uOttawa

L'Université canadienne
Canada's university

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

FACULTÉ DES ÉTUDES SUPÉRIEURES
ET POSTOCTORALES

FACULTY OF GRADUATE AND
POSDOCTORAL STUDIES

Maryam Haghighi
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (Mathematics)
GRADE / DEGREE

Department of Mathematics and Statistics
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Graph-Theoretic Analysis of Gene Expression Networks

TITRE DE LA THÈSE / TITLE OF THESIS

Dr. M. Sajna
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Dr. S. Boyd

Dr. K. Cheung

Dr. S. Findlay

Gary W. Slater
Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

GRAPH-THEORETIC ANALYSIS OF GENE EXPRESSION NETWORKS

Maryam Haghighi

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the Master of Science Degree in Mathematics[1]

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

©Maryam Haghighi, Ottawa, Canada, 2007

[1]The Master's Program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-32452-3
Our file Notre référence
ISBN: 978-0-494-32452-3

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

© Copyright 2007
by
Maryam Haghighi

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Dedication

To my parents;

They were my very first teachers, and continue to be the most important influence in my life.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

I would like to gratefully acknowledge all of the people who assisted me with this thesis. I am thankful for the guidance and patience of my supervisor, Dr. Mateja Šajna. Thanks to the Center for Cancer Therapeutics at the Ottawa Health Research Institute for providing access to the database and their financial support, in particular to Dr. Michael McBurney, Dr. Miguel Andrade, and Paul Krzyzanowski. Many thanks to Dr. Scott Findlay for the initiation of the project and his advice with the biological aspects of this research. Thanks to Dr. Mayer Alvo for his assistance with answering statistical questions. Thanks to the Department of Mathematics and Statistics for financial support and guidance, especially to Dr. Vladimir Pestov, and Chantal Giroux. I am grateful to Abelkarim El-Basraoui, Andrea Burgess, Paul Elliott, Camelia Karimianpour, Aziz Khanchi, Termeh Kousha, Susanna Wienns, and all my friends at the Department of Mathematics for their assistance and friendship. Thanks to Derek, who was always there to encourage me, and special thanks to my mother and father for everything else during these last few years, and throughout my life.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

We use graph theory to model a database of gene expression levels and provide a tool that can assist in designing biological experiments which could help to better understand the interactions between genes. The data was provided by StemBase (http://www.stembase.ca). We model a portion of this database as a graph, and study some parameters on it that may be biologically relevant. We include theoretical discussion of the parameters, and presentation of algorithms for their computation. The focus is on structural properties of the graph; thus, we investigate the graph's bipartiteness, connected components, distance between vertices, radius, diameter, center, cut-vertices and blocks, and its maximal and maximum cliques. We develop the terminology and results used for modeling the database, and implement some of the algorithms in MATLAB. We analyze the results of running the code on the database and discuss their relevance.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

Dedication ii

Acknowledgements iii

Abstract iv

1 Introduction 1

1.1 Background 2

1.1.1 Agglomerative Algorithms 3

1.1.2 Divisive Algorithms 3

1.1.3 Pattern-Based Algorithms 3

1.1.4 K-means Algorithm 3

1.1.5 Self-Organizing Map Algorithms 4

1.1.6 Model-Based Algorithms 4

1.1.7 Graph-Based Algorithms 4

2 Preliminaries 6

2.1 Graