical set for some index $j \neq i$. If $S(K_i)$ is not optional we say it is *mandatory*. In the example of Figure 5.10 (bottom), let $K_1 = \{s_1, s_2, c_1\}$, $K_2 = \{s_3\}$, and $K_3 = \{s_4, s_5, c_2\}$. The set $U = \{c_1\}$ is (K_1, K_2)-critical, so $S(K_1) = \{s_1, s_2\}$ is optional. In contrast, in Figure 5.10 (top) no clade is optional.

Theorem 5.5.4 *The tree produced by Algorithm A is the general solution if and only if all its clades are mandatory.*

Proof: $\Rightarrow$ Suppose that $\mathcal{T}_{alg}$ is the general solution of an instance $\mathcal{A}$. Suppose to the contrary that it contains an optional clade. Without loss of generality, assume it is maximal, i.e., during the recursive call Alg_A($\mathcal{A}$), $G' = G(\widetilde{\mathcal{A}})$ has $r \geq 3$ connected components, $K_1, \ldots, K_r$, and there exists a (K_i, K_j)-critical set U (for some $1 \leq i \neq j \leq r$). Let $\mathcal{A}_i, \mathcal{A}_j$, and $\mathcal{A}_{ij}$ be the sub-instances induced on K_i, K_j, and $K_i \cup K_j$, respectively. Consider the tree $\mathcal{T}'$ which is produced by a small modification to the execution of Alg_A($\mathcal{A}$): Instead of recursively invoking Alg_A($\mathcal{A}_i$) and Alg_A($\mathcal{A}_j$), call Alg_A($\mathcal{A}_{ij}$). Then $\mathcal{T}'$ is a phylogenetic tree which explains $\mathcal{A}$ and includes the clade $S(K_i \cup K_j)$. Since removing U from $G(\widetilde{\mathcal{A}})$ disconnects $S(K_i)$, $|S(K_i)| \geq 2$ so $S(K_i)$ is non-trivial. Moreover, $S(K_i)$ is not a clade of $\mathcal{T}'$ for the same reason. Hence, $\mathcal{T}'$ does not contain all clades of $\mathcal{T}_{alg}$, in contradiction to the generality of $\mathcal{T}_{alg}$.

$\Leftarrow$ Suppose that $\mathcal{T}_{alg}$ is not general the general solution of an instance $\mathcal{A}$, i.e., there exists a solution $\mathcal{T}^*$ of $\mathcal{A}$ such that $\mathcal{T}_{alg} \not\subseteq \mathcal{T}^*$. We shall prove the existence of an optional clade in $\mathcal{T}_{alg}$. (The reader is referred to the example in Figure 5.13 for notation and intuition. The example follows the steps of the proof, leading to the identification of an optional clade.) Let $\mathcal{B}^*$ be a completion of $\mathcal{A}$ which is explained by $\mathcal{T}^*$, and denote $G^* = G(\mathcal{B}^*)$. Let $S' \in \mathcal{T}_{alg} \setminus \mathcal{T}^*$ be the largest clade reported by Algorithm A which is not a clade of $\mathcal{T}^*$. Without loss of generality (as argued in the proof of Theorem 5.5.3), S' is a maximal clade of $\mathcal{T}_{alg}$, and let $S' = S(K_1)$, where $K_1, \ldots, K_r$ are the connected components of $G(\widetilde{\mathcal{A}})$.

Observe that a binary matrix has at most one phylogenetic tree. Thus, an application of Algorithm A to $\mathcal{B}^*$ necessarily outputs $\mathcal{T}^*$. Consider such an application, and let $\{S_i^*\}_{i=1}^t$ be the nested set of reported clades in $\mathcal{T}^*$ which contain S': $S = S_1^* \supset \cdots \supset S_t^* \supset S'$ (see Figure 5.11). For each $i = 1, \ldots, t$, let $\mathcal{B}_i^*$ be the instance invoked in the recursive call which reports S_i^*, and let H_i^* be the graph $G(\widetilde{\mathcal{B}_i^*})$, computed in Step 1a of that recursive call. Let C_i^* be the set of characters in H_i^*. Equivalently, C_i^* is the set of characters in $\mathcal{B}_i^*$ whose 1-set is non-empty and is properly contained in S_i^*. Furthermore, define H_i to be the subgraph of $G(\mathcal{A})$ induced on $S_i^* \cup C_i^*$. Observe that H_i^* is the subgraph of G^* induced on the same vertex set. Since G^* is a supergraph of $G(\mathcal{A})$, each H_i^* is a supergraph of H_i.

Claim 5.5.5 *S' is disconnected in H_t^*, and therefore also in H_t.*

Proof: Suppose to the contrary that S' is contained in some connected component K^* of H_t^*. K^* is thus computed during the t-th recursive call (with argument $\mathcal{B}_t^*$), and $S(K^*)$ is reported as a clade in $\mathcal{T}^*$ by a nested recursive call. Therefore, $S_t^* \supset S(K^*) \supset S'$, where the first proper containment follows from the fact that H_t^* is disconnected, and the second from the assumption that S' is not a clade of $\mathcal{T}^*$. Hence, we arrive at a contradiction to the minimality of S_t^*. ■

We now return to the proof of Theorem 5.5.4. Recall that S' is connected in $H_1 = G(\widetilde{A})$. Thus, the previous claim implies that $t > 1$. Let K_p be a connected component of $G(\widetilde{\mathcal{A}})$ such that $S(K_p) \subseteq S \setminus S_2^*$ (see Figure 5.11). Let l be the minimal index such that there exists some connected component K_i of $G(\widetilde{\mathcal{A}})$ for which $S(K_i)$ is disconnected in H_l. l is properly defined as $S(K_1) = S'$ is disconnected in H_t. $l > 1$, since otherwise some K_i is disconnected in H_1 and, therefore, also in its subgraph $G(\widetilde{\mathcal{A}})$, in contradiction to the definition of $K_1, \ldots, K_r$.

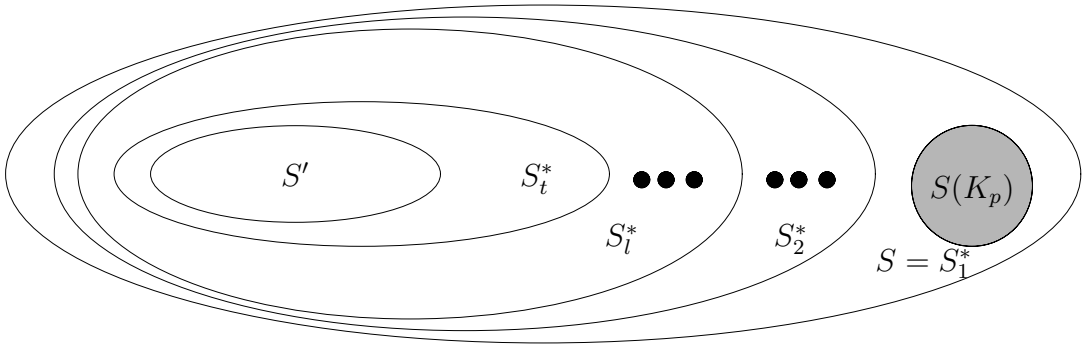


Figure 5.11: The clades $S = S_1^* \supset S_2^* \supset \cdots \supset S_t^* \supset S'$.

By minimality of l, $S_l^* \supseteq S(K_i)$. Also, $S_l^* \supseteq S_t^* \supset S' = S(K_1)$, so $S_l^* \neq S(K_i)$. We now claim that there exists some connected component K_j of $G(\widetilde{\mathcal{A}})$, $j \neq i$, such that $S(K_j) \subseteq S_l^*$. Indeed, if $i \neq 1$ then $j = 1$. If $i = 1$ then $l = t$ (by an argument similar to that in the proof of Claim 5.5.5), and since $S_l^* \setminus S'$ is non-empty, it intersects $S(K_j)$ for some $j \neq i$. By minimality of l, $S(K_j)$ is properly contained in $S_l^* \setminus S'$.

Define $U \equiv W_{G^*}(S_l^*)$. We now prove that U is a (K_i, K_j)-critical set. By definition all characters in U are S_l^*-universal in G^*, and are thus both K_i-semi-universal and K_j-semi-universal in $\mathcal{A}$. $S(K_i)$ is disconnected in $H_l = G(\mathcal{A}|_{C_l^*, S_l^*})$.

Since K_i is a connected component of $G(\widetilde{\mathcal{A}})$, $S(K_i)$ is disconnected in $G(\mathcal{A}|_{C_l^*,S})$, implying that U is a (K_i, K_j)-critical set. Also, K_i, K_j and K_p are distinct, implying that $r \geq 3$ (see Figure 5.12). In conclusion, U demonstrates that $S(K_i)$ is optional. ■

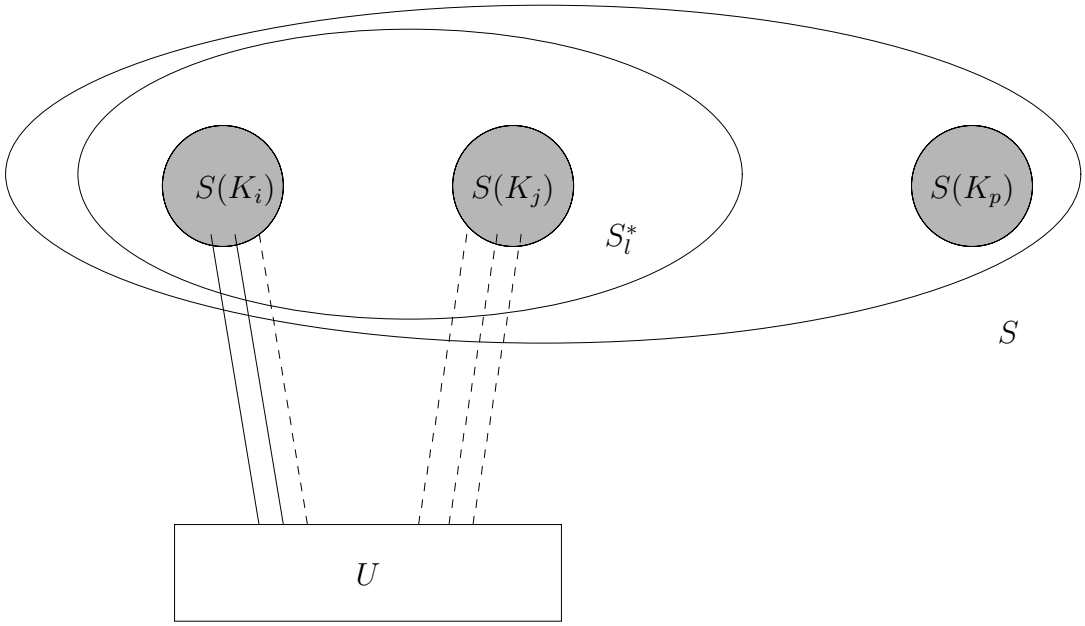


Figure 5.12: The identification of an optional clade. Note that removing U disconnects $S(K_i)$.

The characterization of Theorem 5.5.4 leads to an efficient algorithm for determining whether a solution $\mathcal{T}_{alg}$ produced by Algorithm A is general.

Theorem 5.5.6 *There is an $O(nm + |E_1|d)$-time algorithm to determine if a given solution $\mathcal{T}_{alg}$ is general, where d is the maximum out-degree in $\mathcal{T}_{alg}$.*

Proof: The algorithm simply traverses $\mathcal{T}_{alg}$ bottom-up, searching for optional clades. For each internal node x visited, whose children are $y_1, \ldots, y_{d(x)}$, the algorithm checks whether any of the clades $L(y_1), \ldots, L(y_{d(x)})$ is optional. If an optional clade is found the algorithm outputs *False*. Correctness follows from Theorem 5.5.4.

We show how to efficiently check whether a clade $L(y_i)$ is optional. If $d(x) = 2$, or y_i is a leaf, then certainly $L(y_i)$ is mandatory. Otherwise, let U_i be the set of characters whose origin (in $\mathcal{T}_{alg}$) is y_i. Let U_j^i denote the set of characters in U_i

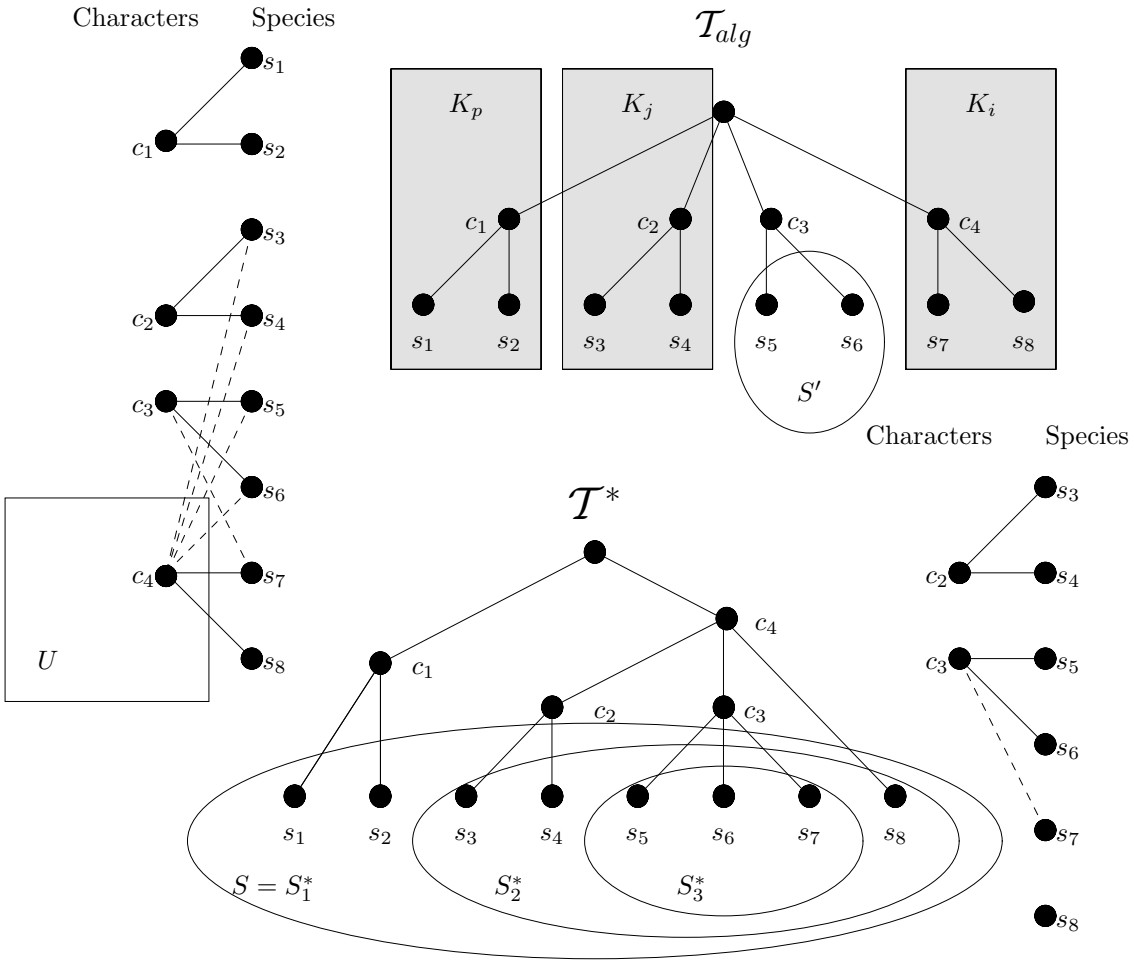


Figure 5.13: An example demonstrating the proof of the 'if' part of Theorem 5.5.4, using the notation in the proof. Left: A graphical representation of an input instance $\mathcal{A}$. Dashed lines denote $E_?$-edges, while solid lines denote E_1-edges. Top right: The tree $\mathcal{T}_{alg}$ produced by Algorithm A. Bottom middle: A tree $\mathcal{T}^*$ corresponding to a completion $\mathcal{B}^*$ that uses all the edges in $E_?$. Bottom right: The graphs H_2 (solid edges) and H_2^* (solid and dashed edges). $\mathcal{T}_{alg} \not\subseteq \mathcal{T}^*$, and $S' = \{s_5, s_6\}$. There are $t = 3$ clades of $\mathcal{T}^*$ which contain S': $S_1^* = \{s_1, \ldots, s_8\}$, $S_2^* = \{s_3, \ldots, s_8\}$, and $S_3^* = \{s_5, s_6, s_7\}$. The component $K_p = \{c_1, s_1, s_2\}$ has its species in $S \setminus S_2^*$. Since $W_{\mathcal{A}}(S) = W_{\mathcal{B}^*}(S) = \emptyset$, $H_1 = G(\mathcal{A})$. Since $W_{\mathcal{B}^*}(S_2^*) = \{c_4\}$, the species set of the connected component $K_i = \{s_7, s_8, c_4\}$ is disconnected in H_2, implying that $l = 2$. For a choice of $K_j = \{s_3, s_4, c_2\}$, the set $U = \{c_4\}$ is (K_i, K_j)-critical, demonstrating that S' is optional.

which are $L(y_j)$-semi-universal, for $j \neq i$. The computation of U_j^i for all i and j takes $O(nm)$ time in total, since for each character c and species s we check at most once whether $(s, c) \in E_?^{\mathcal{A}}$, for an input instance $\mathcal{A}$.

It remains to show how to efficiently check whether for some j, U_j^i disconnects $L(y_i)$ in the appropriate subgraph encountered during the execution of Algorithm A. To this end, we define an auxiliary bipartite graph H^i whose set of vertices is $W_i \cup U_i$, where $W_i = \{w_1, \ldots, w_{d(y_i)}\}$ is the set of children of y_i in $\mathcal{T}_{alg}$. We include the edge (w_r, c_p) in H^i, for $w_r \in W_i, c_p \in U_i$, if $(c_p, s) \in E_1^{\mathcal{A}}$ for some species $s \in L(w_r)$. We construct for each $j \neq i$ a subgraph H_j^i of H^i induced on $W_i \cup (U_i \setminus U_j^i)$. All we need to report is whether H_j^i is connected.

For each i we construct H^i by considering all $E_1^{\mathcal{A}}$ edges connecting characters in U_i to species in $L(y_i)$. This takes $O(|E_1^{\mathcal{A}}|)$ time in total. There are $d(y_i)$ subgraphs H_j^i for every y_i. Hence, computing H_j^i for all j, and determining whether each H_j^i is connected, takes $O\left(|E(H^i)|d(y_i)\right)$ time. Since $\sum_i |E(H^i)| \leq |E_1^{\mathcal{A}}|$, the total time complexity is $O\left(mn + \sum_i |E(H^i)|d(y_i)\right) = O(mn + |E_1^{\mathcal{A}}| \cdot \max_{v \in \mathcal{T}_{alg}} d(v))$. ■

5.6 An Application to Biological Data

We have implemented Algorithm A in C++. The input to the program is an incomplete matrix, and the output is a phylogenetic tree $\mathcal{T}$ in Newick format. We demonstrate our algorithm by reanalyzing the data of Nikaido et al. [151]. This dataset consists of 11 cetartiodactyls species and 20 SINE insertion loci. The input matrix is shown in Table 5.1.

Beaked whale	1	1	1	1	1	1	1	0	?	1	0	1	1	0	0	0	?	1	0	0
Camel	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	?	0	0	0
Chevrotain	?	0	?	?	?	?	?	?	?	1	0	?	?	?	1	1	0	?	0	0
Cow	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0
Deer	0	0	0	0	0	0	0	1	?	1	1	1	1	1	1	?	1	1	0	0
Giraffe	?	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0
Hippopotamus	0	?	0	1	1	1	1	0	1	1	0	1	1	0	0	0	?	1	0	0
Humpback whale	1	1	1	1	1	1	1	0	1	1	0	1	1	0	0	0	?	?	0	0
Peccary	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	1	1
Pig	0	0	0	?	0	0	0	0	?	0	0	0	?	?	0	0	?	1	1	1
Sheep	0	0	0	0	0	?	0	1	1	1	1	1	1	1	1	1	1	1	0	0

Table 5.1: The input matrix of [151].

The final tree, shown in Figure 5.14, is the same tree obtained by Nikaido et al. [151]. It is in fact a general solution for the input instance. The tree supports the following conclusions, reported in [151]:

- Cetaceans are deeply nested within Artiodactyla.
- Cetaceans and hippopotamuses form a monophyletic group.
- Pigs and peccaries form a monophyletic group to the exclusion of hippopotamuses.
- Chevrotains diverged first among ruminants.
- Camels diverged first among cetartiodactyls.

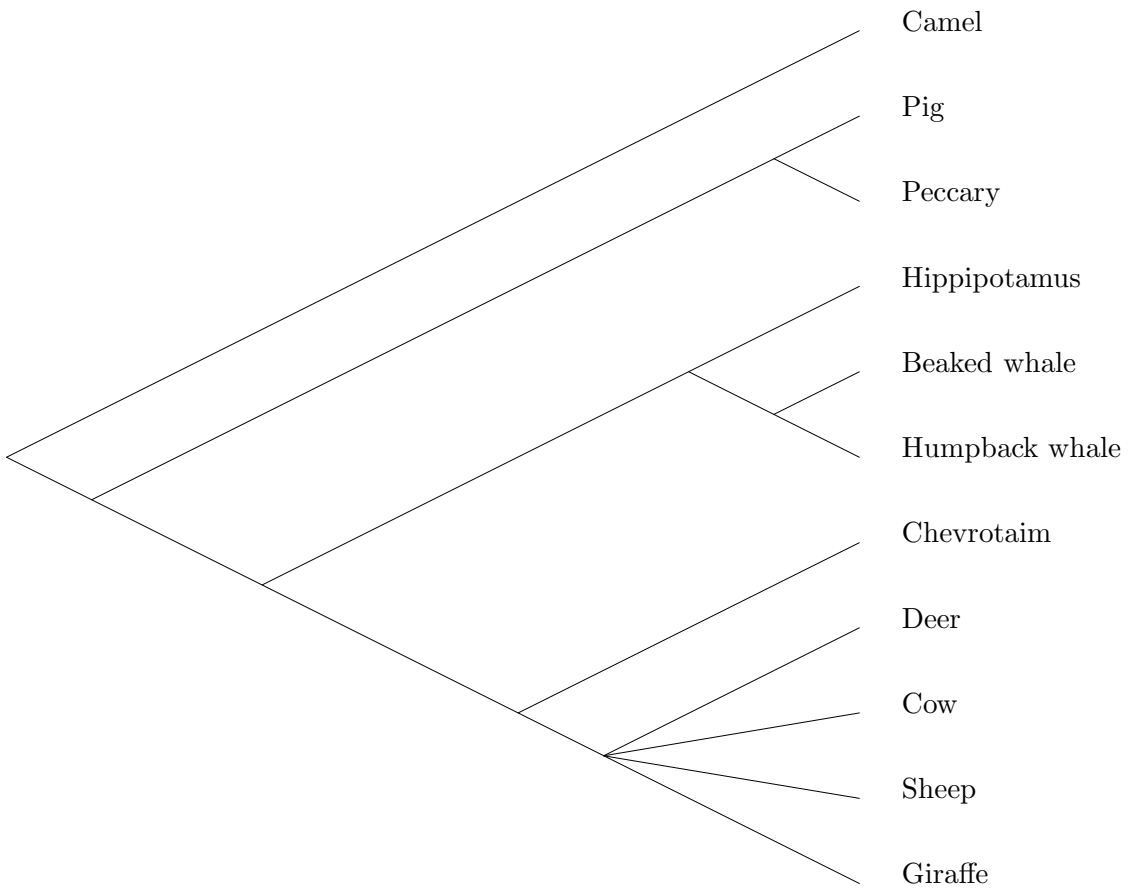


Figure 5.14: The phylogenetic tree obtained on the dataset of [151].

Chapter 6

Clustering Gene Expression Data

This chapter presents a novel clustering algorithm, called CLICK (CLuster Identification via Connectivity Kernels), and its applications to gene expression analysis. The algorithm utilizes graph-theoretic and statistical techniques to identify tight groups (kernels) of highly similar elements, which are likely to belong to the same true cluster. Several heuristic procedures are then used to expand the kernels into the full clusters. We report on the application of CLICK to a variety of biological datasets, ranging from gene expression, cDNA oligo-fingerprinting to protein sequence similarity. In all those applications it outperformed extant algorithms according to several common figures of merit. CLICK is also very fast, allowing clustering of thousands of elements in minutes, and over 100,000 elements in a couple of hours on a standard workstation.

One application of CLICK on which we report in detail is a study of expression data related to the Ataxia-Telangiectasia degenerative disease, done in collaboration with Prof. Y. Shiloh's group, Sackler Faculty of Medicine, Tel-Aviv University, and QBI Enterprises. A-T is a complex multisystem disease resulting from deficiency of the ATM protein kinase. Most notably, A-T cells exhibit profound defects in their responses to ionizing radiation. A-T patients show progressive degeneration of the cerebellum and thymus. Gene expression profiles were constructed for the cerebellum, thymus, and cerebrum of ATM- knockout mice and of wild-type animals, with and without prior X-irradiation. Gene expression patterns were clustered using CLICK. Marked differences were observed in the post- irradiation response between the three tissues and the two genotypes. Unexpectedly, ATM-deficient thymus and

cerebellum from unirradiated animals displayed constitutive activation or repression of numerous genes that the corresponding wild-type tissues showed only after irradiation. This constitutive response to sustained internal genotoxic stress, which correlates with tissue degeneration in human A-T patients, points to an important new characteristic of A-T.

We also show the utility of CLICK in extracting other biological information from gene expression data: We apply CLICK successfully for the identification of common regulatory motifs in the upstream regions of co-regulated genes. Furthermore, we demonstrate how CLICK can be used to accurately classify tissue samples into disease types, based on their expression profiles, achieving success ratios of over 90% on two real datasets.

Finally, we present a new java-based graphical tool, called EXPANDER (EXPression ANalyzer and DisplayER), for gene expression analysis and visualization. This software provides graphical user interface to several clustering methods including CLICK, K-Means, hierarchical clustering and self organizing maps. It enables visualizing the raw expression data and the clustered data in several ways. The EXPANDER tool [174] is used in dozens of laboratories world-wide.

Some of the results in this chapter were published in [175], [171], [173] and [161]. Another application of CLICK in a large scale project of sequencing a super-family of genes is reported in [67].

6.1 Introduction

Technologies for generating high-density arrays of cDNAs and oligonucleotides are developing rapidly and changing the landscape of biological and biomedical research. They enable, for the first time, a global, simultaneous view on the transcription levels of many thousands of genes, when the cell undergoes specific conditions or processes. For several organisms that had their genomes completely sequenced, the full set of genes can already be monitored this way today. The potential of such technologies is tremendous: The information obtained by monitoring gene expression levels in different developmental stages, tissue types, clinical conditions and different organisms can help in understanding gene function and gene networks, assist in the diagnostic of disease conditions and reveal the effects of medical treatments.

A key step in the analysis of gene expression data is the identification of groups of genes that manifest similar expression patterns. This translates to the algorithmic problem of clustering gene expression data. A clustering problem consists of elements and (in most applications) a characteristic vector for each element. A measure of similarity is defined between pairs of such vectors. (In gene expression, elements are usually genes, the vector of each gene contains its expression levels under each of the monitored conditions, and similarity can be measured, for example, by the correlation coefficient between vectors.) The goal is to partition the elements into subsets, which are called *clusters*, so that two criteria are satisfied: *Homogeneity* - elements in the same cluster are highly similar to each other; and *separation* - elements from different clusters have low similarity to each other. Clustering is a fundamental problem which has numerous other applications in biology as well as in many other disciplines. It also has a very rich literature, going back at least a century, and according to some authors, all the way to Aristo.

This chapter is organized as follows: In Section 6.2 we describe the DNA microarray technology for generating gene expression data. In Section 6.3 we formalize the clustering problem and give some background. In Section 6.4 we review the main algorithmic approaches for clustering expression data. In Section 6.5 we present CLICK, a novel clustering algorithm for gene expression analysis. In Section 6.6 we describe applications of CLICK to various biological datasets, and compare its performance to that of other clustering methods. In Section 6.7 we present an analysis of gene expression data related to the Ataxia-Telangiectasia disease. In Sections 6.8 and 6.9 we show the utility of CLICK in regulatory motif finding and in classification problems. Finally, in Section 6.10 we present a graphical tool, called EXPANDER, for visualization and analysis of gene expression data.

6.2 Biological Background

In this section we outline three technologies that generate large scale gene expression data. All three are based on performing a large number of hybridization experiments in parallel on high density arrays (a.k.a. "DNA chips"), between probes and targets. They differ in the nature of the probes and the targets and in other technological aspects, which raise different computational issues in analyzing the data. For more on the technologies and their applications see, e.g., [1, 56, 132, 139, 159].

6.2.1 cDNA Microarrays

cDNA microarrays [167, 168, 139, 159] are high-density arrays which contain large sets of cDNA sequences immobilized on a solid substrate. In an array experiment many gene-specific cDNAs are spotted on a single matrix. The matrix is then simultaneously probed with fluorescently tagged cDNAs corresponding to total RNA pools from test and reference cells, allowing one to determine the relative amount of transcript present in the pool by the type of fluorescent signal generated. Current technology can generate arrays with over 10,000 cDNAs per square centimeter.

cDNA microarrays are produced by spotting PCR products of length approximately 0.6-2.4 KB representing specific genes onto a matrix. The spotted cDNAs are usually chosen from appropriate databases, e.g., GenBank [19] and UniGene [170]. Additionally, cDNAs from any library of interest (whose sequences may be known or unknown) can be used. Each array element is generated by the deposition of a few nanoliters of purified PCR product. Printing is carried out by a robot that spots a sample of each gene product onto a number of matrices in a serial operation.

To maximize the reliability and precision with which quantitative differences in the abundance of each RNA species are detected, one directly compares two samples (test and reference) by labeling them with spectrally distinct fluorescent dyes and mixing the two probes for simultaneous hybridization to one array. The relative representation of a gene in the two samples is assayed by measuring the ratio of the (normalized) fluorescent intensities of the two dyes at the target element. Cy3-dUTP and Cy5-dUTP are frequently used as the fluorescent labels. For the comparison of multiple samples, e.g., in time-course experiments, one often uses the same reference sample with each of the test samples.

6.2.2 Oligonucleotide Microarrays

In oligonucleotide microarrays [64, 95, 131], each spot on the array contains a short synthetic oligonucleotide (oligo), typically 20-30 bases long. The design of oligos is based on the knowledge of the DNA (or EST) target sequences, to ensure high affinity and specificity of each oligo to a particular target gene. Moreover, they should not be near-complementary to other RNAs that may be highly abundant in the sample (e.g., rRNAs, tRNAs, alu-like sequences etc.).

One of the leading approaches to construction of high-density DNA probe arrays employs photolithography and solid-phase DNA synthesis. First, synthetic linkers, modified with a photochemically removable protecting groups, are attached to a glass substrate. At each phase, light is directed through a photolithographic mask to specific areas on the surface to produce localized deprotection. Specific hydroxyl-protected deoxynucleosides are incubated with the surface, and chemical coupling occurs at those sites that have been illuminated. Current technology allows for over 300,000 oligos to be synthesized on a 1.28×1.28 cm array. Key to this approach is the use of multiple distinct oligonucleotides designed to hybridize to different regions of the same RNA. This use of multiple detectors greatly improves signal-to-noise ratio and accuracy of RNA quantitation, and reduces the rate of false-positives and miscalls.

An additional level of redundancy comes from the use of mismatch control probes that are identical to their perfect match partners except for a single base difference in a central position. These probes act as specificity controls: They allow the direct subtraction of both background and cross-hybridization signals, and allow discrimination between 'real' signals and those due to non-specific or semi-specific hybridizations.

6.2.3 Oligonucleotide Fingerprinting

Historically, the Oligonucleotide Fingerprinting (ONF) method preceded the other two [129, 50, 51, 52, 53, 144]. It was initially proposed in the context of Sequencing By Hybridization, as an alternative to DNA sequencing. While that approach to sequencing is currently not competitive, ONF has found other applications, including gene expression. It can be used to extract gene expression information about a cDNA library from a specific tissue under analysis, without prior knowledge on the genes involved. Conceptually, it takes the "reverse" approach to that of the oligo microarrays: The target is on the array, and the oligos are "in the air".

To describe the technique, let us assume that the targets are cDNAs. The ONF method is based on spotting the cDNAs on high density nylon membranes (about 31,000 different cDNA can be spotted currently in duplicates on one filter [53]). Many copies of a short synthetic oligo, typically 7-12 bases long, radioactively labeled, are put in touch with the membrane in proper conditions, and the oligos

hybridize to those cDNAs that contain a DNA sequence complementary to that of the oligo. By inspecting the filter one can detect which of the cDNAs the oligo hybridized to. Ideally, the result of such an experiment is one 1/0 bit for each of the cDNAs.

The experiment is repeated with p different oligos, giving rise to a p-long vector for each cDNA spot, indicating which of the (complements of) oligo sequences are contained in each cDNA. This *fingerprint* vector, similar to a bar-code, identifies the cDNA. Thus, distinct spots of cDNAs originating from the same gene should have similar fingerprints. By clustering these fingerprints, one can identify cDNAs originating from the same gene, and the larger that number – the higher the expression level of the corresponding gene. Gene identification can subsequently be obtained by sample sequencing, or by comparison of average cluster fingerprints to a sequence database [157].

Because of the short oligos used, the hybridization information is rather noisy, but this can be compensated by using longer fingerprints. The method is somewhat less efficient than the other two methods, which measure abundance directly in a single spot. However, it has the advantage of applicability to species with unknown genomes, which oligo microarrays cannot handle, and it requires relatively lower mRNA quantities than cDNA microarrays.

6.3 Mathematical Formulations and Background

Let $N = \{e_1, \ldots, e_n\}$ be a set of n elements, and let $\mathcal{C} = (C_1, \ldots, C_l)$ be a partition of N into subsets. Each subset is called a *cluster*, and $\mathcal{C}$ is called a *clustering solution*, or simply a *clustering*. Two elements e_i and e_j are called *mates with respect to* $\mathcal{C}$ if they are members of the same cluster in $\mathcal{C}$. In the gene expression context, the elements are the genes and we often assume that there exists some correct partition of the genes into "true" clusters. When $\mathcal{C}$ is the true clustering of N, elements that belong to the same true cluster are simply called *mates*.

The input data for a clustering problem is typically given in one of two forms: (1) *Fingerprint data* - each element is associated with a real-valued vector, called its *fingerprint*, or *pattern*, which contains p measurements on the element, e.g., expression levels of an mRNA at different conditions (cf. [56]). (2) *Similarity data*

- pairwise similarity values between elements. These values can be computed from fingerprint data, e.g., by correlation between vectors. Alternatively, the data can represent pairwise dissimilarity, e.g., by computing distances. Fingerprints contain more information than similarity data, but the latter is completely generic and can be used to represent the input to clustering in any application. Note that there is also a practical consideration regarding the presentation: The fingerprint matrix is of order $n \times p$ while the similarity matrix is of order $n \times n$, and in gene expression applications often $n \gg p$.

The goal in a clustering problem is to partition the set of elements N into homogeneous and well-separated clusters. That is, we require that elements from the same cluster will be highly similar to each other, while elements from different clusters will have low similarity to each other. Note that this formulation does not define a single optimization problem: Homogeneity and separation can be defined in various ways, leading to a variety of optimization problems (cf. [92]). Even when the homogeneity and separation are precisely defined, those two objectives are typically conflicting: The higher the homogeneity – the lower the separation, and vice versa.

For a set of elements $K \subseteq N$, we define the *fingerprint* or *centroid* of K to be the mean vector of the fingerprints of the members of K. For two fingerprints x and y we denote their similarity by $S(x, y)$ and their dissimilarity by $d(x, y)$. We say that a symmetric similarity function S is *linear* if for any three vectors u, v, and w, we have $S(u, v + w) = S(u, v) + S(u, w)$. A *similarity graph* is a weighted graph in which vertices correspond to elements and edges are weighted by the similarity values between the corresponding elements.

An alternative formulation of the clustering problem is hierarchical: Rather than asking for a single partition of the elements, one seeks an iterated partition: A *dendrogram* is a rooted weighted tree, with leaves corresponding to elements. Each edge defines the cluster of elements contained in the subtree below that edge. The edge's weight (or length) reflects the dissimilarity between that cluster and the remaining elements. In this formulation the clustering solution is the dendrogram, and each non-singleton cluster, corresponding to a rooted subtree, is split into subclusters. The determination of disjoint clusters is left to the judgment of the user. Typically, one tends to consider as genuine clusters elements of a subtree just below a connecting edge of high weight.

Irrespective of the representation of the clustering problem input, judicious pre-

processing of the raw data is key to meaningful clustering. This preprocessing is application dependent and must be chosen in view of the expression technology used and the biological questions asked. The goal of the preprocessing is to normalize the data and calculate the pairwise element (dis)similarity, if applicable. Common procedures for normalizing fingerprint data include transforming each fingerprint to have mean zero and variance one, a fixed norm or a fixed maximum entry. Statistically based methods for data normalization have also been developed recently (see, e.g., [120]).

6.3.1 Assessment of Solutions

A key question in the design and analysis of clustering techniques is how to evaluate solutions. We present in this section figures of merit for measuring the quality of a clustering solution. Different measures are applicable in different situations, depending on whether a partial true solution is known or not, and whether the input is fingerprint or similarity data. We describe below some of the applicable measures in each case. For other possible figures of merit we refer the reader to [61, 92, 197].

Assessment given the True Solution

Suppose at first that the true solution is known, and we wish to compare it to a suggested solution. Any clustering solution can be represented by a binary $n \times n$ matrix C, in which $C_{ij} = 1$ if and only if i and j belong to the same cluster in that solution. Let T and C be the matrices for the true solution and the suggested solution, respectively. Let n_{kl}, $k, l = 0, 1$, denote the number of pairs (i, j) $(i < j)$ for which $T_{ij} = k$ and $C_{ij} = l$. Thus, n_{11} is the number of true mates which are also mates in the suggested solution, n_{00} is the number of non-mates correctly identified as such, while n_{01} and n_{10} count the disagreements between the true solution and the suggested one.

The *Minkowski measure* (cf. [176]) is defined as $\frac{\|T-C\|}{\|T\|}$ or, equivalently:

$$\sqrt{\frac{n_{01} + n_{10}}{n_{11} + n_{10}}}$$

Hence, it measures the proportion of disagreements to the total number of mates in the true solution. A perfect solution has score zero, and the lower the score – the

better the solution. The *Jaccard coefficient* (cf. [61]) is the ratio

$$\frac{n_{11}}{n_{11}+n_{10}+n_{01}}$$

It is the proportion of correctly identified mates to the sum of the correctly identified mates plus the total number of disagreements. Hence, a perfect solution has score one, and the higher the score – the better the solution. This measure is a lower bound for both the sensitivity ($\frac{n_{11}}{n_{11}+n_{10}}$) and the specificity ($\frac{n_{11}}{n_{11}+n_{01}}$) of the suggested solution.

Note that both measures do not (directly) involve the term n_{00}, since solution matrices tend to be sparse and this term would dominate the other three in good and bad solutions alike. When the true solution is known only for a subset $N^* \subset N$, the Minkowski and Jaccard measures can be computed on the submatrices corresponding to N^*. In some cases, e.g., for cDNA oligo-fingerprint data, we have the additional information that no element of N^* has a mate in $N \setminus N^*$. In these cases, the Minkowski and Jaccard measures are evaluated using all the (unordered) pairs $\{(i,j) : i \in N^*, j \in N \cup N^*, i \neq j\}$.

Assessment when the True Solution is Unknown

When the true solution is unknown, we evaluate the quality of a suggested solution by computing two figures of merit that measure its homogeneity and separation. We define the *homogeneity of a cluster* as the average similarity between its members, and the *homogeneity of a clustering* as the average similarity between mates (with respect to the clustering). Precisely, if $F(i)$ is the fingerprint of element i and the total number of mate pairs is M then:

$$H_{Ave} = \frac{1}{M} \sum_{i,j \text{ are mates}, i<j} S(F(i), F(j)) \ .$$

Similarly, we define the *separation of a clustering* as the average similarity between non-mates:

$$S_{Ave} = \frac{2}{n(n-1)-2M} \sum_{i,j \text{ are non-mates}, i<j} S(F(i), F(j)) \ .$$

Related measures that take a worst case instead of average case approach are minimum cluster homogeneity:

$$H_{Min} = \min_{C} \frac{\sum_{i,j \in C, i<j} S(F(i), F(j))}{\binom{|C|}{2}}$$

and maximum average similarity between two clusters:

$$S_{Max} = \max_{C,C'} \frac{\sum_{i \in C, j \in C'} S(F(i), F(j))}{|C||C'|} .$$

Hence, a solution improves if H_{Ave} or H_{Min} increase, and if S_{Ave} or S_{Max} decrease. In computing all the above measures, singletons are considered as additional one-member clusters. Note that for fingerprint data and a linear similarity function, H_{Ave} and S_{Ave} can be computed in $O(np)$ time (see Section 6.5.6).

For binary similarity data, we use a measure suggested by Z. Yakhini (private communication): Suppose that the input is a similarity graph $G = (V, E)$ with edges representing high similarity (exceeding some threshold). Homogeneity is evaluated by the fraction of edges inside clusters, and separation is evaluated by the percentage of edges between different clusters. That is,

$$\begin{aligned} H &= \frac{|\{(i,j) : i,j \text{ are mates and } (i,j) \in E\}|}{M} \\ S &= \frac{2|\{(i,j) : i,j \text{ are non-mates and } (i,j) \in E\}|}{n(n-1) - 2M} \end{aligned}$$

In any case, the two types of measures, intra-cluster homogeneity and inter-cluster separation, are inherently conflicting, as an improvement in one will correspond to worsening of the other. There are several approaches that address this difficulty. One approach is to fix the number of clusters and seek a solution with maximum homogeneity. This is done for example by the classical K-means algorithm. For methods to evaluate the number of clusters see, e.g., [96, 187]. Another approach is to present a curve of homogeneity vs. separation over a range of parameters for the clustering algorithm used [15]. For another approach for comparing solutions across a range of parameters, see [197].

6.4 Approaches to Clustering

Several algorithmic techniques were previously used in clustering gene expression data, including hierarchical clustering [57], self organizing maps [181], and graph theoretic approaches [97, 17, 175]. We describe these approaches in the sequel. For other approaches to clustering expression patterns, see [144, 8, 76, 104]. Much more information and background on clustering is available, cf. [96, 61, 146, 92].

6.4.1 Hierarchical Clustering

Hierarchical clustering solutions are typically represented by a dendrogram. Algorithms for generating such solutions often work either in a top-down manner, by repeatedly partitioning the set of elements, or in a bottom-up fashion. We shall describe here the latter. Such *agglomerative* hierarchical clustering algorithms are among the oldest and most popular clustering methods [37]. They proceed from an initial partition into singleton clusters by successive merging of clusters until all elements belong to the same cluster. Each merging step corresponds to joining two clusters. The general scheme due to Lance and Williams [125] is presented in Figure 6.1. It is assumed that $D = (d_{ij})$ is the input dissimilarity matrix.

1. Find a minimal entry $d_{i^*j^*}$ in D, and merge clusters i^* and j^*.
2. Modify D by deleting rows and columns i, j and adding a new row and column $i^* \cup j^*$, with their dissimilarities defined by:
$$d_{k,i^*\cup j^*} = d_{i^*\cup j^*,k} = \alpha_{i^*} d_{ki^*} + \alpha_{j^*} d_{kj^*} + \gamma |d_{ki^*} - d_{kj^*}|$$
3. **If** there is more than one cluster **then** go to Step 1.

Figure 6.1: The agglomerative hierarchical clustering scheme.

Common variants of this scheme are the following:

- *Single-linkage*: $d_{k,i^*\cup j^*} = \min\{d_{ki^*}, d_{kj^*}\}$. Here $\alpha_{i^*} = \alpha_{j^*} = 1/2$ and $\gamma = -1/2$.
- *Complete-linkage*: $d_{k,i^*\cup j^*} = \max\{d_{ki^*}, d_{kj^*}\}$. Here $\alpha_{i^*} = \alpha_{j^*} = 1/2$ and $\gamma = 1/2$.
- *Average-linkage*: $d_{k,i^*\cup j^*} = n_{i^*} d_{ki^*}/(n_{i^*} + n_{j^*}) + n_{j^*} d_{kj^*}/(n_{i^*} + n_{j^*})$, where n_i denotes the number of elements in cluster i. Here $\alpha_{i^*} = \frac{n_{i^*}}{n_{i^*}+n_{j^*}}$, $\alpha_{j^*} = \frac{n_{j^*}}{n_{i^*}+n_{j^*}}$ and $\gamma = 0$.

Eisen et al. [57] developed a clustering software package based on the average-linkage hierarchical clustering algorithm. The software package is called Cluster, and the accompanying visualization program is called TreeView. The gene similarity metric used is a form of correlation coefficient. The algorithm iteratively merges

elements whose similarity value is the highest, as explained above. The output of the algorithm is a dendrogram and an ordered fingerprint matrix. The rows in the matrix are permuted based on the dendrogram, so that groups of genes with similar expression patterns are adjacent. The ordered matrix is represented graphically by coloring each cell according to its content. Cells with neutral values (log ratio 0, in case ratio value is log transformed) are colored black, increasingly positive values with reds of increasing intensity, and increasingly negative values with greens of increasing intensity. This presentation has the intuitive appeal of giving a complete view of the clustered data and the solution.

6.4.2 K-Means

K-means [135, 12] is another classical clustering algorithm. It assumes that the number of clusters k is known, and aims to minimize the distances between elements and the centroids of their assigned clusters. Let M be the $n \times m$ fingerprint matrix. For a partition P of the elements in $\{1, \ldots, n\}$ denote by $P(i)$ the cluster assigned to i, and by $c(j)$ the centroid of cluster j. Let $d(v_1, v_2)$ denote the Euclidean distance between the fingerprint vectors v_1 and v_2. K-means tries to find a partition P for which the error-function $E_P = \sum_{i=1}^{n} d(i, c(P(i)))$ is minimum.

Each iteration of K-means updates the current partition by checking all possible modifications of the solution in which one element is moved to another cluster, and making a switch that reduces the error function the most. Figure 6.2 describes the most basic scheme. This algorithm is very easy to implement and is used in many applications.

1. Start with an arbitrary partition P of N into k clusters.
2. For each element i and cluster $j \neq P(i)$ let E_P^{ij} be the cost of a solution in which i is moved to cluster j. **If** $E_P^{i^*j^*} = min_{ij} E_P^{ij} < E_P$ **then** move i^* to cluster j^* and repeat Step 2. Otherwise **halt**.

Figure 6.2: The K-means algorithm.

A heuristic inspired by K-means was developed by Herwig et al. [102] to cluster cDNA oligo-fingerprints. Unlike the standard K-means algorithm, this algorithm does not require a pre-specified number of clusters. Instead, it uses two parameters:

γ is the maximal admissible similarity of two distinct clusters, and ρ is the maximal admissible similarity between an element and a cluster different from its own cluster. (Similarity to a cluster is defined as similarity to its centroid.) Elements are handled one at a time, added to sufficiently close clusters or, otherwise, forming a new cluster. Whenever centroids become too close, their clusters are merged. Unlike the K-means algorithm, an element may be tentatively assigned to more than one cluster and, thus, influence the location of several centroids to which it is sufficiently close. The algorithm is shown in Figure 6.3. Here $S(i, C)$ is the similarity between element i and cluster C.

Start with a set of sufficiently different elements as clusters.
For each remaining element i **do:**
    For each cluster C s.t. $S(i, C) \geq \rho$ **do:**
        add i to C.
        While there exists a cluster C' s.t. $S(C, C') > \gamma$, merge C' into C.
    If i was not added to any cluster **then** form a new cluster $\{i\}$.
Assign each element to the cluster to which it is most similar.

Figure 6.3: The K-menas variant of Herwig et al. [102].

6.4.3 HCS

The HCS (Highly Connected Subgraph) algorithm [97, 98] uses a graph theoretic approach to clustering: The input data is represented as an *unweighted* similarity graph, in which there is an edge between two vertices if and only if the similarity between their corresponding elements exceeds a predefined threshold. The algorithm recursively partitions the current set of elements into two subsets. Before a partition, the algorithm considers the subgraph induced by the current subset of elements. If the subgraph satisfies a stopping criterion then it is declared a cluster. Otherwise, a minimum cut is computed in that subgraph, and the set is split into the two subsets separated by that cut. This scheme is detailed in Figure 6.4.

The following notion is key to the algorithm: A *highly connected subgraph* is an induced subgraph H of G, whose minimum cut value exceeds $|V(H)|/2$. That is, H remains connected if any $\lfloor |V(H)|/2 \rfloor$ of its edges are removed. The algorithm

HCS(G):
If $V(G) = \{v\}$ **then** move v to the singleton set.
Else if G is a cluster **then** output $V(G)$.
Else
    $(H, \bar{H}) \leftarrow$MinCut(G).
    HCS(H).
    HCS($\bar{H}$).

Figure 6.4: The basic scheme of HCS. Procedure MinCut(G) computes a minimum cut of G and returns a partition of G into two subgraphs H and $\bar{H}$ according to this cut.

identifies highly connected subgraphs as clusters.

The HCS algorithm possesses several good properties for clustering [98]: The diameter of each cluster it produces is at most two, and each cluster is at least half as dense as a clique. Both properties indicate strong cluster homogeneity. Inter-cluster separation is not proved, but it is argued that if errors are random, any non-trivial set split by the algorithm is unlikely to have diameter two unless the involved sets are small.

To improve separation in practice, several heuristics are used to expand the clusters and speed up the algorithm:

Iterated-HCS: When the minimum cut value is obtained by several distinct cuts, the HCS algorithm chooses one arbitrarily. This process may break small clusters into singletons. To overcome this, several (1-5) HCS iterations are carried out until no new cluster is found.

Singletons Adoption: Singletons can be "adopted" by clusters: For each singleton element x we compute the number of neighbors it has in each cluster and in the singletons set S. If the maximum number of neighbors is sufficiently large, and is obtained by one of the clusters (rather than by S), then x is added to that cluster. The process is repeated several times.

Removing Low Degree Vertices: When the similarity graph contains vertices with low degrees, one iteration of the minimum cut algorithm may simply separate a low degree vertex from the rest of the graph. This is computationally very expensive, not informative in terms of the clustering, and may happen many times if the graph is large. Removing low degree vertices from G eliminates such iterations, and significantly reduces the running time. The process is repeated with several thresholds on the degree.

6.4.4 CAST

Ben-Dor et al. [17] developed a polynomial algorithm for finding the true clustering with high probability, under the following stochastic model of the data: The underlying correct cluster structure is represented by a cluster graph, and errors are subsequently introduced to the graph by independently removing an existing edge or adding a new edge between each pair of vertices with probability α. If all clusters are of size at least cn, for some constant $c > 0$, the algorithm solves the clustering problem with high probability.

The algorithm uses as input the similarity matrix S. The *affinity* of an element v to a putative cluster C is defined as $a(v) = \sum_{i \in C} S(i, v)$. The polynomial algorithm motivated the use of affinity to develop a faster heuristic called CAST (Clustering Affinity Search Technique) [17], which is implemented in the BioClust package. The algorithm uses a single parameter t. Clusters are generated one by one. Each new cluster is started with a single element, and elements are added or removed from the cluster if their relative affinity is larger or lower than t, respectively, until the process stabilizes. The algorithm is shown in Figure 6.5.

An additional heuristic is employed at the end of the algorithm: A series of moving steps aims at a clustering in which the affinity of every element to its assigned cluster is higher than to any other cluster.

6.4.5 Self Organizing Maps

The self organizing maps were developed by Kohonen [123] as a method for fitting a number of ordered discrete reference vectors to the distribution of vectorial input samples. A self organizing map (SOM) assumes that the number of clusters is known.

While there are unclustered elements **do**:
Pick an unclustered element to start a new cluster C.
Repeat ADD and REMOVE until no changes occur:
ADD: add an unclustered element v with maximum affinity to C if $a(v) > t|C|$.
REMOVE: remove an element u from C with minimum affinity if $a(u) \leq t|C|$.
Add C to the list of final clusters.

Figure 6.5: The CAST algorithm.

Those clusters are organized as a set of nodes in a hypothetical "elastic network", with a simple neighborhood structure on the nodes, e.g., a two-dimensional $k \times l$ grid, and a distance function $d(x, y)$ on the nodes. Each of these nodes is associated with a reference vector in $\mathcal{R}^n$. In the process of running the algorithm, the input vectors direct the movement of the reference vectors, so that an organization of the input vectors over the network emerges. In the following we describe the SOM algorithm in the Euclidean space.

The SOM process is iterative. Denote by $f_i(n)$ the position of the reference vector of node n at the i-th iteration. The initial positioning f_1 is random. The algorithm iteratively selects a random data point p, identifies the nearest reference vector of a node n_p, and updates the reference vectors according to a learning function $\tau(\cdot)$, where vectors of nodes closer to n_p in the neighborhood structure are updated more. The magnitude of the updates decreases with the iteration number. The algorithm is described in Figure 6.6. The function $\tau(\cdot)$ represents the "stiffness" of the network. The intuition for this learning process is that the nodes that are close enough to p will "activate" each other to learn something from p.

The learning function $\tau(\cdot)$ monotonically decreases with $d(n, n_p)$ and with the iteration number i. Two popular choices for the learning function are:

- Neighborhood function: For each node n denote by $N_i(n)$ the set of nodes within some distance from n in the neighborhood structure. Define $\tau(n, n_p, i) = 0$ if $n \notin N(n_p)$ and $\tau(n, n_p, i) = \alpha(i)$ otherwise. $\alpha(i)$ is called the learning-rate and decreases with i.

Arbitrarily set the reference vectors $f_1(v) \in R^n$ for each node v.
For $i = 1$ until no node location is changed by more than ϵ **do**:
    Randomly pick a data point p.
    Compute the node n_p with reference vector $f(n_p)$ closest to p.
    Update all reference vectors: $f_{i+1}(n) = f_i(n) + \tau(n, n_p, i)[p - f_i(n)]$.
Assign each data point to the cluster with the closest reference vector.

Figure 6.6: The Self Organizing Map algorithm.

- Gaussian function: $\tau(n, n_p, i) = \alpha(i) \cdot exp(-\frac{d(n,n_p)^2}{2\sigma^2(i)})$, where $\alpha(i)$ and $\sigma(i)$ decrease with i.

For much more on self organizing maps the reader is referred to [123].

Tamayo et al. [181] devised a gene expression clustering software, GeneCluster, which uses the SOM algorithm. In their implementation they incorporated a neighborhood learning function, for which $\alpha(i) = 0.02T/(T + 100i)$, where T is the maximum number of iterations; and $N_i(n_p)$ contains all nodes whose distance to n_p is at most $\rho(i)$, where $\rho(i)$ decreases linearly with i, $\rho(0) = 3$.

GeneCluster accepts an input file of expression levels together with a two dimensional grid geometry for the nodes. The number of grid points is the prescribed number of clusters. The resulting clusters are visualized by presenting for each cluster its average expression pattern with error-bars. Clusters are presented in their grid order, as clusters of close nodes tend to be similar.

Another implementation of SOM for clustering gene expression profiles was developed in [188].

6.5 The CLICK Clustering Algorithm

In this section we present a new clustering algorithm, which we call CLICK (CLuster Identification via Connectivity Kernels). The algorithm builds on the HCS algorithm of Hartuv and Shamir [98]. It utilizes graph-theoretic and statistical techniques to identify tight groups (kernels) of highly similar elements, which are likely to belong to the same true cluster. Several heuristic procedures are then used to expand the

kernels into the full clusters. CLICK has been implemented and tested on a variety of biological datasets, ranging from gene expression, cDNA oligo-fingerprinting to protein sequence similarity. In all those applications it outperformed extant algorithms according to several common figures of merit.

This section is organized as follows: We first describe the probabilistic framework underlying CLICK. We then describe the algorithm and its extension to large datasets. We next present results of CLICK on simulated data. Finally, we discuss the limitations of CLICK. Results on real biological data are described in later sections.

6.5.1 The Probabilistic Framework

A key modeling assumption in developing CLICK is that pairwise similarity values between elements are normally distributed: Similarity values between mates are normally distributed with mean μ_T and variance σ_T^2, and similarity values between non-mates are normally distributed with mean μ_F and variance σ_F^2, where $\mu_T > \mu_F$. This situation was observed on simulated and real data and can be theoretically justified under certain conditions by the Central Limit Theorem. We detail the arguments in Section 6.5.8. Another modeling parameter is p_{mates}, the probability that two randomly chosen elements are mates.

We denote by $f(x|\mu_T, \sigma_T)$ the *mates probability density function*. We denote by $f(x|\mu_F, \sigma_F)$ the *non-mates probability density function*.

Since CLICK requires knowledge of the parameters $\mu_T, \mu_F, \sigma_T, \sigma_F$, and p_{mates}, an initial step of the algorithm is estimating them. There are two possible methods to compute these parameters: (1) In many cases the true partition for a subset of the elements is known. This is the case, for example, if the clustering of some of the genes in a cDNA oligo-fingerprint experiment is found experimentally (see, e.g., [97]), or more generally, if a subset of the elements has been analyzed using prior biological knowledge (see, e.g., [177]). Based on this partition one can compute the sample mean and sample variance for similarity values between mates and between non-mates, and use these as maximum likelihood estimates for the distribution parameters. The proportion of mates among all known pairs can serve as an estimate for p_{mates}, if the subset was randomly chosen. (2) In case no additional information is given, these parameters can be estimated using the EM algorithm (see, e.g., [146,

Section 3.2.7]). For completeness we outline the algorithm below.

Let $x = (S_{ij})$ be a vector of similarity values and let y be a binary vector, where y_{ij} represents the hidden data of whether i and j are mates. Let $\Theta = \{\mu_T, \mu_F, \sigma_T, \sigma_F, p_{mates}\}$ be the set of parameters of the model. Each EM iteration tries to maximize the function

$$Q(\Theta|\Theta^r) = \sum_y Pr(y|x, \Theta^r) \log Pr(x, y|\Theta)$$

where $\Theta^r = \{\mu_T^r, \mu_F^r, \sigma_T^r, \sigma_F^r, p_{mates}^r\}$ is the set of parameters determined in the previous iteration. (All logarithms in this section are natural-base logarithms.) For iteration r, denote by f_0^r and f_1^r the probability density functions for non-mates and mates, respectively, as implied by Θ^r. Define $p_1^r \equiv p_{mates}^r$ and $p_0^r \equiv 1 - p_1^r$. In the following we omit the superscript r when it is clear from the context.

In the E-step, the expectation of y_{ij} given x and Θ^r is calculated by

$$E(y_{ij}|S_{ij}, \Theta^r) = \frac{f_1(S_{ij})p_1}{f_1(S_{ij})p_1 + f_0(S_{ij})p_0} \ .$$

Define $g_1(S_{ij}) \equiv E(y_{ij}|S_{ij}, \Theta^r)$ and $g_0(S_{ij}) \equiv 1 - g_1(S_{ij})$. Simple manipulations give

$$Q(\Theta|\Theta^r) = \sum_{i<j} \sum_{t=0}^{1} g_t(S_{ij}) \log(f_t(S_{ij})p_t)$$

In the M-step we find the parameters maximizing Q. By differentiating Q according to each of the parameters we find that the optimal parameters for the next iteration are: $\mu_t = \frac{\sum_{i<j} g_t(S_{ij})S_{ij}}{\sum_{i<j} g_t(S_{ij})}$, $\sigma_t^2 = \frac{\sum_{i<j} g_t(S_{ij})(\mu_t - S_{ij})^2}{\sum_{i<j} g_t(S_{ij})}$ and $p_t = \frac{\sum_{i<j} g_t(S_{ij})}{\sum_{i<j} 1}$.

For efficiency, the EM algorithm is executed on a random subset of the input similarity values. In order to initialize the model parameters we do the following: We assume that $\sigma_T = \sigma_F = \sigma$. We enumerate p_{mates} and the distance in standard deviation units between μ_T and μ_F. Using the enumerated values and the mean m and variance v of the input similarity values, we can extract μ_T, μ_F and σ. This can be seen by observing that $m = p_{mates}\mu_T + (1 - p_{mates})\mu_F$ and $v = p_{mates}(1 - p_{mates})(\mu_T - \mu_F)^2 + \sigma^2$. For each enumerated combination we compare the normal distributions with the calculated parameters to the empirical data distribution and choose the best combination for the initialization.

6.5.2 The Basic CLICK Algorithm

The CLICK algorithm works in two phases. In the first phase tightly homogeneous groups of elements, called kernels, are identified. In the second phase these kernels are expanded to the final clusters. In this section we describe the kernel identification step.

The input to this phase is a matrix of similarity values S, where S_{ij} is the similarity value between elements e_i and e_j. When the input is fingerprint data, a *preprocessing step* computes all pairwise similarity values between elements, using a given similarity function. We assume throughout that S is completely stored in the memory. Later we shall describe modifications to the algorithm for handling large datasets whose similarity matrices are too large to fit in the memory.

The algorithm represents the input data as a weighted *similarity graph* $G = (V, E, w)$. In this graph vertices correspond to elements and edge weights are derived from the similarity values. The weight w_{ij} of an edge (i, j) reflects the probability that i and j are mates, and is set to be

$$w_{ij} = \log \frac{Pr(i,j \text{ are mates}|S_{ij})}{Pr(i,j \text{ are non-mates}|S_{ij})} = \log \frac{p_{mates} f(S_{ij}|\mu_t, \sigma_T)}{(1-p_{mates}) f(S_{ij}|\mu_F, \sigma_F)} \tag{6.1}$$

Here $f(S_{ij}|i,j \text{ are mates}) = f(S_{ij}|\mu_T, \sigma_T)$ is the value of the mates probability density function at S_{ij}:

$$f(S_{ij}|i,j \text{ are mates}) = \frac{1}{\sqrt{2\pi}\sigma_T} e^{-\frac{(S_{ij}-\mu_T)^2}{2\sigma_T^2}}$$

Similarly, $f(S_{ij}|i,j \text{ are non-mates})$ is the value of the non-mates probability density function at S_{ij}:

$$f(S_{ij}|i,j \text{ are non-mates}) = \frac{1}{\sqrt{2\pi}\sigma_F} e^{-\frac{(S_{ij}-\mu_F)^2}{2\sigma_F^2}}$$

Hence,

$$w_{ij} = \log \frac{p_{mates}\sigma_F}{(1-p_{mates})\sigma_T} + \frac{(S_{ij}-\mu_F)^2}{2\sigma_F^2} - \frac{(S_{ij}-\mu_T)^2}{2\sigma_T^2} \ .$$

Note that G is a complete graph.

The basic CLICK algorithm can be described recursively as follows: In each step the algorithm handles some connected component of the subgraph induced by the

yet-unclustered elements. If the component contains a single vertex, then this vertex is considered a *singleton* and is handled separately. Otherwise, a stopping criterion (which will be described later) is checked. If the component satisfies the criterion, it is declared a *kernel*. Otherwise, the component is split according to a minimum weight cut. The algorithm outputs a list of kernels which serves as a basis for the eventual clusters, and a list of singletons. It is detailed in Figure 6.7. We assume that procedure MinWeightCut(G) computes a minimum weight cut of G and returns a partition of G into two subgraphs H and $\bar{H}$ according to this cut. The scheme is very similar to that of the HCS algorithm with minimum cut computations replaced by minimum weight cut computations.

Basic-CLICK(G):
If $V(G) = \{v\}$ **then** move v to the singleton set R.
Else if G is a kernel **then**
    Output $V(G)$.
Else
    $(H, \bar{H}) \leftarrow$MinWeightCut(G).
    Basic-CLICK(H).
    Basic-CLICK($\bar{H}$).

Figure 6.7: The basic CLICK algorithm.

The idea behind the algorithm is the following. Given a connected graph G, we would like to decide whether $V(G)$ is a subset of some true cluster, or $V(G)$ contains elements from at least two true clusters. In the former case we say that G is *pure*. In order to make this decision we test for each cut C in G the following two hypotheses:

- H_0^C: C contains only edges between non-mates.
- H_1^C: C contains only edges between mates.

We let $Pr(H_i^C|C)$ denote the posterior probability of H_i^C, for $i = 0, 1$. If G is pure then H_1^C is true for every cut C of G. On the other hand, if G is not pure then there exists at least one cut C for which H_0^C holds. We therefore determine that G is pure if and only if H_1^C is accepted for every cut C of G. In case we decide that G

is pure, we declare it to be a kernel. Otherwise, we partition $V(G)$ into two disjoint subsets, according to a cut C in G. Choosing a cut C which maximizes $Pr(H_0^C|C)$ would favor low weight, unbalanced cuts that separate few vertices from the rest of the graph. This reason, along with efficiency considerations which will become clear shortly, motivate us to choose a cut C for which the posterior probability ratio $\frac{Pr(H_1^C|C)}{Pr(H_0^C|C)}$ is minimum. We call such a partition a *weakest bipartition* of G.

We first show how to find a weakest bipartition of G. To this end, we make a simplifying probabilistic assumption that for a cut C in G the random variables $\{S_{ij}\}_{(i,j)\in C}$ are pairwise independent given that the corresponding element pairs are all mates or all non-mates. We also assume that mate relations between pairs $(i,j) \in C$ are pairwise independent. We denote the weight of a cut C by $W(C)$ and its number of edges by $|C|$. We denote by $f(C|H_0^C)$ the likelihood that the edges of C connect only non-mates, and by $f(C|H_1^C)$ the likelihood that the edges of C connect only mates. We let $Pr(H_i^C)$ denote the prior probability of H_i^C, $i = 0, 1$.

Lemma 6.5.1 *Let G be a complete graph. Then for any cut C in G*

$$W(C) = \log \frac{Pr(H_1^C|C)}{Pr(H_0^C|C)} .$$

Proof: Using Bayes Theorem (cf. [45]) we find that

$$\frac{Pr(H_1^C|C)}{Pr(H_0^C|C)} = \frac{Pr(H_1^C)f(C|H_1^C)}{Pr(H_0^C)f(C|H_0^C)}$$

The joint probability density function of the weights of the edges in C, given that they are independent and normally distributed with mean μ_T and variance σ_T^2, is

$$f(C|H_1^C) = \prod_{(i,j)\in C} \frac{1}{\sqrt{2\pi}\sigma_T} e^{-\frac{(S_{ij}-\mu_T)^2}{2\sigma_T^2}}$$

Similarly,

$$f(C|H_0^C) = \prod_{(i,j)\in C} \frac{1}{\sqrt{2\pi}\sigma_F} e^{-\frac{(S_{ij}-\mu_F)^2}{2\sigma_F^2}}$$

The prior probability for H_1^C is $p_{mates}^{|C|}$ and for H_0^C is $(1-p_{mates})^{|C|}$. Therefore,

$$\begin{aligned}
\log \frac{Pr(H_1^C|C)}{Pr(H_0^C|C)} &= \log \frac{Pr(H_1^C)f(C|H_1^C)}{Pr(H_0^C)f(C|H_0^C)} \\
&= |C| log \frac{p_{mates}\sigma_F}{(1-p_{mates})\sigma_T} + \sum_{(i,j)\in C} \frac{(S_{ij}-\mu_F)^2}{2\sigma_F^2} - \sum_{(i,j)\in C} \frac{(S_{ij}-\mu_T)^2}{2\sigma_T^2} \\
&= W(C) .
\end{aligned}$$

■

Lemma 6.5.1 implies that with our specific edge weight definition, a minimum weight cut of G induces a weakest bipartition of G. However, the computation of a minimum weight cut in a graph with negative edge weights is NP-hard [69]. We give in the next section a heuristic procedure to compute a minimum weight cut.

It remains to show how to decide if G is pure or, equivalently, which stopping criterion to use. For a cut C, we accept H_1^C if and only if $Pr(H_1^C|C) > Pr(H_0^C|C)$. That is, we accept the hypothesis with higher posterior probability.

Let C be a minimum weight cut of G. By Lemma 6.5.1, for every other cut C' of G

$$\log \frac{Pr(H_1^C|C)}{Pr(H_0^C|C)} = W(C) \leq W(C') = \log \frac{Pr(H_1^{C'}|C')}{Pr(H_0^{C'}|C')}$$

Therefore, H_1^C is accepted for C if and only if $H_1^{C'}$ is accepted for every cut C' in G. Thus, we accept H_1^C and declare that G is a kernel if and only if $W(C) > 0$. In practice, we also require a kernel to have at least k elements, with a default value of $k = 15$.

6.5.3 Computing a Minimum Cut

The minimum weight cut problem is polynomial on graphs with non-negative edge weights. The bottleneck in the basic algorithm is the computation of a minimum weight cut in a graph with negative edge weights. This problem is NP-hard even for a complete graph with all its weights 1 or -1 [172]. We overcome this problem using a two-phase process. In the first phase we split the input graph iteratively using a heuristic procedure for computing a minimum weight cut, which is based on a 2-approximation for the related maximum weight cut problem. In the second phase we filter from the resulting components all negative weight edges and then apply the basic CLICK algorithm.

Our heuristic for computing a minimum weight cut applies two steps:

- MAX-CUT approximation: Let w^* be the maximum weight in the input graph. Transform all weights using the transformation $f(w) = w^* - w + \epsilon$, for small $\epsilon > 0$, resulting in positive edge weights. Apply a 2-approximation for MAX-CUT

(cf. [106]) on the weight-transformed graph, and let (V_1, V_2) be the resulting cut.

- Greedy improvement: Starting from (V_1, V_2) greedily move vertices between sides so as to decrease the weight of the implied cut (using the original edge weights).

This heuristic is applied to the input graph recursively, and the recursion stops whenever the output partition for a component is the trivial one (all vertices are on one side of the partition). Since we expect a pure graph (corresponding to a cluster, or part of a cluster) to have minimum cut of positive weight, the recursion is expected to stop at such components. For refining those candidate kernels we execute Basic-CLICK on each resulting component, after filtering negative weight edges from the component. Since filtering negative weight edges leaves us with an incomplete graph, we have to compensate for them and modify Basic-CLICK accordingly. Consider first the decision of whether G is pure or not. It is now possible that H_1^C will be accepted for C but rejected for some other cut of G. Nevertheless, a test based on $W(C)$ approximates the desired test. In order to apply our test criterion we include the filtered edges with their negative weights in the computation of $W(C)$. In case we decide that G is not pure, we use C in order to partition G into two components. This yields an approximation of a weakest bipartition of G.

In order to reduce the running time of the algorithm on large connected components, for which computing a minimum weight cut is very costly, we screen low weight vertices prior to the execution of Basic-CLICK. The screening is done as follows: We first compute the average vertex weight W in the component, and multiply it by a factor which is proportional to the logarithm of the size of the component. We denote the resulting threshold by W^*. We then remove vertices whose weight is below W^*, and continue to do so updating the weight of the remaining vertices, until the updated weight of every (remaining) vertex is greater than W^*. The removed vertices are marked as singletons and handled at a later stage.

6.5.4 The Full Algorithm

The basic CLICK algorithm produces kernels of clusters, which should be expanded to yield the full clusters. The expansion is done by considering the singletons which

were found during the execution of Basic-CLICK. We denote by $\mathcal{L}$ and R the current lists of kernels and singletons, respectively. An *adoption step* repeatedly searches for a singleton v and a kernel K whose pairwise similarity (defined below) is maximum among all pairs of singletons and kernels. If the value of this similarity exceeds some predefined threshold, then v is *adopted* to K, that is, v is added to K and removed from R. Otherwise, the iterative process ends. For some theoretical justification of the adoption step see [17]. For efficiency, the following iterative variant of this step is used in practice, yielding similar results: Each singleton is adopted to its most similar kernel if their similarity value exceeds a threshold. Iteration halts when no adoptions occur.

After the adoption step takes place, we start a recursive clustering process on the set R of remaining singletons. This is done by discarding all other vertices from the initial graph. We iterate that way until no change occurs.

At the end of the algorithm a *merging step* merges similar clusters. The merging is done iteratively, each time merging two kernels whose similarity is the highest, provided that this similarity exceeds a predefined threshold. When two kernels are merged, they are removed from $\mathcal{L}$ and the newly merged kernel is added to $\mathcal{L}$. Finally, a last singleton adoption step is performed.

The full algorithm is detailed in Figure 6.8. Recall that G_R is the subgraph of G induced by the vertex set R. Procedure Split(G) performs the recursive splitting of G using the minimum weight cut heuristic described in Section 6.5.3. Procedure Adoption($\mathcal{L}, R$) performs the singleton adoption step and updates the set of remaining singletons. Procedure Merge($\mathcal{L}$) performs the merging step.

It remains to describe how to calculate the similarity value between a singleton and a kernel, or between two kernels, and how to set the thresholds for the adoption and merging processes. These processes are heuristic by nature, and we have experimented with a variety of alternatives before reaching the scheme described below.

Suppose we are considering the adoption of a singleton v to a kernel K. We shall compute the posterior probability for the event that v belongs to K and compare it to the posterior probability that v and the elements of K belong to two different true clusters. We shall then choose the hypothesis with greater probability. This is done by computing the logarithm of the posterior probability ratio and comparing it to 0. Since all elements of K are believed to belong to the same true cluster C, the

$R \leftarrow N$.
While some change occurs **do**:
    Split(G_R).
    Let $\mathcal{S}$ be the set of resulting components.
    For each $C \in \mathcal{S}$ **do**:
        Remove edges with negative weight from C.
        Filter low-degree vertices from C.
        Basic-CLICK(C).
    Let $\mathcal{L}'$ be the list of kernels produced.
    Let R be the set of remaining singletons.
    Adoption($\mathcal{L}', R$).
    $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}'$.
Merge($\mathcal{L}$).
Adoption($\mathcal{L}, R$).

Figure 6.8: The full CLICK algorithm.

prior probability that v belongs to C is p_{mates}. Hence, the logarithm of the posterior probability ratio is

$$L_{v,K} = \log \frac{p_{mates}}{1 - p_{mates}} + \sum_{k \in K} \log \frac{f(S_{vk}|\mu_T, \sigma_T)}{f(S_{vk}|\mu_F, \sigma_F)} .$$

We adopt v to K only if $L_{v,K} > 0$. Note, that $L_{v,K}$ can be easily computed from the sum of weights of the edges connecting v to K.

Similarly, when considering the merging of two kernels K_1 and K_2 we compute the logarithm of the posterior probability ratio for this merging as follows:

$$L_{K_1,K_2} = \log \frac{p_{mates}}{1 - p_{mates}} + \sum_{k_1 \in K_1, k_2 \in K_2} \log \frac{f(S_{k_1k_2}|\mu_T, \sigma_T)}{f(S_{k_1k_2}|\mu_F, \sigma_F)} .$$

We merge K_1 and K_2 only if $L_{K_1,K_2} > 0$. Again, L_{K_1,K_2} can be easily computed from the sum of weights of the edges connecting the elements of K_1 and K_2.

An Alternative Adopt and Merge Scheme

We describe below an alternative scheme for adoption and merging that we have implemented and tried. Although it is not used in the final version of the algorithm,

we chose to describe it for reasons that will be explained below. In this scheme, our goal is to use the computed kernels in order to produce the most likely clustering. For a clustering $\mathcal{C}$, we define its posterior probability score as:

$$\begin{aligned} L(\mathcal{C}) &= \sum_{i,j \text{ are mates in } \mathcal{C}, i<j} \log(p_{mates} f(S_{ij}|i,j \text{ are mates})) \\ &+ \sum_{i,j \text{ are non-mates in } \mathcal{C}, i<j} \log((1-p_{mates}) f(S_{ij}|i,j \text{ are non-mates})) \end{aligned}$$

The change in score of the current clustering when adding a singleton v to a kernel K, amounts to

$$\sum_{k \in K} (\log \frac{p_{mates}}{1-p_{mates}} + \log \frac{f(S_{vk}|\mu_T, \sigma_T)}{f(S_{vk}|\mu_F, \sigma_F)}) \ .$$

This is exactly the sum of weights of the edges connecting v to the elements of K. Correspondingly, we define the similarity between v and K as this sum of weights. Similarly, the similarity of two kernels is defined as the sum of weights of the edges connecting these kernels. We adopt v to K if this value is positive, implying that the adoption increases the score of the clustering. The same threshold (zero) is used for all adoption steps, except the last one.

In the last stage of the algorithm the merging step and a last adoption step are performed. Our experiments show that trying different thresholds for these steps and choosing the best consumes a reasonable amount of time and improves the quality of the clustering. Hence, the algorithm enumerates several thresholds, computing the score of each resulting clustering. The merging and adoption steps are performed with the threshold that yielded the highest scoring clustering.

This scheme is theoretically more appealing than the former, as it utilizes the same weights as in the kernel identification step. However, in practice, on simulated and real data, it produces somewhat inferior results as the zero threshold for adoption is too strict.

6.5.5 Handling Large and Partial Datasets

Up until now we assumed that the number of elements in the input dataset is small enough to allow storing all pairwise similarity values between elements in the memory. When the number of elements exceeds several thousands, the memory requirements become a serious bottleneck. We therefore employ a different strategy.

We start by partitioning the set of elements into *super-components*, each having a limited size, which enables storing and processing all pairwise similarity values for that component. For each super-component we evaluate the parameters of its similarity distributions and apply the full algorithm to it (except the merge step and the last adoption step, which are performed later for the whole graph).

In order to form the super-components, we start with the full graph and iteratively increase a weight threshold. When the threshold is high enough the graph is split into connected components not larger than the required size. The algorithm is described in Figure 6.9. Initially it is called with the input graph G, a threshold t^* which is set according to the memory size (based on the similarity data distribution), a limit on the size of a super-component $l = 1000$, and $\Delta = \frac{\sigma_T + \sigma_F}{20}$.

Partition(G,t,l):
Let G^t be the subgraph of G spanned by edges of weight$\geq t$.
For every connected component C of G^t **do:**
    If $|C| \leq l$ **then** output C.
    Else Partition(C,$t + \Delta$,l).

Figure 6.9: An algorithm for computing super-components.

After the CLICK algorithm is applied to each super component, we perform a merging step and an adoption step on all resulting kernels and singletons. To this end we need to approximate the weight of the edges (i, j) that are missing from the graph (since their weight is below t^*). The approximation is done as follows: Denote by $\Phi(\cdot)$ the cumulative standard normal distribution function. We set

$$w_{ij}^* = \log \frac{Pr(i,j \text{ are mates} | S_{ij} < t^*)}{Pr(i,j \text{ are non-mates} | S_{ij} < t^*)} = \log \frac{p_{mates} \Phi((t^* - \mu_T)/\sigma_T)}{(1 - p_{mates}) \Phi((t^* - \mu_F)/\sigma_F)} \ .$$

We note that the same heuristics are applied to handle similarity datasets that are incomplete (part of the pairwise similarity values are missing). We also note that some classical clustering algorithms are based on finding the connected components of the similarity graph, e.g., single-linkage hierarchical clustering and the SYSTERS algorithm [124].

6.5.6 Fingerprint Data Enhancements

Several enhancements can be incorporated to CLICK when the input is fingerprint data, which allows various computations that are infeasible with similarity data. Specifically, for linear similarity functions it is possible to exactly compute the average similarity between a singleton and a kernel, or between two kernels, in time and space proportional to the length of a fingerprint, even if some similarity values are missing. To see this, observe that for two kernels K_1 and K_2 with centroids $C(K_1)$ and $C(K_2)$, respectively, it holds that

$$\frac{\sum_{k_1 \in K_1, k_2 \in K_2} S(k_1, k_2)}{|K_1||K_2|} = S(C(K_1), C(K_2)) \ .$$

By storing the centroid of each kernel, these computations can be done in time proportional to the length of a fingerprint, and are thus feasible also on large datasets, for which storing and processing all similarity values is not practical. Note that these fast computations are also possible for similarity functions that are linear on normalized vectors. For example, correlation coefficient is linear when restricted to vectors with mean 0 and variance 1, since the correlation between any two such vectors is simply their dot-prodcut. Direct computations using similarity data may take $\Omega(n^2)$ time.

Efficiency and quality considerations (with respect to large datasets) motivate us to devise a variant of CLICK for clustering fingerprint data. This variant is also designed to give the user control over the homogeneity of the resulting clustering. Let h be an *homogeneity parameter* given by the user with a default value of μ_T. We describe below a variant of CLICK that aims at producing a clustering with homogeneity at least h. Note, that this variant is limited to linear similarity functions.

Recall that the homogeneity of the output clustering is controlled by the kernel identification step, the adoption steps and the merging step. To ensure the tightness of the kernels produced by Basic-CLICK, we require a kernel to have homogeneity at least h. For efficiency, we also filter from the graph all edges that represent similarity values below h, just before Basic-CLICK is called. For the adoption and merge processes we define the similarity between a singleton and a kernel, or between two kernels, as the average similarity between their elements. We set the adoption and merge thresholds to h. In the last adoption step we enumerate several

adoption thresholds and choose the lowest threshold that induces a clustering with homogeneity greater or equal to h.

6.5.7 Implementation and Simulation Results

We have implemented the CLICK algorithm in C++. Our implementation uses the Hao-Orlin algorithm [93] for minimum weight cut computations. This algorithm was shown to outperform other minimum cut algorithms in practice (cf. [31]). Its running time using highest label selection (cf. [31]) is $O(n^2\sqrt{m})$. Table 6.1 describes the running times of CLICK on simulated datasets (described below) of various sizes containing 10 equal-size clusters. The running times were measured on Pentium III 600MHz operated with LINUX. It can be seen that the running time is approximately linear in the number of elements. This is a result of the time reduction heuristics incorporated to CLICK.

#Elements	Total time (min)	Net time (min)
500	0.16	0.15
1000	0.67	0.65
2000	2.4	2.2
5000	4.25	3.47
10000	9.87	7.15

Table 6.1: A summary of the time performance of CLICK on simulated datasets of various sizes. For a given number of elements, recorded are the total time of executing CLICK, and the execution time excluding the preprocessing step (which computes all pairwise similarity values).

We have created an environment for simulating expression data and measuring CLICK's performance on the synthetic data. We use the following simulation setup: The *cluster structure*, i.e., the number and size of clusters, is pre-specified. Each cluster has an associated mean pattern, also called its centroid. Each coordinate of this pattern is drawn uniformly at random from $[0, R]$ for some R, independently from the other coordinates. Each element fingerprint is drawn at random according to a multivariate normal distribution around the corresponding mean pattern. These normal distributions have identical diagonal covariance matrices and differ only in

their expectation vector. In other words, each coordinate i is drawn independently of the other coordinates with a pre-specified standard deviation σ_i. Similar distribution models are used in other works that model gene expression data (see, e.g., [77]).

In our simulations we wished to analyze the performance of the algorithm as a function of the cluster structure and the distance Δ in standard deviation units between μ_T and μ_F (due to the nature of the simulations, $\sigma_T \approx \sigma_F$). This distance can be controlled by changing R. Table 6.2 presents CLICK's results for several simulation setups as measured by the average Jaccard coefficient over 20 runs. The simulated fingerprints in all cases were of length 200. We used $\sigma_i = \sigma(= 5)$ for all coordinates. It can be seen that CLICK performs well (Jaccard coefficient above 0.8) on all cluster structures even for distances as low as one standard deviation.

Cluster structure	$\Delta = 0.75$	$\Delta = 1$	$\Delta = 1.5$	$\Delta = 2$	$\Delta = 2.5$
100×5	0.81	0.95	0.99	1	1
50×10	0.39	0.8	0.97	1	1
50,60,. . .,100	0.75	0.93	0.99	1	1

Table 6.2: CLICK's accuracy in simulations. The reported results are average Jaccard coefficients of CLICK's solutions vs. the correct solutions. Δ is specified in standard deviation units.

6.5.8 Limitations of CLICK

In this section we discuss the limitations of our clustering approach and possible extensions to overcome these shortcomings.

The Normality Assumption

Our key assumption concerns the distribution of similarity values. Theoretically, the normality assumption can be justified in certain cases as shown below. In practice, the normality assumption often holds, as demonstrated by the results in the next section. However, in some applications, e.g., in protein classification, the distribution of similarity values cannot be approximated using a normal distribution. In such cases, other data models can be constructed using prior knowledge or by analyzing

the empirical distribution using standard statistical methods (cf. [162]). Once the mates and non-mates distributions are evaluated, CLICK can be applied to the data.

We shall argue for the validity of the normality assumption under certain conditions on fingerprint data and a Euclidean based similarity function. The fingerprints are assumed to be created as in the simulation in Section 6.5.7. We need the following variant of the Central Limit Theorem proven by Lyapunov (cf. [45]):

Theorem 6.5.2 (Lyapunov, 1901) *Let $x_1, \ldots, x_n$ be independent random variables with expectation $E(x_i) = \mu_i$ and variance $V(x_i) = \sigma_i^2$. Let $X_n = \sum_{i=1}^n x_i$. If $E(|x_i - \mu_i|^3) < \infty$ for all i, and*

$$\lim_{n\to\infty} \frac{\sum_{i=1}^n E(|x_i - \mu_i|^3)}{(\sum_{i=1}^n \sigma_i^2)^{3/2}} = 0$$

then X_n approaches the standard normal distribution as n approaches infinity.

Let $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ be the fingerprints of two random mates belonging to cluster C. Let $S_{xy} = -\sum_{i=1}^n (x_i - y_i)^2$ be their similarity value. S_{xy} is minus the squared Euclidean distance between these fingerprints. Suppose that $E(x_i) = E(y_i) = \mu_i(C)$ and recall that $V(x_i) = V(y_i) = \sigma_i^2$, where $E(\cdot)$ and $V(\cdot)$ denote the expectation and variance of a random variable, respectively. Straightforward computations give $E(S_{xy}) = -2\sum_{i=1}^n \sigma_i^2$ and $V(S_{xy}) = -8\sum_{i=1}^n \sigma_i^4$. The random variables $(x_i - y_i)^2$ can be empirically shown to satisfy Lyapunov's condition and, thus, S_{xy} is asymptotically normally distributed. As evident from the above calculations, this distribution does not depend on the identity of the cluster, since its parameters depend on the σ_i-s only.

For any two true clusters we can argue similarly that non-mate similarity values (between pairs of elements, one from each cluster) are approximately normally distributed. However, the parameters of these distributions depend on the identity of the clusters: Consider two clusters C and C' and let $x \in C, y \in C'$ be two random elements. Then S_{xy} is distributed with parameters:

$$E(S_{xy}) = -\sum_{i=1}^n (2\sigma_i^2 + (\mu_i(C) - \mu_i(C'))^2) \ ,$$

$$V(S_{xy}) = -\sum_{i=1}^n 2\sigma_i^4 + 4\sigma_i^2(\mu_i(C)^2 + \mu_i(C')^2) \ .$$

Hence, in general the non-mates distribution is not normal. However, if $\sigma_i = \sigma$ for all i, the cluster centers have identical norms, and the angle between every pair of centroid vectors is fixed, then the non-mates distribution is asymptotically normal.

Normality Testing

In this section we shall test for the normality of the similarity distributions obtained on simulated and real data. We first describe the normality test we use, and then apply it to the data.

Suppose we are given a set of ordered random similarity values $s_1 < \ldots < s_n$ and we wish to test whether these values follow a normal distribution. To this end we use the probability plot method (cf. [162]). The test statistic is the correlation coefficient r between the vector of ordered values $(s_1, \ldots, s_n)$ and the vector of distribution quantiles $(\Phi^{-1}(\frac{1}{n+1}), \ldots, \Phi^{-1}(\frac{n}{n+1}))$, where Φ is the cumulative standard normal distribution. For $n = 75$ and a significance level of 0.01 the test accepts for $r \geq 0.9752$.

As a first "sanity check" we simulated fingerprint data, as detailed in Section 6.5.7. We used correlation coefficient as a similarity measure for the simulated data. Table 6.3A gives the correlation coefficient of the mates and non-mates probability plots for two cluster structures. The results indicate the validity of the normality assumption.

In order to test our assumptions on real data we used two datasets. The first dataset is a monocytes cDNA oligo-fingerprinting dataset of size 2,329, for which an approximate true solution is known [97]. The second is a yeast cell-cycle expression dataset of 698 genes, for which an approximate true solution is available from biological knowledge [177]. Both datasets are described in detail in Section 6.6. The similarity function used for the first dataset was vector dot-product. For the second dataset we used the correlation coefficient as a similarity measure. Table 6.3B presents the test results on the two datasets. The normality hypothesis is accepted for both datasets. Figure 6.10 shows the distribution of mates and non-mates similarity values for the oligo-fingerprinting data.

Cluster structure	Mates	Non-mates
50×10	0.997	0.996
50,60,. . .,100	0.997	0.994

A

Dataset	Mates	Non-mates
Monocytes	0.983	0.995
Yeast cell-cycle	0.995	0.997

B

Table 6.3: A test for normality of mates and non-mates distributions on simulated (A) and real (B) fingerprint data. The normality hypothesis is accepted in all cases.

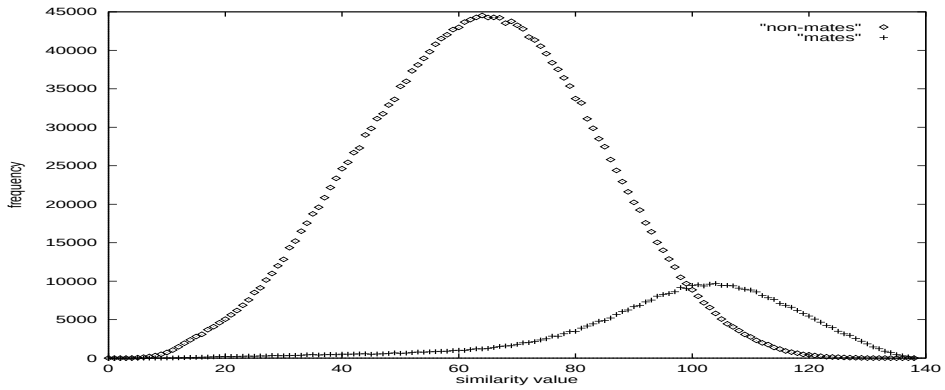


Figure 6.10: Similarity values between mates and between non-mates in the peripheral blood monocytes cDNA dataset of [97].

Other Limitations

Independence Assumption: The probabilistic computations in the algorithm are also based on independence assumptions with respect to similarity values and mate relations. These assumptions enable us to formulate the likelihood ratio of a cut as the sum of weights of its edges. Similar assumptions are implicit in other clustering algorithms (see, e.g., [17]).

Edge Weights: The weight of an edge can be viewed as a sum of two terms (see Equation 6.1): The likelihood ratio of its associated similarity value (i.e., $\log \frac{f(S_{ij}|\mu_T,\sigma_T)}{f(S_{ij}|\mu_F,\sigma_F)}$) and $\log \frac{p_{mates}}{1-p_{mates}}$. When the value of p_{mates} is small and so is the distance between μ_T and μ_F (in standard deviation units), the resulting weight may be negative for the majority of the true mate pairs. In such cases we heuristically adjust the weights in the graph by replacing the term $\log \frac{p_{mates}}{1-p_{mates}}$ with $-E_T$, where E_T is the expected likelihood ratio for an edge connecting a pair of mates. E_T is evaluated using a Monte-Carlo process by drawing at random similarity values

according to the mate distribution and calculating their average likelihood ratio. Based on our experiments, we chose to apply this adjustment whenever the weight corresponding to similarity value $\mu_T + \frac{\sigma_T}{2}$ is negative (i.e., edge weights are negative for approximately 70% of the similarity values between mates).

6.6 Applications to Biological Data

In this section we describe CLICK's results on several biological datasets ranging from gene expression, cDNA oligo-fingerprinting to protein sequence similarity.

6.6.1 Gene Expression

CLICK was first tested on the yeast cell cycle dataset of Cho et al. [32]. That study monitored the expression levels of 6,218 *S. cerevisiae* putative gene transcripts (ORFs) measured at 10-minutes intervals over two cell cycles (160 minutes). We compared CLICK's results to those of GeneCluster [181]. To this end, we applied the same filtering and data normalization procedures of [181]. The filtering removes genes that do not change significantly across samples, leaving a set of 826 genes. The data preprocessing includes the removal of the 90-minutes time-point and normalizing the expression levels of each gene to have mean zero and variance one within each of the two cell-cycles.

CLICK clustered the genes into 18 clusters and left no singletons. These clusters are shown in Figure 6.11. A summary of the homogeneity and separation parameters for the solutions produced by CLICK and GeneCluster is shown in Table 6.4. CLICK obtained better results in all the measured parameters. A putative true solution for a subset of the genes was obtained through manual inspection by Cho et al. [32]. Cho et al. identified 416 genes that have periodic patterns and partitioned 383 of them into five cell-cycle phases according to their peak time. We calculated Jaccard coefficients for the two solutions based on 250 of these genes that passed the variation filtering. The results are shown in Table 6.4. It can be seen that CLICK's solution is much more aligned with the solution reported in [32]. In particular, two putative true clusters are the sets of late G1-peaking genes and M-peaking genes, reported in [32]. Out of the 107 late G1-peaking genes that passed the filtering, CLICK placed 93% (100 genes) in a single cluster of size 191 (Figure 6.11, cluster 1). In

contrast, in the solution of Tamayo et al. [181] 86% of these genes were contained in three clusters of total size 139. Out of the 40 M-peaking genes that passed the filtering, CLICK placed 88% (35 genes) in a single cluster of size 105 (Figure 6.11, cluster 2), while in GeneCluster's solution 93% of these genes were spread among three clusters of total size 99.

Program	#Clusters	Homogeneity		Separation		Jaccard
		H_{Ave}	H_{Min}	S_{Ave}	S_{Max}	
CLICK	18	0.62	0.46	-0.05	0.33	0.54
GeneCluster	30	0.59	0.22	-0.01	0.81	0.28

Table 6.4: A comparison between CLICK and GeneCluster on a yeast cell-cycle dataset of [32].

As another test, we analyzed the dataset of Iyer et al. [112] which studied the response of several human fibroblasts to serum. It contains expression levels of 8,613 human genes obtained as follows: Human fibroblasts were deprived of serum for 48 hours and then stimulated by addition of serum. Expression levels of genes were measured at 12 time-points after the stimulation. An additional data-point was obtained from a separate unsynchronized sample. A subset of 517 genes whose expression levels changed substantially across samples was analyzed by the hierarchical clustering method of [57]. The data was normalized by dividing each entry by the expression level at time zero, and taking a logarithm of the result. For ease of manipulation, we also transformed each fingerprint to have norm 1. The similarity function used was dot-product, giving values identical to those used in [57]. CLICK clustered the genes into 6 clusters with no singletons. These clusters are shown in Figure 6.12. Table 6.5 presents a comparison between the clustering quality of CLICK and the hierarchical clustering of [57] on this dataset. The two clusterings are incomparable since CLICK's solution has better separation while the solution of [57] has better average homogeneity. In order to directly compare the two algorithms we reclustered the data using CLICK with homogeneity parameter 0.76 (instead of the default value $\mu_T = 0.65$, see Section 6.5.6), since this value is the average homogeneity of the hierarchical solution. CLICK produced 6 new clusters and 28 singletons. The solution parameters are given in Table 6.5. Note that CLICK performs better in all parameters.

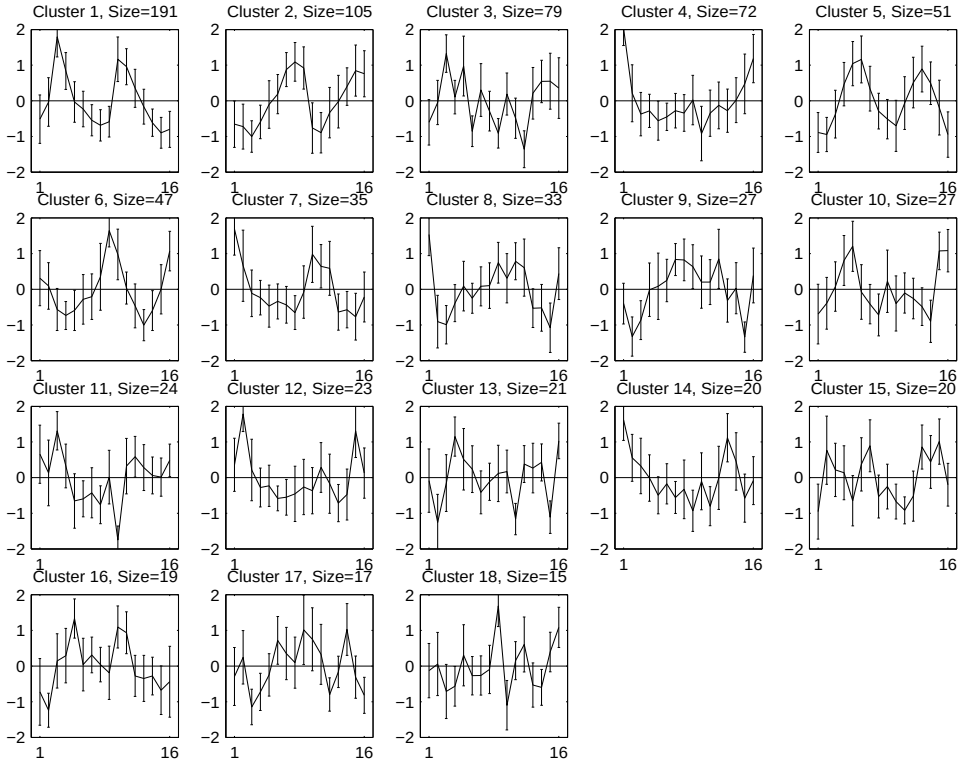


Figure 6.11: CLICK's clustering of the yeast cell cycle data of [32]. x-axis: Time points 0-80,100-160 at 10-minutes intervals. y-axis: Normalized expression levels. The solid line in each sub-figure plots the average pattern for that cluster. Error bars display the measured standard deviation. The cluster size is printed above each plot.

6.6.2 cDNA oligo-fingerprints

We next studied two datasets of oligonucleotide fingerprints of cDNAs obtained from Max Planck Institute of Molecular Genetics in Berlin. The first dataset we analyzed contains 2,329 cDNAs fingerprinted using 139 oligos. This dataset was part of a library of some 100,000 cDNAs prepared from purified peripheral blood monocytes by the Novartis Forschungsinstitut in Vienna, Austria (see [97]). An approximate true clustering of these 2,329 cDNAs is known from back hybridization experiments performed with long, gene-specific oligonucleotides. It contains 18 gene clusters varying in size from 709 to 1. The second dataset contains 20,275 cDNAs originating from sea urchin egg, fingerprinted using 217 oligos (see [157]). For this dataset an approximate true solution is known on a subset of 1,811 cDNAs. Fingerprint

Program	#Clusters	Homogeneity		Separation	
		H_{Ave}	H_{Min}	S_{Ave}	S_{Max}
CLICK.1	6	0.72	0.42	-0.29	0.55
CLICK.2	6	0.78	0.68	-0.19	0.54
Hierarchical	10	0.76	0.65	-0.08	0.75

Table 6.5: A Comparison between CLICK and the hierarchical clustering of [57] on the dataset of response of human fibroblasts to serum [112]. CLICK.1 represents the first CLICK solution with the default homogeneity parameter. CLICK.2 represents a solution of CLICK with homogeneity parameter 0.76.

normalization was done as explained in [143].

Table 6.6 shows a comparison of CLICK's results on the blood monocytes dataset with those of the HCS algorithm [97]. Table 6.7 shows a comparison of CLICK's results on the sea urchin dataset with those of the K-means algorithm of [102]. CLICK outperforms the other algorithms in all figures of merit.

Program	#Clusters	#Singletons	Minkowski	Jaccard
CLICK	16	20	0.63	0.66
HCS	16	206	0.71	0.55

Table 6.6: A comparison between CLICK and HCS on the blood monocytes cDNA dataset.

Program	#Clusters	#Singletons	Minkowski	Jaccard
CLICK	128	12,186	0.77	0.47
K-Means	3,486	2,473	0.79	0.4

Table 6.7: A comparison between CLICK and K-means on the sea urchin cDNA dataset.

We note that the difference between the solutions of K-Means and CLICK in the number of clusters and singletons on the sea urchin dataset is due to the fact that by default CLICK identifies clusters of size at least 15. The number of clusters with

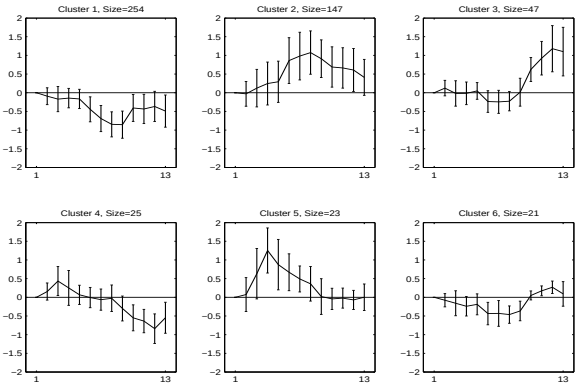


Figure 6.12: CLICK's clustering of the fibroblasts serum response data [112]. x-axis: 1-12: Synchronized time points. 13: Unsynchronized point. y-axis: Normalized expression levels. The solid line in each sub-figure plots the average pattern for that cluster. Error bars display the measured standard deviation. The cluster size is printed above each plot.

15 or more elements in the K-Means solution is 129, and the number of elements that are not members of such clusters is 14,924.

6.6.3 Protein Classes

CLICK was also applied to two protein sequence similarity datasets. The first dataset contains 72,623 proteins from the ProtoMap project [198]. The second originated from the SYSTERS project [124] and contains 117,835 proteins. Both datasets contain for each pair of proteins an E-value of their similarity as computed by BLAST [9].

Protein classification is inherently hierarchical, so the assumption of normal distribution of similarity values does not seem to hold. In order to apply CLICK to the data, we made the following modifications:

1. The weight of an edge (i, j) was set to be $w_{ij} = \log \frac{p_{mates}(1-p_{ij})}{(1-p_{mates})p_{ij}}$, where p_{ij} is the E-value and, hence, also practically the p-value, of the similarity between i and j. We removed edges whose corresponding E-value was above 10^{-20}.

2. The weight of a missing (removed) edge was evaluated as $\log \frac{p_{mates}}{1-p_{mates}}$.

3. For the adoption step we calculated for each singleton r and each kernel K

the ratio

$$\frac{\sum_{k\in K} w_{rk}}{|K|} .$$

We then chose the pair r, K with the highest ratio and r was adopted to K if this ratio exceeded some predefined threshold w^*.

4. For the merging step we calculated for each pair of kernels K_1 and K_2 the ratio

$$\frac{\sum_{k_1\in K_1, k_2\in K_2} w_{k_1k_2}}{|K_1||K_2|} .$$

We then chose the pair K_1, K_2 with the highest ratio and merged K_1 and K_2 if this ratio exceeded w^*.

For the evaluation of the ProtoMap dataset we used a Pfam classification for a subset of the data consisting of 17,244 single-domain proteins, which is assumed to be the true solution for this subset. We compared our results to the results of ProtoMap with a confidence level of 10^{-20} on this dataset. The comparison is shown in Table 6.8. The results are very similar, with a slight advantage to CLICK.

Program	#Clusters	#Singletons	Minkowski	Jaccard
CLICK	7,747	16,612	0.88	0.39
ProtoMap	7,445	16,408	0.89	0.39

Table 6.8: A comparison between CLICK and ProtoMap on a dataset of 72,623 proteins.

For the SYSTERS dataset, no "true solution" was available, so we evaluated the solutions of CLICK and SYSTERS using the figures of merit described in Section 6.3.1. Table 6.9 presents the results of the comparison. The results show a significant advantage to CLICK.

Program	#Clusters	#Singletons	Homogeneity	Separation
CLICK	9,429	17,119	0.24	0.03
SYSTERS	10,891	28,300	0.14	0.03

Table 6.9: A comparison between CLICK and SYSTERS on a dataset of 117,835 proteins.

6.6.4 A Blind Test

In order to compare the characteristics of each of the clustering methods described in the beginning of this chapter, we applied them in a blind test to a yeast cell-cycle dataset of Spellman et al. [177] containing the gene expression levels of yeast ORFs over 79 conditions.

The original dataset contains samples from yeast cultures synchronized by four independent methods: α factor arrest (samples taken every 7 minutes for 119 minutes), arrest of a cdc15 temperature sensitive mutant (samples taken every 10 minutes for 290 minutes), arrest of a cdc28 temperature sensitive mutant (this part of the data is from [32]; samples taken every 10 minutes for 160 minutes), and elutriation (samples taken every 30 minutes for 6.5 hours). It also contains separate experiments in which G1 cyclin Cln3p or B-type cyclin Clb2p were induced.

Spellman et al. identified in this data 800 genes that are cell-cycle regulated [177]. The dataset that we used contains the expression levels of 698 out of those 800 genes, which have up to three missing entries, over the 72 conditions that cover the α factor, cdc28, cdc15, and elutriation experiments. (As in [181], the 90 minutes datapoint was omitted from the cdc15 experiment.) Each row of the 698×72 matrix was normalized to have mean 0 and variance 1. Note that by normalizing the variance different gene amplitudes are deemphasized and periodicity is more prominent.

Based on the analysis conducted by Spellman et al., we expect to find in the data five main clusters: G1-peaking genes, S-peaking genes, S/G2-peaking genes, G2/M-peaking genes, and M/G1-peaking genes. Each of these was shown to contain biologically meaningful sub-clusters.

The 698×72 dataset was clustered using four of the methods described above: K-means, SOM, CAST, and CLICK. The similarity measure used was Pearson correlation coefficient. The authors of each of the programs were given the dataset and asked to provide a clustering solution. The identity of the dataset was not described and genes were permuted in an attempt to perform a "blind" test. (Yet, admittedly, anyone familiar with the gene expression literature could have identified the nature of the data.) The authors were told that the average homogeneity and average separation would be used to evaluate the quality of the solutions. However, the exact formulas used are somewhat different from those originally planned and reported to the authors, as we later found that the new formulas are more adequate.

The following table summarizes the solutions produced by each program and their homogeneity and separation parameters. The so-called 'True' clustering, reported in [177], is that obtained manually by Spellman et al., by inspecting the expression patterns and comparing to the literature. The solution of CLICK contains 23 singletons.

Program	#Clusters	Homogeneity		Separation	
		H_{Ave}	H_{Min}	S_{Ave}	S_{Max}
K-Means	49	0.45	-0.04	0.03	0.63
CAST	5	0.37	0.24	-0.05	0.09
GeneCluster	6	0.39	0.25	-0.03	0.23
CLICK	6	0.39	0.32	-0.04	0.22
'True'	5	0.33	0.26	-0.04	0.25

Table 6.10: A summary of the clustering solutions and their figures of merit for the data of [177].

Figure 6.13 depicts the values of each solution on a plot of the homogeneity vs. separation. It can be seen that CLICK's solution outperforms the 'True' solution and GeneCluster's solution. The solution of CAST also outperforms the 'True' solution. K-means, which generated many more clusters, achieved the highest homogeneity, at the expense of very poor separation. The solutions of CAST and CLICK are incomparable.

6.7 Application to Ataxia-Telangiectasia

In this section we report on the use of CLICK in investigating a human genetic disorder, ataxia-telangiectasia (A-T) using gene expression profiling. This work was done in collaboration with Y. Shiloh's group, Sackler Faculty of Medicine, Tel-Aviv University, and QBI Enterprises [161].

6.7.1 The Ataxia-Telangiectasia Disease

A-T is a rare recessive disease. It is characterized by cerebellar degeneration, immunodeficiency, chromosomal instability, gonadal and thymic dysgenesis, premature

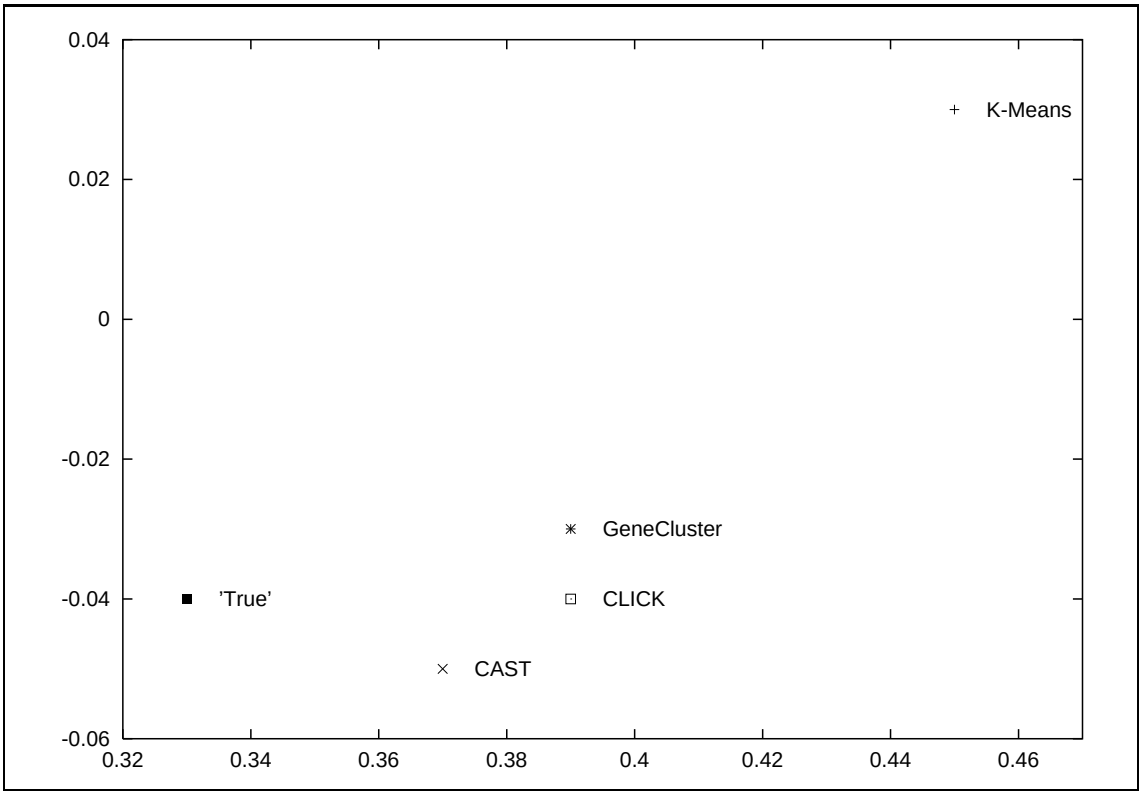


Figure 6.13: A comparison of homogeneity (x-axis) and separation (y-axis) values for all solutions of the yeast cell-cycle data of [177].

aging, marked predisposition to cancer, and acute sensitivity to ionizing radiation. A-T cells exhibit a broad defect in their response to ionizing radiation and radiomimetic chemicals [126]. The responsible gene, ATM, encodes a multifunctional protein kinase that controls an extensive array of signaling pathways, most notably those that are activated by DNA double-strand breaks [13]. Animal models of A-T were generated in several laboratories by inactivating the ATM gene in mice. These animals recapitulate the organismal and cellular phenotypes of A-T, and usually die with thymic lymphomas by the age of several months [13, 24, 59, 103, 192]. The development of the cerebellar phenotype is slower in ATM-deficient mice than in human A-T patients, and is subtly noticed only in certain strains with prolonged survival [24].

6.7.2 Experimental Design and Data Preprocessing

In order to obtain a global look at the A-T phenotype, we examined gene expression profiles in the thymus, cerebellum and cerebrum of wild type (WT) and ATM-deficient (ATM -/-) mice without treatment, and at two time points (30 and 120 min) after whole body X-irradiation. Microarrays containing 8,085 mouse expressed sequence tags (ESTs) were probed with cDNAs representing all 18 combinations of genotype, tissue, treatment and time point. Each microarray was hybridized simultaneously with a Cy5-labeled test probe representing one of the 18 combinations, and a Cy3-labeled reference probe representing a mixture of wild type brain, lung, liver, spleen, heart, and thymus. Only valid elements with high signal-to-background ratio (> 2.0), large spot hybridization cover area ($> 40\%$), and no missing values were included in the final analysis. Hybridization intensities were balanced (as described in GemTools 2.3.1 user manual), and the expression level of each element was divided by its level in the reference probe of the corresponding tissue. Valid data over 6 conditions (three measurements for each of the two genotypes) were obtained for 6,641 genes from the thymus samples, 7,754 genes in the cerebellum, and 7,106 genes in the cerebrum.

In each tissue, the expression of each gene over the six conditions was normalized by dividing by the expression level in the untreated wild type tissue and taking a base 2 logarithm of the result. The genes were ordered according to the maximum absolute value of their normalized levels. Genes in the top 5% of this list were categorized as significant responders (332 genes in the thymus, 387 genes in the cerebellum, and 355 genes in the cerebrum). They all showed at least a 1.75-fold change compared to untreated wild type tissue in at least one of the other 5 combinations of genotype and treatment. The responder genes were then subjected to cluster analysis. Clustering carried out simultaneously over the 3 tissues included a total of 977 responder genes: 745 of them had valid values over all 18 combinations and those genes were used for clustering.

6.7.3 Tissue Clustering

Hierarchical cluster analysis was first used to demonstrate the relative distance between overall expression patterns in the different tissues under the different conditions. Euclidean distances were used as the dissimilarity measure and were obtained

over all genes that met the above mentioned validity criteria. The average linkage hierarchical clustering method was applied using the SPSS statistical package.

Unexpectedly, expression profiles in unirradiated ATM-/- thymus and cerebellum were much closer to those of the corresponding irradiated wild type or ATM-/- tissues, while unirradiated wild type tissues stood out far from all other combinations (see Figure 6.14). The cerebral samples, on the other hand, showed the pattern initially expected: Untreated wild type and ATM-/- tissues were close to each other and far from all the irradiated samples. This result pointed to a sustained stress response in ATM-deficient thymus and cerebellum, the most affected organs in A-T patients. This response was less prominent in the cerebrum, whose degeneration in patients is considerably slower.

6.7.4 Gene Clustering

Cluster analysis was used to study this unusual constitutive response phenomenon. Responder genes whose expression level in any sample differed significantly from their basal expression in untreated wild type tissues were separated into individual clusters of genes with similar expression patterns (see Figures 6.15 and 6.16). Similarity between genes was computed as $\max\{1 - d/2, 0\}$, where d is the Euclidean distance between their normalized expression vectors.

When the clustering was carried out over the 3 tissues, CLICK divided the 745 responder genes into 28 clusters, leaving 87 genes as singletons. When the six thymus conditions were clustered, 332 responder genes fell into 9 clusters and 28 remained as singletons. In the cerebellum, 387 responders fell into 10 clusters, leaving 27 singletons. In the cerebrum 355 responders were divided into 8 clusters with 8 genes left as singletons.

The analysis shows that different sets of genes responded to the treatment in each tissue, with some exceptions (see Figure 6.15A). It is also of note that the cerebral and cerebellar responses to irradiation had already peaked at 30 min, in contrast to the slower thymic response. In spite of the marked differences between the general response patterns of the 3 tissues, the constitutive stress response of ATM-/- thymus and cerebellum could be seen in the 3-tissue clusters (see Figure 6.15A), and was more clearly observed in single tissue clusters (see Figure 6.16A,B). Significantly, the dominant pattern exhibited by the thymic and cerebellar clusters showed the

constitutive stress response, that is, the expression levels of untreated ATM-/- tissues were similar to those of the treated rather than the untreated wild type tissues. In contrast, the dominant pattern among the cerebral clusters showed symmetry between the radiation responses of the two genotypes, with the constitutive stress response being considerably less prominent than in the thymus and cerebellum (see Figure 6.15B). Of note, small groups of genes showed constitutively altered expression in ATM-/- tissues and no response to irradiation in wild type tissues; others showed similar basal activity in the two genotypes and responded to irradiation in only one of the genotypes (see Figures 6.15B and 6.16A,B).

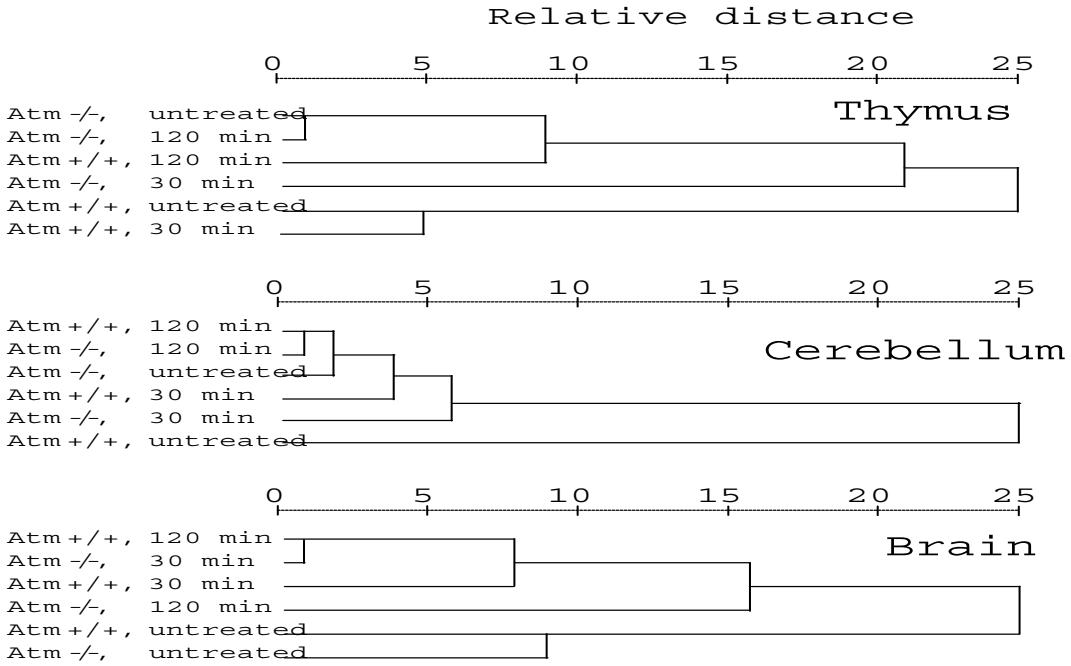


Figure 6.14: Hierarchical cluster analysis of expression profiles of unirradiated and irradiated mouse tissues. Genotypes and post-irradiation time points are indicated. The zero time point represents an unirradiated tissue. Note in the thymus and cerebellum the high similarity between unirradiated ATM-/- tissues and irradiated tissues of the two genotypes, and the great distance between the unirradiated wild type tissues and all other combinations. The small distance between untreated thymus and thymus at 30 min post-irradiation indicates that most of the response occurs in this tissue at a later time. In the cerebrum, on the other hand, unirradiated wild type and ATM-/- tissues are closer to each other and far from all irradiated samples.

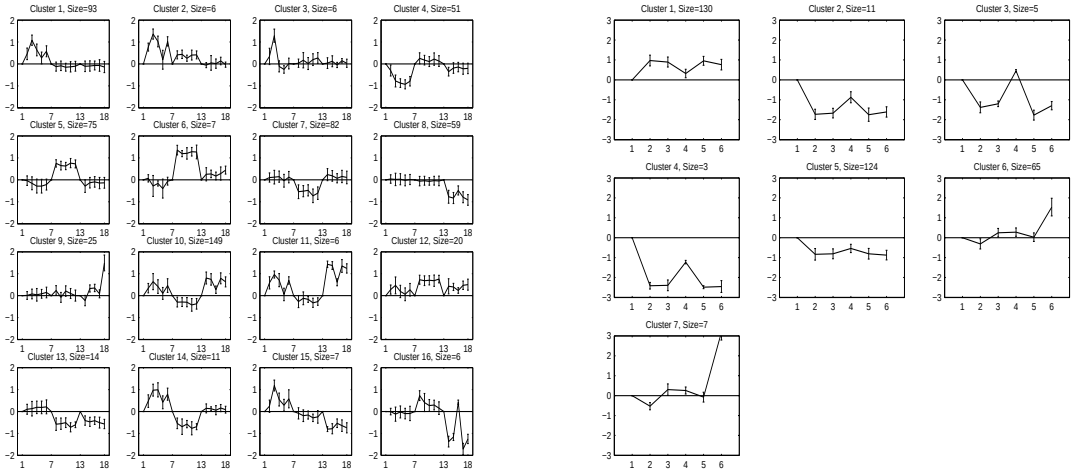
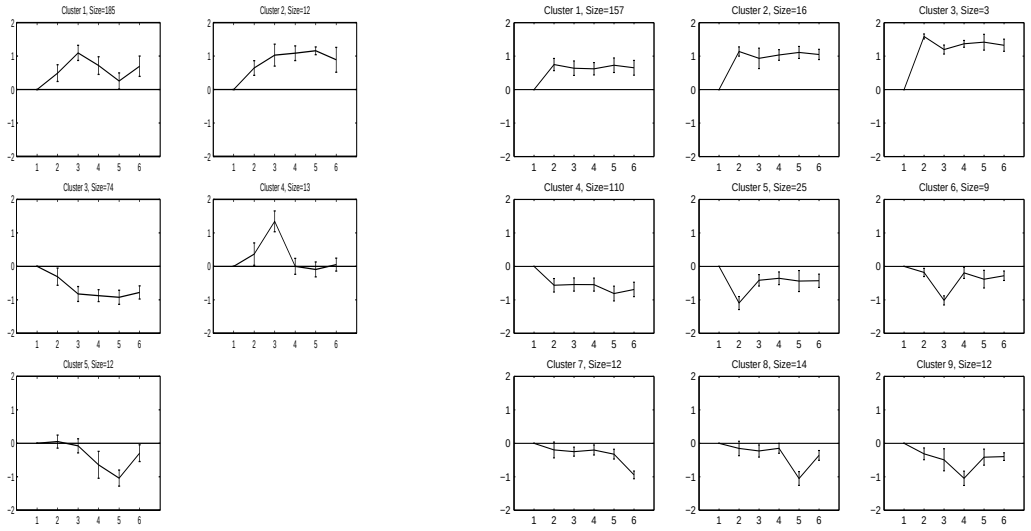


Figure 6.15: Results of CLICK on the A-T expression data. Shown for each cluster are the average and standard deviations of expression levels of the cluster's genes. A: Clusters containing at least 6 members that were obtained over the entire set of 3 tissues, genotypes, treatments and time points. Points 1-6 along the x-axis correspond to thymus. 1: WT, untreated. 2: WT, 30 min post irradiation. 3: WT, 120 min post irradiation. 4: ATM-/-, untreated. 5: ATM-/-, 30 min post irradiation. 6: ATM-/-, 120 min post irradiation. Points 7-12 and 13-18 represent the cerebellum and cerebrum, respectively, with the same order of genotypes and treatments. Clusters 1-9 represent genes whose expression was significantly modified in only one tissue. Clusters 10-12 represent genes that show similar expression pattern in two tissues. Clusters 13-16 include genes that show opposite patterns in different tissues. B: Cerebrum clusters containing at least 3 members. The points on the x-axis follow the same order of genotypes and treatments as the first six points in panel A. Clusters 1-4 show symmetric patterns for wild type and ATM-/- tissues, indicating a similar response of the two genotypes, while cluster 5 displays a certain degree of constitutive stress response. Clusters 2 and 4 can be interpreted also as showing mild constitutive response.



A B

Figure 6.16: Tissue-specific clusters for the thymus (A) and cerebellum (B). The points on the x-axis follow the same order of genotypes and treatments as in Figure 6.15B. Only clusters containing at least 3 genes are shown. The constitutive stress response in ATM-/- tissues is represented in thymic clusters 1-3 and cerebellar clusters 1-4. Thymic cluster 4 as well as cerebellar clusters 5 and 6 represent genes with normal basal activity in unirradiated ATM-/- tissue, that respond to the treatment only in the wild type tissue. Thymic cluster 5 and cerebellar clusters 7-9 depict genes that respond more vigorously to irradiation in ATM-/- tissues.

6.7.5 Discussion

The constitutive stress response in untreated ATM-/- thymus and cerebellum reflects an ongoing, but probably sub-optimal, effort to respond to constant, low-level damage throughout life. A considerable toll in energy and resources is most certainly paid by the cells to keep up this effort, and may contribute to the degenerative processes and premature aging in A-T patients.

This constitutive stress response corroborates recent findings of constitutive activation of several damage response pathways in cultured human A-T cells, such as the activation of the p53, p21 and Cdc2 proteins [71]. Lack of ATM may lead to stress, which activates enzymes redundant with ATM that mediate this constant stress response. An example of such an enzyme is ATR, a member of the family of PI3- kinase-related protein kinases that includes ATM. Following radiation damage,

ATR phosphorylates p53 on the same site as ATM, albeit at slower kinetics [186]. What could cause this tenuous stress in ATM-deficient cells? Among other activities the ATM protein is involved in the repair of double-strand breaks [113]. Lack of ATM should diminish the cell's capacity to repair the DNA breaks that are continuously being created by cellular metabolites or normal DNA processing. In the cerebellum, the high production of nitric oxide by the granule cells may be a steady source of DNA strand breakage, while in the thymus the maturation of the immune system genes is an important source of DNA discontinuities. Another source of constant genotoxic damage in ATM-deficient cells is elevated oxidative stress expressed as high levels of reactive oxygen species [166].

A considerable fraction of the ESTs with significant radiation or constitutive response were annotated using bioinformatic analysis. This annotation shows that the genes that participate in the acute response of wild type tissues and the constitutive response of ATM-/- tissues are involved in many cellular processes representing most aspects of cellular physiology, and are not limited to DNA repair and cell cycle checkpoint activation.

6.8 Identifying Regulatory Motifs

In this section we demonstrate the utility of clustering in general, and CLICK in particular, in identifying regulatory sequence motifs.

Global gene expression data enable the delineation of genetic regulatory networks via direct or indirect approaches. In the direct approach, the effects of activation or repression of a specific transcription factor (TF) on gene expression are monitored. This way, genes located downstream from p53 [200], BRCA1 [94] and C-myc [35] in their respective pathways, were identified. The indirect approach for the delineation of regulatory networks relies on the hypothesis that genes exhibiting similar expression patterns across a large panel of biological conditions are likely to share common regulatory elements in their promoter regions. In other words, co-expression is correlated with co-regulation [183, 199, 26]. The regulatory elements in the promoter region represent the "switches" that respond to signals from various cellular signaling pathways. The response can be either as part of the normal developmental program of the organism, or in response to external perturbations, stresses and alterations in physiological conditions. The binding of transcription factors to their

binding sites in the promoter region enhances (or represses) the transcription initiation complex recruitment and assembly on the basal promoter in the proximity of the transcription start site, thereby influencing transcription initiation.

The approach that aims to detect cis-regulatory TF's binding sites from co-expression comprises two steps: (I) Cluster analysis aimed at the identification of clusters of genes sharing similar expression patterns. (II) Sequence analysis, which searches for sequence patterns that are over-represented in upstream regions of members of the same cluster. The derivation of regulatory networks through the identification of common cis-regulatory elements shared by co-regulated genes was successfully demonstrated in yeast [32, 114, 177] and *Arabidopsis Thaliana* [137].

In order to test the utility of CLICK for motif identification, we have analyzed the dataset published recently by Jelinsky et al. [114]. In that experiment, expression levels of all 6,200 ORFs of the yeast *Saccharomyces Cerevisiae* were measured in order to study the cellular response to DNA damage. In total, gene expression profiles in 26 biological conditions were measured, including treatments with various DNA damaging agents at several time points and doses. 2,610 genes that changed by a factor of 3 or more in at least one condition were subjected to cluster analysis. The clustering reported in [114] consists of 18 clusters, obtained by GeneCluster. In comparison, CLICK identified 33 clusters with more than 10 members.

Once the clusters are identified, the motif finding algorithm is applied to promoter regions of the genes in each cluster. In this study, the search was performed on the 500 bases upstream to the ORFs' translation start sites. The analysis was performed using the AlignACE package [165, 109] as done in [114]. (As the motif finding software was recently modified, in addition to its application to CLICK's clustering, we also reapplied AlignACE to the clustering reported in [114].) AlignACE employs Gibbs sampling for detecting over-represented motifs in a target set of sequences. It utilizes the mononucleotide frequencies as genomic background.

To focus on regulatory motifs that potentially form the mechanistic basis for the observed co-expression, as well as to reduce false-positives, only motifs that exceed two score thresholds are reported. The first is an alignment score, which gauges the statistical significance of the identified motif over the genomic background. The second is a specificity score, which gauges how specific is the identified motif to promoters of genes in the cluster, relative to promoter regions of other genes in the genome [109]. In total, 26 significant motifs were identified in CLICK's clusters, and

Algorithm	#Motifs found	#Motifs verified
CLICK	26	17
GeneCluster	30	19
Both	17	13

Table 6.11: Statistics on motifs identified in CLICK's clusters, GeneCluster's clusters and in both clusterings. First column: Total number. Second column: Number of motifs with a match in the SCPD DB.

30 such motifs were identified in GeneCluster's clusters.

The identified motifs were matched against the SCPD database of experimentally verified yeast's TF binding sites [201]. Table 6.11 summarizes the number of motifs identified in each clustering as well as the number of motifs that had a match in the SCPD DB. For both clustering methods, more than 60% of the motifs had a verified TF binding site match, a fact that indicates the utility of this approach. In addition, motifs with no match to known binding site were detected as well. Each of these motifs forms a hypothesis that should be subjected to further biological research. Of the 26 motifs identified in CLICK's clusters, 17 were common with GeneCluster's motifs. Common motifs were identified using CompareAce (part of the AlignAce package), which calculates a similarity coefficient for pairs of input motifs. Motifs whose similarity exceeded a threshold of 0.7 were regarded as common. Table 6.12 lists those common motifs. It is interesting to note that the percent of verified common motifs is particularly high (more than 75%). Hence, the four common, unidentified motifs are more likely to be true, as they were obtained by two different methods.

6.9 Tissue classification

An important application of gene expression analysis is the classification of tissue types according to their gene expression profiles [81]. The power of gene expression analysis is directed at two main problems in this context: Cancer type classification and drug assessment. Several recent studies [8, 81, 7] demonstrated that gene expression data can be used in distinguishing between similar cancer types, whose distinction is hard otherwise, thereby allowing more accurate diagnosis and

CLICK	Consensus	Putative TF	GeneCluster
1_2 16% (37/237)	GGTGGCAAAW	UASPHR	14_2 30% (61/205)
2_1 63% (119/189)	RAAAAAAAAA	PHO2,SWI5	4_4 38% (52/136)
2_2 44% (83/189)	ATGTAYGGRTK	RAP1	1_2 52% (85/165)
2_3 30% (56/189) 2_11 23% (44/189) 9_3 87% (33/38)	RAAAAATTT AAAAAWTTT TGAAAAWTTTT	DAL82	2_2 65% (48/74) 4_2 61% (83/136)
2_7 15% (29/189)	NSYAGGCNGNR	RAP1*, BUF*	1_4 17% (28/165) 1_8 11% (18/165) 1_9 15% (25/165)
2_9 12% (23/189)	CYCNSCNRGNNGGA	MCM1*	1_5 15% (25/165)
3_3 21% (31/149)	YNCGGNSNNNSGGS	RAP1*	3_8 13% (23/175)
3_4 19% (28/149)	RRCCAATCAN	ABF1,BAF1*	3_2 21% (36/175)
3_6 12% (18/149) 3_7 8% (12/149)	GGCNGGGCRKC TSGGCGGCNNTT	URS1H	3_4 8% (14/175)
3_9 19% (28/149)	SGGNNNNNNNGGNNNGG	BUF *	3_6 16% (28/175)
2_8 15% (28/189)	SCNGCNNSCNGNNGSG	—	1_6 17% (28/165)
3_11 13% (19/149)	MNNNGGGNNNRNNNRNGGGR	—	6_7 25% (28/114)
3_21 9% (13/149)	NCCGNYGGNCCGR	—	3_8 13% (23/175)
26_2 90% (9/10)	AGGGGCGGNG	—	9_3 15% (23/150)

Table 6.12: Motifs identified in the clusters of both CLICK and GeneCluster. Left column: CLICK motif name. Each name is denoted by two numbers: The first is the cluster number and the second is the motif serial number in the cluster (AlignACE is capable of finding multiple motifs in a target set of sequences by an iterative masking procedure). The second part of the left column contains statistics on the prevalence of the motif in the cluster. Second column: the motif's consensus sequence. Third column: An experimentally verified TF which matches the consensus sequence. (For TFs denoted by an '*', there is one mismatch between the TF binding site consensus and the identified motif consensus. Otherwise, there is a perfect match.) Forth column: The corresponding GeneCluster's motifs (motifs with similarity coefficient above 0.7 to CLICK's consensus). Motifs found more than once are grouped together.

treatment. Drug assessment is aided by expression profiles before, during and after treatment: The profiles pinpoint drug responsive genes, and indicate treatment outcome [34].

Here we focus on the application of gene expression analysis in general, and cluster analysis in particular, to cancer classification. In gene expression studies of cancer, the data consist of expression levels of thousands of genes in several tissues. The tissues originate from two or more known classes, e.g., normal and tumor. The analysis aims at studying the typical expression profile of each class and predicting the classification of new unlabeled tissues. Classification methods employ supervised learning techniques, i.e., the known classifications of the tissues are used to guide the algorithm in building a classifier. These include support vector machines [16, 68], boosting [16], clustering [16], discriminant analysis [191] and weighted correlation [81]. Classification can be aided by first filtering the dataset from genes that are irrelevant to the required distinction. Several methods have been suggested to choose subsets of informative genes, on which improved classification accuracy can be attained [16, 54, 68, 191].

Ben-Dor et al. [16] have demonstrated the strength of clustering in classification problems. Key to their method is combining the labeling (known classification) information in the clustering process. Suppose we use a clustering algorithm with at least one free parameter. Given an unlabeled tissue, the clustering algorithm is applied repeatedly with different parameter values on the set of all tissues (known and unknown). Each solution is scored by its level of compatibility with the labeling information, and the best solution is chosen. The classification of each unlabeled tissue is then determined according to the distribution of classified tissues in the cluster containing it, assigning it the most represented class in this cluster.

The compatibility score for a clustering solution used by Ben-Dor et al. is simply the number of tissue pairs that are mates or non-mates in both the true labeling and the clustering solution. Singletons are considered as 1-member clusters for this computation. The clustering algorithm used in [16] was CAST with Pearson correlation as the similarity function.

We have studied two classification datasets using CLICK. The first dataset of Alon et al. [8] contains 62 samples of colon epithelial cells, collected from colon-cancer patients. They are divided into 40 'tumor' samples collected from tumors, and 22 'normal' samples collected from normal colon tissues of the same patients.

Dataset	Method	Correct	Incorrect	Unclassified
Colon	CLICK	87.1	12.9	0.0
	CAST	88.7	11.3	0.0
Leukemia	CLICK	94.4	2.8	2.8
	CAST	87.5	12.5	0.0

Table 6.13: A comparison of the classification quality of CLICK and CAST on the colon data of [8] and the leukemia data of [81]. For each dataset and clustering algorithm the percents of correct classifications (in the LOOCV iterations), incorrect classifications and unclassified elements are specified.

Of the ∼6,000 genes represented in the experiment, 2,000 genes were selected based on the confidence in the measured expression levels. The second dataset of Golub et al. [81] contains 72 leukemia samples. These samples are divided into 25 samples of acute myeloid leukemia (AML) and 47 samples of acute lymphoblastic leukemia (ALL). Of the ∼7,000 genes represented in the experiment, 3,549 were chosen based on their variability in the dataset.

The application of CLICK to classify these datasets enumerates several homogeneity parameters for CLICK, and chooses the solution which is most compatible with the given labels. We used the same similarity function and compatibility score as in [16]. A sample is not classified if it is either a singleton in the clustering obtained, or no class has a majority in the cluster assigned to that sample. In order to assess the performance of CLICK we employed the leave one out cross validation (LOOCV) technique, as done in [16]. According to this technique, one trial is performed for each tissue in the dataset. In the i-th trial, the algorithm tries to classify the i-th sample based on the known classifications of the rest of the samples. The average classification accuracy is thus computed. Table 6.13 presents a comparison between the classification based on CLICK and that of CAST, as reported in [16]. The results are comparable, with CAST performing slightly better on the colon dataset, and CLICK performing better on the leukemia dataset.

Next, we tested CLICK's utility in differentiating between two very similar types of cancer. We concentrated on part of the leukemia dataset composed of the 47 ALL samples only. For these samples an additional sub-classification into either T-cell or B-cell, is provided. An application of CLICK to this dataset resulted in an almost

Dataset	Size	Correct	Incorrect	Unclassified
Colon	2000	87.1	12.9	0.0
	50	90.3	9.7	0.0
Leukemia	3549	94.4	2.8	2.8
	50	97.2	2.8	0.0
ALL	3549	97.9	0.0	2.1
	50	97.9	2.1	0.0

Table 6.14: A summary of the classifications obtained by CLICK on the colon data of [8], the whole leukemia dataset of [81], and part of the leukemia dataset which contains ALL samples only. For each dataset classifications were performed with respect to the total number of genes, and with respect to the 50 most informative genes. The percents of correct classifications (in the LOOCV iterations), incorrect classifications and unclassified elements are specified.

perfect classification (see Table 6.14).

Finally we examined the influence of feature selection on the classification accuracy. To this end, we sorted the genes in each dataset according to the ratio of their between-sum-of-squares and within-sum-of-squares values, as suggested in [54]. This ratio is computed by the following formula:

$$\frac{BSS(g)}{WSS(g)} = \frac{\sum_{i=1,2} n_i (x_{g,i} - x_g)^2}{\sum_{i=1,2} \sum_{k \in i} (x_g^k - x_{g,i})^2}$$

Here i denotes the class number, n_i its size, k denotes the sample number, $x_{g,i}$ is the average expression level of gene g at class i, x_g is the average expression level of gene g, and x_g^k is the expression level of gene g at sample k. For each LOOCV iteration we chose the 50 genes with the highest value and performed the classification procedure on the reduced dataset which contained the expression levels of these 50 genes only. The results of this analysis are shown in Table 6.14. For both the colon and leukemia datasets the performance was improved on the reduced dataset.

6.10 The EXPANDER Clustering and Visualization Tool

We have developed a java-based graphical tool, called EXPANDER (EXPression ANalyzer and DisplayER), for gene expression analysis and visualization. This software provides graphical user interface to several clustering methods. It enables visualizing the raw expression data and the clustered data in several ways. In the following we outline the visualization options and demonstrate them on the yeast cell-cycle dataset of [177]. This dataset contains the expression levels of 698 yeast genes over 72 conditions. The reader is referred to Section 6.6 for a detailed description of this dataset. Figure 6.17 shows the initial screen when the tool is invoked.

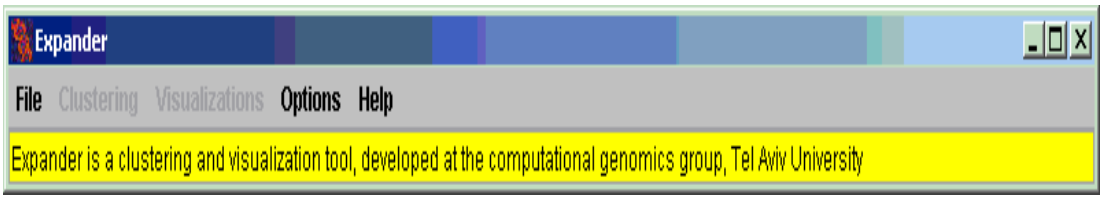


Figure 6.17: The EXPANDER initial screen.

6.10.1 Clustering Methods

EXPANDER implements several clustering algorithms including CLICK, K-means, hierarchical clustering and SOM. The user can specify the parameters of each algorithm: Homogeneity parameter for CLICK, number of clusters for K-means, type of linkage (single, average or complete) for hierarchical clustering, and the size of grid for SOM. In addition, the user can upload an external clustering solution.

6.10.2 Matrix Visualizations

EXPANDER includes visualizations for the expression matrix and the similarity matrix. In these visualizations the matrices are represented graphically by coloring each cell according to its content. Cells with neutral values are colored black, increasingly positive values with reds of increasing intensity, and increasingly negative values with greens of increasing intensity. Each matrix is shown in two ways: (1)

In its raw form; and (2) after reordering the rows of the matrix so that elements from the same cluster appear consecutively. (The columns are also reordered in the similarity matrix.) These visualizations are demonstrated in Figure 6.18.

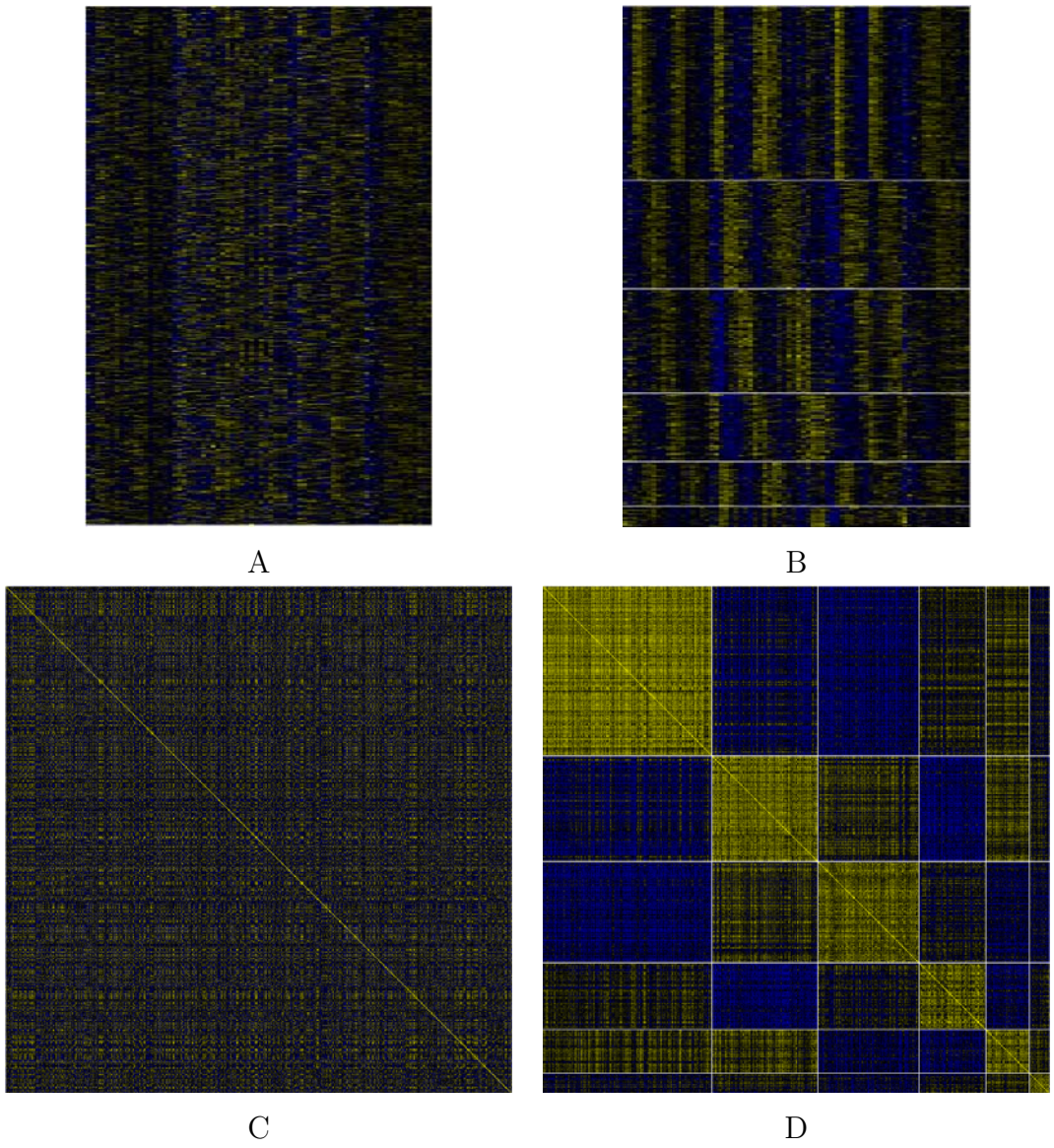


Figure 6.18: Matrix visualizations in EXPANDER. A: The raw yeast cell-cycle data matrix of [177]. B: The same data matrix after clustering the genes and reordering the raws accordingly. C: The similarity matrix. D: The similarity matrix after clustering the genes and reordering the raws and columns accordingly.

Ideally, in the reordered expression matrix we expect to see unique patterns for each cluster, while the reordered similarity matrix should be composed of light squares, each corresponding to a cluster, in dark background.

When using hierarchical clustering, the solution dendrogram is displayed along with the expression matrix, in which the genes are reordered according to the dendrogram (see Figure 6.19).

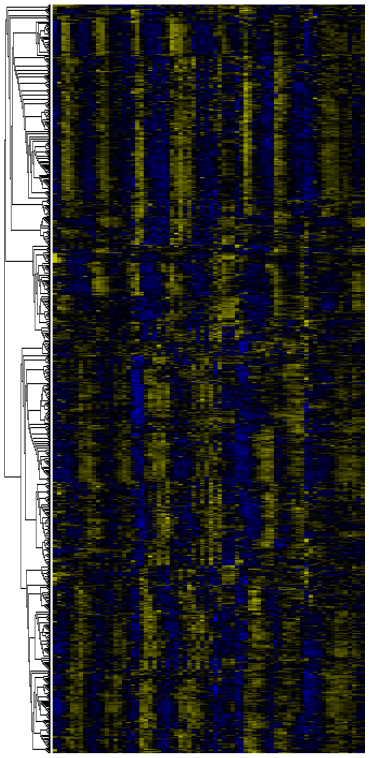
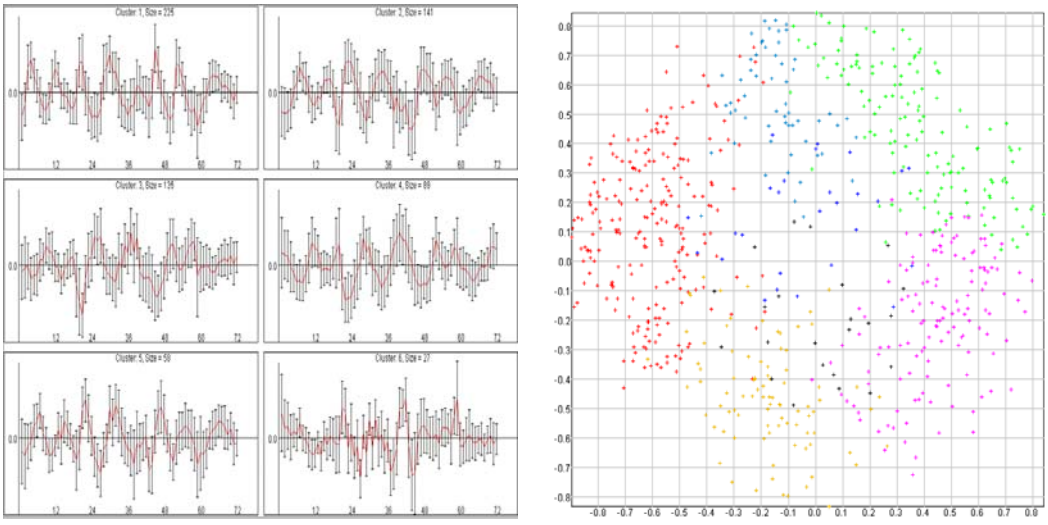


Figure 6.19: A dendrogram of the yeast cell-cycle data of [177] created by EXPANDER using average-linkage hierarchical clustering.

6.10.3 Clustering Visualizations

EXPANDER provides several visualizations of a clustering solution. A graphical overview of the solution is produced by showing for each cluster its mean expression pattern along with error bars indicating the standard deviation in each condition (see Figure 6.20A). Alternatively, for each single cluster a superposition of all the patterns of its members can be shown.

One other data visualization method provided in EXPANDER is principal component analysis (cf. [115]). This is a method for reducing data dimensionality by projecting high-dimensional data into a low-dimensional space spanned by the vectors that capture maximum variance of the data. In EXPANDER we reduce the data dimension to 2, by computing the two axes that capture maximum variance of the data. The projected data is visualized as points in the plane. Given a clustering solution the points are colored according to their assigned clusters, as depicted in Figure 6.20B.



A B

Figure 6.20: Clustering visualizations in EXPANDER. A: A presentation of CLICK's clustering of the yeast cell cycle data of [177]. For each cluster a sub-figure shows its mean pattern along with error-bars. B: principal component analysis of the data. Each gene is projected to the plane according to its first two principal components. Elements of each cluster are drawn using a separate color.

6.10.4 Functional Enrichment

As a simple aid for the interpretation of clustering results using biological knowledge, EXPANDER can quantify the enrichment of gene functions in a clustering solution. Given a functional annotation (an assignment of an attribute, such as functional category) of the genes in an input dataset, the abundant functional categories in each cluster are shown in a pie-chart. For each such category we compute its enrichment

in the cluster by computing a hypergeometric p-value. Formally, if a cluster contains k elements, r of which belong to a certain functional category of total size f, and there are n elements overall, then

$$p = \sum_{i=r}^{f} \frac{\binom{f}{i} \cdot \binom{n-f}{k-i}}{\binom{n}{k}} .$$

In CLICK's solution for the yeast cell-cycle dataset, the most enriched categories were transport (cluster 3, $p = 1.7 \cdot 10^{-7}$) and developmental processes (cluster 4, $p = 1.8 \cdot 10^{-6}$). The pie charts for these clusters are shown in Figure 6.21.

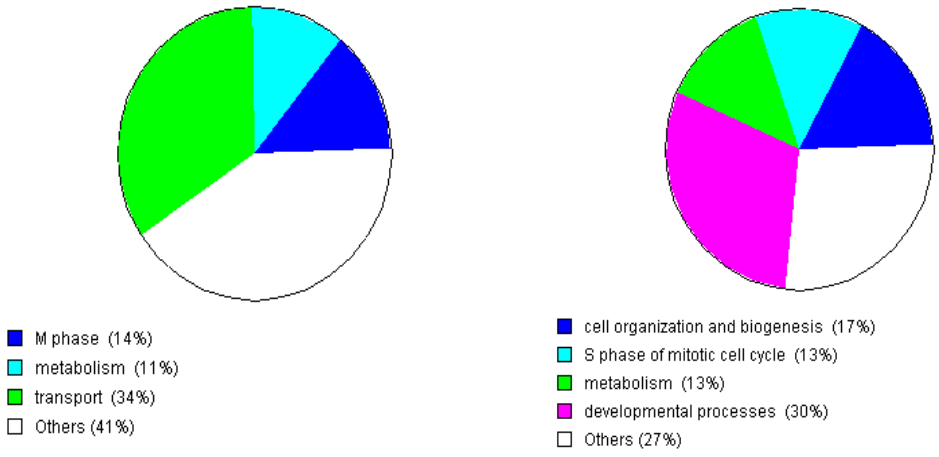


Figure 6.21: Functional enrichment pie charts for CLICK's clusters 3 (left) and 4 (right) computed on the yeast cell-cycle of [177]. Shown are functional categories containing at least 10% of the genes in a cluster. The most enriched categories are transport (in cluster 3, $p = 1.7 \cdot 10^{-7}$) and developmental processes (in cluster 4, $p = 1.8 \cdot 10^{-6}$).

Bibliography

[1] The chipping forecast. Special supplement to Nature Genetics Vol. 21, 1999.

[2] N. Abbas and L.K. Stewart. Biconvex graphs: ordering and algorithms. *Discrete Applied Mathematics*, 103:1–19, 2000.

[3] K. Abrahamson, R. Downey, and M. Fellows. Fixed-parameter intractability II. In *Proceedings of the 10th Symposium on Theoretical Aspects of Computer Science (STACS'93)*, volume 665 of *LNCS*, pages 374–385. Springer-Verlag, Berlin, 1993.

[4] R. Agarwala and D. Fernández-Baca. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. *SIAM Journal on Computing*, 23(6):1216–1224, 1994.

[5] A. Agrawal, P. Klein, and R. Ravi. Cutting down on fill using nested dissection: provably good elimination orderings. In A. George, J.R. Gilbert, and J.W.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 31–55. Springer, 1993.

[6] A.V. Aho, Y. Sagiv, T.G. Szymanski, and J.D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, 10(3):405–421, 1981.

[7] A.A. Alizadeh, M.B. Eisen, R.E. Davis, C. Ma, I.S. Lossos, A. Rosenwald, J.C. Boldrick, H. Sabet, T. Tran, X. Yu, J.I. Powell, L. Yang, G.E. Marti, T. Moore, J. Hudson, L. Lu, D.B. Lewis, R. Tibshirani, G. Sherlock, W.C. Chan, T.C. Greiner, D.D. Weisenburger, J.O. Armitage, R. Warnke, and L.M. Staudt. Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling. *Nature*, 403(6769):503–511, 2000.

[8] U. Alon, N. Barkai, D.A. Notterman, G. Gish, S. Ybarra, D. Mack, and A.J. Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. *Proc. Natl. Acad. Sci. USA*, 96:6745–6750, June 1999. `http://www.sph.uth.tmc.edu/hgc`.

[9] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–10, 1990.

[10] T. Asano. An application of duality to edge-deletion problems. *SIAM Journal on Computing*, 16(2):312–331, 1987.

[11] T. Asano and T. Hirata. Edge-deletion and edge-contraction problems. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 245–254, San Francisco, California, 5–7 May 1982.

[12] G. Ball and D. Hall. A clustering technique for summarizing multivariate data. *Behaviorial Sciences*, 12(2):153–155, 1967.

[13] C. Barlow, S. Hirotsune, R. Paylor, M. Liyanage, M. Eckhaus, F. Collins, Y. Shiloh, J.N. Crawley, T. Ried, D. Tagle, and A. Wynshaw-Boris. ATM-deficient mice: a paradigm of ataxia-telangiectasia. *Cell*, 86:159–171, 1996.

[14] A. Ben-Dor, 1996. Private communication.

[15] A. Ben-Dor, 1999. Private communication.

[16] A. Ben-Dor, L. Bruhn, N. Friedman, I. Nachman, M. Schummer, and Z. Yakhini. Tissue classification with gene expression profiles. *Journal of Computational Biology*, 7(3/4):559–583, 2000.

[17] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6(3/4):281–297, 1999.

[18] C. Benham, S. Kannan, M. Paterson, and T.J. Warnow. Hen's teeth and whale's feet: generalized characters and their compatibility. *Journal of Computational Biology*, 2(4):515–525, 1995.

[19] D.A. Benson, M.S. Boguski, D.J. Lipman, J. Ostell, B.F. Ouellette, B.A. Rapp, and D.L. Wheeler. Genbank. *Nucleic Acids Res.*, 27(1):12–17, 1999.

[20] H. L. Bodlaender, M. R. Fellows, and M. T. Hallet. Beyond NP-Completeness for problems of bounded width: Hardness for the W hierarchy. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, pages 449–458. ACM Press, New York, 1994.

[21] H.L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[22] H.L. Bodlaender and B. de Fluiter. On intervalizing k-colored graphs for DNA physical mapping. *Discrete Applied Math.*, 71:55–77, 1996.

[23] H.L. Bodlaender, M.R. Fellows, M.T. Hallett, H.T. Wareham, and T.J. Warnow. The hardness of perfect phylogeny, feasible register assignment and other problems on thin colored graphs. *Theoretical Computer Science*, 244(1–2):167–188, 2000.

[24] P.R. Borghesani, F.W. Alt, A. Bottaro, L. Davidson, S. Aksoy, G.A. Rathbun, T.M. Roberts, W. Swat, R.A. Segal, and Y. Gu. Abnormal development of purkinje cells and lymphocytes in ATM mutant mice. *Proc. Natl. Acad. Sci. USA*, 97:3336–3341, 2000.

[25] A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes - a Survey*. SIAM, Philadelphia, 1999.

[26] A. Brazma and J. Vilo. Gene expression data analysis. *FEBS Letters*, 480:17–24, 2000.

[27] J.R. Bunch and D.J. Rose, editors. *Sparse Matrix Computations*. Academic Press, 1976.

[28] L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58:171–176, 1996.

[29] J.H. Camin and R.R. Sokal. A method for deducing branching sequences in phylogeny. *Evolution*, 19:409–414, 1965.

[30] A.V. Carrano. Establishing the order of human chromosome-specific DNA fragments. In A.D. Woodhead and B.J. Barnhart, editors, *Biotechnology and the Human Genome*, pages 37–50. Plenum Press, New York, 1988.

[31] C.S. Chekuri, A.V. Goldberg, D.R. Karger, M.S. Levine, and C. Stein. Experimental study of minimum cut algorithms. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333, 1997.

[32] R.J. Cho, M.J. Campbell, E.A. Winzeler, L. Steinmetz, A. Conway, L. Wodicka, T.G. Wolfsberg, A.E. Gabrielian, D. Landsman, D.J. Lockhart, and R.W. Davis. A genome-wide transcriptional analysis of the mitotic cell cycle. *Mol. Cell*, 2:65–73, 1998.

[33] K. Cirino, S. Muthukrishnan, N.S. Narayanaswamy, and H. Ramesh. Graph editing to bipartite interval graphs: Exact and asymptotic bounds. In *Proceedings of the Seventeenth Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *LNCS*, pages 37–53, 1997.

[34] P.A. Clarke, M. George, D. Cunningham, I. Swift, and P. Workman. Analysis of tumor gene expression following chemotherapeutic treatment of patients with bowel cancer. In *Proc. Nature Genetics Microarray Meeting*, page 39, 1999.

[35] H. Coller, C. Gradori, P. Tamayo, T. Colbert, E. Lander, R. Eisenman, and T.R. Golub. Expression analysis with oligonucleotide reveals that C-Myc regulates genes involved in growth, cell-cycle, signaling and adhesion. *Proc. Natl. Acad. Sci. USA*, 97(7):3260–3265, 2000.

[36] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[37] R.M. Cormack. A review of classification (with discussion). *J. Royal Statistical Society, Series A*, 134:321–367, 1971.

[38] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.

[39] D.G. Corneil, H. Kim, S. Natarajan, S. Olariu, and A.P. Sprague. Simple linear time recognition of unit interval graphs. *Inf. Proc. Letts.*, 55:99–104, 1995.

[40] D.G. Corneil, H. Lerchs, and L. Stewart Burlingham. Complement reducible graphs. *Discrete Applied mathematics*, 3:163–174, 1981.

[41] D.G. Corneil, Y. Perl, and L. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985.

[42] A. Cournier and M. Habib. A new linear algorithm for modular decomposition. In *Proceedings of Colloquium on Trees in Algebra and Programming (CAAP)*, volume 787 of *LNCS*, pages 68–84, 1994.

[43] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 24th Nat. Conf. ACM*, pages 157–172, 1969.

[44] E. Dahlhaus, J. Gustedt, and R.M. McConnell. Efficient and practical modular decomposition. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 26–35, 1997.

[45] M.H. DeGroot. *Probability and Statistics*. Addison-Wesley, 1989.

[46] X. Deng, P. Hell, and J. Huang. Linear time representation algorithms for proper circular arc graphs and proper interval graphs. *SIAM Journal on Computing*, pages 390–403, 1996.

[47] L. Dollo. Le lois de l'évolution. *Bulletin de la Societé Belge de Géologie de Paléontologie et d'Hydrologie*, 7:164–167, 1893.

[48] R.G. Downey and M.R. Fellows. Parameterized computational feasibility. In K. Ambos-Spies, S. Homer, and U. Schoning, editors, *Complexity Theory: Current Research*, pages 166–191. Cambridge University Press, New York, 1993.

[49] R.G. Downey and M.R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1997.

[50] R. Drmanac, S. Drmanac, I. Labat, R. Crkvenjakov, A. Vicentic, and A. Gemmell. Sequencing by hybridization: towards an automated sequencing of one milion M13 clones arrayed on membranes. *Electrophoresis*, 13:566–573, 1992.

[51] R. Drmanac, G. Lennon, S. Drmanac, I. Labat, R. Crkvenjakov, and H. Lehrach. Partial sequencing by oligohybridization: Concept and applications in genome analysis. In C. Cantor and H. Lim, editors, *Proceedings of the first international conference on electrophoresis supercomputing and the human genome*, pages 60–75. World Scientific, 1991.

[52] S. Drmanac and R. Drmanac. Processing of cDNA and genomic kilobase-size clones for massive screening mapping and sequencing by hybridization. *Biotechniques*, 17:328–336, 1994.

[53] S. Drmanac, N.A. Stavropoulos, I. Labat, J. Vonau, B. Hauser, M.B. Soares, and R. Drmanac. Gene-representing cDNA clusters defined by hybridization of 57419 clones from infant brain libraries with short oligonucleotide probes. *Genomics*, 37:29–40, 1996.

[54] S. Dudoit, J. Fridlyand, and T.P. Speed. Comparison of discrimination methods for the classification of tumors using gene expression data. Technical Report 576, Dept. of Statistics, university of California, Berkeley, 2000.

[55] I. Duff, editor. *Sparse matrices and their uses*. Academic Press, 1981.

[56] M.B. Eisen and P.O. Brown. DNA arrays for analysis of gene expression. In *Methods in Enzymology*, volume 303, pages 179–205. 1999.

[57] M.B. Eisen, P.T. Spellman, P.O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proc. Natl. Acad. Sci. USA*, 95:14863–14868, 1998. `http://rana.lbl.gov/`.

[58] E.S. El-Mallah and C.J. Colbourn. The complexity of some edge deletion problems. *IEEE Transactions on Circuits and Systems*, 35(3):354–362, 1988.

[59] A. Elson, Y. Wang, C.J. Daugherty, C.C. Morton, F. Zhou, J. Campos-Torres, and P. Leder. Pleiotropic defects in ataxia-telangiectasia protein-deficient mice. *Proc. Natl. Acad. Sci. USA*, 93(23):13084–13089, 1996.

[60] D. Eppstein, Z. Galil, G.F. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997.

[61] B. Everitt. *Cluster analysis*. Edward Arnold, London, third edition, 1993.

[62] M.R. Fellows, M.T. Hallet, and H.T. Wareham. DNA physical mapping: Three ways difficult. In *Proc. First European Symp. on Algorithms (ESA '93)*, volume 726 of *LNCS*, pages 157–168. Springer, 1993.

[63] J. Felsenstein. *Inferring Phylogenies*. Sinaur Associates, Sunderland, Massachusetts, 2002. In press.

[64] S.P. Fodor, R.P. Rava, X.C. Huang, A.C. Pease, C.P. Holmes, and C.L. Adams. Multiplexed biochemical assays with biological chips. *Nature*, 364:555–556, 1993.

[65] L.R. Foulds and R.L. Graham. The Steiner problem in phylogeny is NP-complete. *Advances in Applied Mathematics*, 3:43–49, 1982.

[66] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[67] T. Fuchs, B. Malecova, C. Linhart, R. Sharan, M. Khen, R. Herwig, D. Shmulevitz, R. Elkon, M. Steinfath, J. O'Brien, U. Radelof, H. Lehrach, D. Lancet, and R. Shamir. Defog: A practical scheme for deciphering families of genes. *Genomics*. To appear.

[68] T.S. Furey, N. Cristianini, N. Duffy, D.W. Bendarski, M. Schummer, and D. Haussler. Support vector machine classification and validation of cancer tissue samples using microarray expression data. *Bioinformatics*, 16:906–914, 2000.

[69] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, 1979.

[70] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[71] M. Gatei, D. Shkedy, K.K. Khanna, T. Uziel, Y. Shiloh, T.K. Pandita, M.F. Lavin, and G. Rotman. Ataxia-telangiectasia: chronic activation of damage-responsive functions is reduced by alpha-lipoic acid. *Oncogene*, 20(3):289–94, 2001.

[72] A. George. Nested dissection of a regular finite element mesh. *SIAM J. on Numerical Analysis*, 10:345–367, 1973.

[73] A. George, J.R. Gilbert, and J.W.H. Liu, editors. *Graph Theory and Sparse Matrix Computation*. Springer, 1993.

[74] A. George and J.W. Liu. *Computer solution of large sparse positive definite systems*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[75] A. George and J.W. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[76] G. Getz, E. Levine, E. Domany, and M.Q. Zhang. Super-paramagnetic clustering of yeast gene expression profiles. *Physica*, A279:457, 2000.

[77] D. Ghosh and A.M. Chinnaiyan. Mixture modelling of gene expression data from microarray experiments. *Bioinformatics*, 18:275–286, 2002.

[78] J.R. Gilbert. Some nested dissection order is near optimal. *Inf. Proc. Letts.*, 26:325–328, 1988.

[79] M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995.

[80] P.W. Goldberg, M.C. Golumbic, H. Kaplan, and R. Shamir. Four strikes against physical mapping of DNA. *Journal of Computational Biology*, 2(1):139–152, 1995.

[81] T.R. Golub, D. Slonim, P. Tamayo, C.M. Huard, J.M. Caasenbeek, H. Coller, M. Loh, J. Downing, M. Caligiuri, C. Bloomfield, and E. Lander. Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring. *Science*, 286:531–537, 1999. `http://www-genome.wi.mit.edu/cancer/`.

[82] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[83] M.C. Golumbic. Matrix sandwich problems. *Linear algebra and its applications*, 277:239–251, 1998.

[84] M.C. Golumbic, H. Kaplan, and R. Shamir. On the complexity of DNA physical mapping. *Advances in Applied Mathematics*, 15:251–261, 1994.

[85] M.C. Golumbic, H. Kaplan, and R. Shamir. Graph sandwich problems. *Journal of Algorithms*, 19:449–473, 1995.

[86] M.C. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: A graph-theoretic approach. *Journal of the ACM*, 40:1108–1133, 1993.

[87] D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21:19–28, 1991.

[88] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[89] S.L. Hakimi, E.F. Schmeichel, and N.E. Young. Orienting graphs to optimize reachability. *Information Processing Letters*, 63(5):229–235, 1997.

[90] P.L. Hammer, T. Ibaraki, and U.N. Peled. Threshold numbers and threshold completions. In P. Hansen, editor, *Studies on Graphs and Discrete Programming*, pages 125–145. North-Holland, 1981.

[91] P.L. Hammer and B. Simeone. The splittance of a graph. *Combinatorica*, 1:275–284, 1981.

[92] P. Hansen and B. Jaumard. Cluster analysis and mathematical programming. *Mathematical Programming*, 79:191–215, 1997.

[93] J. Hao and J.B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994.

[94] D.P. Harkin, J. Bean, D. Miklos, Y. Song, V. Maheswaram, J. Oliver, and D. Haber. Induction of GADD45 and JNK/SAPK-dependent apoptosis following inducible expression of BRCA1. *Cell*, 97:575–586, 1999.

[95] C.A. Harrington, C. Rosenow, and J. Retief. Monitoring gene expression using DNA microarrays. *Curr. Opin. Microbiol.*, 3(3):285–291, 2000.

[96] J.A. Hartigan. *Clustering Algorithms*. John Wiley and Sons, 1975.

[97] E. Hartuv, A. Schmitt, J. Lange, S. Meier-Ewert, H. Lehrach, and R. Shamir. An algorithm for clustering cDNA fingerprints. *Genomics*, 66(3):249–256, 2000.

[98] E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(200):175–181, 2000.

[99] P. Hell, R. Shamir, and R. Sharan. A fully dynamic algorithm for recognizing and representing proper interval graphs. *SIAM Journal on Computing*, 31(1):289–305, 2002. A preliminary version appeared in the Proceedings of the

Seventh Annual European Symposium on Algorithms (ESA), volume 1643 of LNCS, pages 527–539. Springer, 1999.

[100] M. Henzinger and M. Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22:351–362, 1998.

[101] M. Henzinger, V. King, and T.J. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24:1–13, 1999.

[102] R. Herwig, A.J. Poustka, C. Meuller, H. Lehrach, and J. O'Brien. Large-scale clustering of cDNA-fingerprinting data. *Genome Research*, 9(11):1093–1105, 1999.

[103] K.H. Herzog, M.J. Chong, M. Kapsetaki, J.I. Morgan, and P.J. McKinnon. Requirement for ATM in ionizing radiation-induced cell death in the developing central nervous system. *Science*, 280:1089, 1998.

[104] L.J. Heyer, S. Kruglyak, and S. Yooseph. Exploring expression data: identification and analysis of coexpressed genes. *Genome Research*, 9(11):1106–1115, November 1999.

[105] D. S. Hochbaum, editor. *Approximation Alogrithms for NP-Hard Problems*. PWS Publishing, Boston, 1997.

[106] D.S. Hochbaum. Approximating clique and biclique problems. *Journal of Algorithms*, 29(1):174–200, 1998.

[107] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001.

[108] W.-L. Hsu. On-line recognition of interval graphs in $O(m + n \log n)$ time. *Lecture Notes in Computer Science*, 1120:27–38, 1996.

[109] J.D. Hughes, P.E. Estep, S. Tavazoie, and G.M. Church. Computational identification of cis-regulatory elements associated with groups of functionally related genes in *Saccharomyces Cerevisiae*. *J. Mol. Biol.*, 296:1205–1214, 2000. `http://atlas.med.harvard.edu/`.

[110] L. Ibarra. Fully dynamic algorithms for chordal graphs. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 923–924, 1999.

[111] L. Ibarra. A fully dynamic algorithm for recognizing interval graphs using the clique-separator graph. Technical report, University of Victoria, Victoria, Canada, 2001.

[112] V.R. Iyer, M.B. Eisen, D.T. Ross, G. Schuler, T. Moore, J.C.F. Lee, J.M. Trent, L.M. Staudt, J. Hudson Jr., M.S. Boguski, D. Lashkari, D. Shalon, D. Botstein, and P.O. Brown. The transcriptional program in the response of human fibroblasts to serum. *Science*, 283(1):83–87, 1999.

[113] P.A. Jeggo, A.M. Carr, and A.R. Lehmann. Splitting the ATM: distinct repair and checkpoint defects in ataxia-telangiectasia. *Trends Genet.*, 14(8):312–316, 1998.

[114] S.A. Jelinsky, P. Estep, Q.M. Church, and L.D. Samson. Regulatory networks revealed by transcriptional profiling of damaged *Saccharomyces Cerevisiae* cells: Rpn4 links base excision repair with proteasomes. *Mol. Cell Biol.*, 20(21):8157–8167, 2000.

[115] R.A. Johnson and D.W. Wichern. *Applied multivariate statistical analysis*. Prentice-Hall Inc., 1982.

[116] S. Kannan and T. Warnow. A fast algorithm for the computation and enumeration of perfect phylogenies. *SIAM Journal on Computing*, 26(6):1749–1763, 1997.

[117] H. Kaplan and R. Shamir. Bounded degree interval sandwich problems. *Algorithmica*, 24 (2):96–104, 1999.

[118] H. Kaplan, R. Shamir, and R.E. Tarjan. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal on Computing*, 28(5):1906–1922, 1999.

[119] T. Kashiwabara and T. Fujisawa. An NP-complete problem on interval graphs. In *Proc. 12th IEEE International Symposium on Circuits and Systems*, pages 82–83, 1979.

[120] M.A. Kerr, M. Martin, and G.A. Churchill. Analysis of variance for gene expression microarray data. *Journal of Computational Biology*, 7(6):819–837, 2000.

[121] B. Klinz, R. Rudolf, and G.J. Woeginger. Permuting matrices to avoid forbidden submatrices. *Discrete applied mathematics*, 60:223–248, 1995.

[122] T. Kloks. *Treewidth*. PhD thesis, Dept. of Computer Science, Utrecht University, 1993.

[123] T. Kohonen. *Self-Organizing Maps*. Springer, Berlin, 1997.

[124] A. Krause, J. Stoye, and M. Vingron. The SYSTERS protein sequence cluster set. *Nucleic Acids Res.*, 28(1):270–272, 2000.

[125] G.N. Lance and W.T. Williams. A general theory of classification sorting strategies. 1. hierarchical systems. *The Computer Journal*, 9:373–380, 1967.

[126] M.F. Lavin and Y. Shiloh. The genetic defect in ataxia-telangiectasia. *Ann. Rev. Immunol.*, 15:177–202, 1997.

[127] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[128] C.G. Lekkerkerker and J.Ch. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundam. Math.*, 51:45–64, 1962.

[129] G.S. Lennon and H. Lehrach. Hybridization analysis of arrayed cDNA libraries. *Trends Genet*, 7:60–75, 1991.

[130] J.M. Lewis and M. Yannakakis. The node deletion problem for hereditary properties is NP-complete. *J. Comput. Sys. Sci.*, 20:219–230, 1980.

[131] R.J. Lipshutz, S.P.A. Fodor, T.R. Gingeras, and D.J. Lockhart. High density synthetic oligonucleotide arrays. *Nature Genetics Supplement*, 21:20–24, 2000.

[132] D.J. Lockhart and E.A. Winzeler. Genomics, gene expression and DNA arrays. *Nature*, 405(6788):827–836, 2000.

[133] P.J. Looges and S. Olariu. Optimal greedy algorithms for indifference graphs. *Comput. Math. Appl.*, 25:15–25, 1993.

[134] C. Lund and M. Yannakakis. The approximation of maximum subgraph problems. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of International Conference on Automata, Languages and Programming (ICALP)*, volume 700 of *LNCS*, pages 40–51, Berlin, Germany, 1993. Springer.

[135] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1965.

[136] N.V.R. Mahadev and U.N. Peled. *Threshold Graphs and Related Topics*. Elsevier, North-Holland, 1995. Annals of Discrete Mathematics, Vol. 56.

[137] K. Maleck, A. Levine, T. Eulgem, A. Morgan, J. Schmid, K.A. Lawton, J.L. Dangl, and R.A. Dietrich. The transcriptome of *Arabidopsis Thaliana* during systematic acquired resistance. *Nature Genetics*, 26:403–410, 2000.

[138] F. Margot. Some complexity results about threshold graphs. *Discrete Applied Mathematics*, 49, 1994.

[139] A. Marshall and J. Hodgson. DNA chips: an array of possibilities. *Nature Biotech.*, 16:27–31, 1998.

[140] R.M. McConnell and J.P. Spinrad. Ordered vertex partitioning. *Discrete Mathematics and Theoretical Computer Science*, 4(1):45–60, 2000.

[141] F. R. McMorris, T. J. Warnow, and T. Wimer. Triangulating vertex colored graphs. *SIAM J. Discrete Math.*, 7:296–306, 1994.

[142] C.A. Meecham and G.F. Estabrook. Compatibility methods in systematics. *Annual Review of Ecology and Systematics*, 16:431–446, 1985.

[143] S. Meier-Ewert, J. Lange, H. Gerst, R. Herwig, A. Schmitt, J. Freund, T. Elge, R. Mott, B. Herrmann, and H. Lehrach. Comparative gene expression profiling by oligonucleotide fingerprinting. *Nucleic Acids Research*, 26(9):2216–2223, 1998.

[144] A. Milosavljevic, Z. Strezoska, M. Zeremski, D. Grujic, T. Paunesku, and R. Crkvenjakov. Clone clustering by hybridization. *Genomics*, 27:83–89, 1995.

[145] P.B. Miltersen, S. Subramanian, R. Tamassia, and J. Vitter. Complexity models for incremental computation. *Theoretical Computer Science*, 130:203–236, 1994.

[146] B. Mirkin. *Mathematical Classification and Clustering*. Kluwer, 1996.

[147] J.H. Muller and J. Spinrad. Incremental modular decomposition. *Journal of the ACM*, 36(1):1–19, 1989.

[148] A. Natanzon. Complexity and approximation of some graph modification problems. Master's thesis, School of Computer Science, Tel Aviv University, Tel-Aviv, 1999.

[149] A. Natanzon, R. Shamir, and R. Sharan. A polynomial approximation algorithm for the minimum fill-in problem. *SIAM Journal on Computing*, 30(4):1067–1079, 2000. A preliminary version appeared in the Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC), pages 41–47. ACM Press, 1998.

[150] A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113:109–128, 2001. A preliminary version appeared in the Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science (WG), volume 1665 of LNCS, pages 65–77. Springer, 1999.

[151] M. Nikaido, A. P. Rooney, and N. Okada. Phylogenetic relationships among cetartiodactyls based on insertions of short and long interspersed elements: Hippopotamuses are the closest extant relatives of whales. *Proc. Natl. Acad. Sci. USA*, 96:10261–10266, 1999.

[152] T. Ohtsuki. A fast algorithm for finding an optimal ordering for vertex elimination on a graph. *SIAM J. Computing*, 5:133–145, 1976.

[153] T. Ohtsuki, L. K. Cheung, and T. Fujisawa. Minimal triangulation of a graph and optimal pivoting order in a sparse matrix. *Journal of Math. Anal. Appl.*, 54:622–633, 1976.

[154] C. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *J. of Computer and System Science*, 43:425–440, 1991.

[155] I. Pe'er, R. Shamir, and R. Sharan. Incomplete directed perfect phylogeny. In *Proceedings of the Eleventh Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1848 of *LNCS*, pages 143–153. Springer, 2000.

[156] I. Pe'er, R. Shamir, and R. Sharan. On the generality of phylogenies from incomplete directed characters. In *Proceedings of the Eighth Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 2368 of *LNCS*, pages 358–367, 2002.

[157] A.J. Poustka, R. Herwig, A. Krause, S. Hennig, S. Meier-Ewert, and H. Lehrach. Toward the gene catalogue of sea urchin development: The construction and analysis of an unfertilized egg cDNA library highly normalized by oligonucleotide fingerprinting. *Genomics*, 59:122–133, 1999.

[158] W.J. Le Quesne. The uniquely evolved character concept and its cladistic application. *Systematic Zoology*, 23:513–517, 1974.

[159] G. Ramsay. DNA chips: State-of-the art. *Nature Biotech.*, 16:40–44, 1998.

[160] S. Rao and A.W. Richa. New approximation techniques for some ordering problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 211–218, 1998.

[161] S. Rashi-Elkeles, A. Regev, R. Elkon, R. Sharan, N. Zak, L. Brodsky, A. Kamsler, A. Leontovitch, N. Weisman, D. Leshkovitz, O. Mor, A Barzilai, E. Feinstein, R. Shamir, Y. Shiloh, and A. Bar-Shira. Constitutive response to genotoxic stress in a mouse model of ataxia-telangiectasia. Technical report, Sackler School of Medicine, Tel-Aviv University, Tel-Aviv, 2000.

[162] J.A. Rice. *Mathematical Statistics and Data Analysis*. Wadsworth, California, second edition, 1995.

[163] F.S. Roberts. Indifference graphs. In F. Harary, editor, *Proof Techniques in Graph Theory*, pages 139–146. Academic Press, New York, 1969.

[164] J.D. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R.C. Reed, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, N.Y., 1972.

[165] F.P. Roth, J.D. Hughes, P.W. Estep, and G.M. Church. Finding DNA regulatory motifs within unaligned noncoding sequences clustered by whole-genome mRNA quantitation. *Nature Biotech.*, 16:939–945, 1998.

[166] G. Rotman and Y. Shiloh. Ataxia-telangiectasia: is ATM a sensor of oxidative damage and stress? *BioEssays*, 19:911–917, 1997.

[167] M. Schena. Genome analysis with gene expression microarrays. *Bioessays*, 18:427–431, 1996.

[168] M. Schena, D. Shalon, R. Heller, A. Chai, P.O. Brown, and R.W. Davis. Parallel human genome analysis: microarray-based expression monitoring of 1000 genes. *Proc. Natl. Acad. Sci. USA*, 93:10614–9, 1996.

[169] U. Schendel. *Sparse matrices: numerical aspects with applications for scientists and engineers*. Ellis Horwood, 1989.

[170] G.D. Schuler. Pieces of the puzzle: expressed sequence tags and the catalog of human genes. *J. Mol. Med.*, 75(10):694–698, 1997.

[171] R. Shamir and R. Sharan. Algorithmic approaches to clustering gene expression data. In T. Jiang, T. Smith, Y. Xu, and M.Q. Zhang, editors, *Current Topics in Computational Biology*, pages 269–299. MIT Press, 2002.

[172] R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. In *Proceedings of the 27th International Workshop Graph-Theoretic Concepts in Computer Science (WG)*, 2002. To appear.

[173] R. Sharan, R. Elkon, and R. Shamir. Cluster analysis and its applications to gene expression data. In H.-W. Mewes, H. Seidel, and B. Weiss, editors, *Proceedings of the 38th Ernst Schering workshop on Bioinformatics and Genome Analysis*, pages 83–108. Springer Verlag, 2002.

[174] R. Sharan, A. Maron-Katz, N. Arbili, and R. Shamir. EXPANDER: EXPression ANalyzer and DisplayER, 2002. Software package, Tel-Aviv University, `http://www.cs.tau.ac.il/~roded/click.html`.

[175] R. Sharan and R. Shamir. CLICK: A clustering algorithm with applications to gene expression analysis. In *Proceedings of the Eighth International Conference*

on Intelligent Systems for Molecular Biology (ISMB), pages 307–316. AAAI Press, 2000.

[176] R.R. Sokal. Clustering and classification: Background and current directions. In J. Van Ryzin, editor, *Classification and Clustering*, pages 1–15. Academic Press, 1977.

[177] P.T. Spellman, G. Sherlock, M. Zhang, V.R. Iyer, K. Anders, M. Eisen, P.O. Brown, D. Botstein, and B. Futcher. Comprehensive identification of cell cycle regulated gene of the yeast *Saccharomyces Cerevisia* by microarray hybridization. *Mol. Biol. Cell*, 9:3273–3297, 1998. `http://cellcycle-www.stanford.edu`.

[178] J. Spinrad. *Two dimensional partial orders*. PhD thesis, Dept. of Computer Science, Princeton University, 1982.

[179] M.A. Steel. The complexity of reconstructing trees form qualitative characters and subtrees. *Journal of Classification*, 9:91–116, 1992.

[180] D.L. Swofford. *PAUP, Phylogenetic Analysis Using Parsimony (and Other Methods)*. Sinaur Associates, Sunderland, Massachusetts, 1998. Version 4.

[181] P. Tamayo, D. Slonim, J. Mesirov, Q. Zhu, S. Kitareewan, E. Dmitrovsky, E.S. Lander, and T.R. Golub. Interpreting patterns of gene expression with self-organizing maps: Methods and application to hematopoietic differentiation. *Proc. Natl. Acad. Sci. USA*, 96:2907–2912, 1999. `http://www-genome.wi.mit.edu/cancer/`.

[182] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Computing*, 13:566–579, 1984.

[183] S. Tavazoie, J. Hughes, M. Campbell, R. Cho, and G.M. Church. Systematic determination of genetic network architecture. *Nature Genetics*, 22:281–285, 1999.

[184] M. Thorup. Decremental dynamic connectivity. *Journal of Algorithms*, 33:229–243, 1999.

[185] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the 32th Annual ACM Symposium on Theory of Computing (STOC)*, pages 343–350, 2000.

[186] R.S. Tibbetts, K.M. Brumbaugh, J.M. Williams, J.N. Sarkaria, W. A. Cliby, S.Y. Shieh, Y. Taya, C. Prives, and R.T. Abraham. A role for ATR in the DNA damage-induced phosphorylation of p53. *Genes Dev.*, 13:152–157, 1999.

[187] R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a dataset via the gap statistics. Technical report, Stanford University, Stanford, 2000.

[188] P. Toronen, M. Kolehmainen, G. Wong, and E. Castren. Analysis of gene expression data using self-organizing maps. *FEBS Letters*, 451:142–146, 1999.

[189] J.D. Watson, M. Gilman, J. Witkowski, and M. Zoller. *Recombinant DNA*. W.H. Freeman, New York, 2nd edition, 1992.

[190] G. Wegner. *Eigenschaften der nerven homologische einfacher familien in R^n*. PhD thesis, Götingen, 1967.

[191] M. Xiong, L. Jin, W. Li, and E. Boerwinkle. Computational methods for gene expression based tumor classification. *Biotechniques*, 29:1264–1270, 2000.

[192] Y. Xu, T. Ashley, E.E. Brainerd, R.T. Bronson, M.S. Meyn, and D. Baltimore. Targeted disruption of ATM leads to growth retardation, chromosomal fragmentation during meiosis, immune defects, and thymic lymphomas. *Genes Dev.*, 10:2411–2422, 1996.

[193] J. Xue. Edge-maximal triangulated subgraphs and heuristics for the maximum clique problem. *Networks*, 24, 1994.

[194] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2(1):77–79, 1981.

[195] M. Yannakakis. Edge deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981.

[196] A. Yao. Should tables be sorted. *Assoc. Comput. Mach.*, 28(3):615–628, 1981.

[197] K.Y. Yeung, D.R. Haynor, and W.L. Ruzzo. Validating clustering for gene expression data. *Bioinformatics*, 17:309–318, 2001.

[198] G. Yona, N. Linial, and M. Linial. Protomap: Automatic classification of protein sequences, a hierarchy of protein families, and local maps of the protein space. *Proteins: Structure, Function, and Genetics*, 37:360–378, 1999.

[199] M.Q. Zhang. Large scale expression data analysis: A new challenge to computational biologist. *Genome Research*, 9:681–688, 1999.

[200] R. Zhao, K. Gish, Y. Yin, D. Notterman, W. Hoffman, E. Tom, D. Mak, and A.J. Levine. Analysis of p53 regulated gene expression patterns using oligonucleotide arrays. *Genes and Dev.*, 14:981–993, 2000.

[201] J. Zhu and M.Q. Zhang. SCPD: a promoter database of the yeast *Saccharomyces Cerevisiae*. *Bioinformatics*, 15:607–611, 1999. `http://cgsigma.cshl.org/jian/`.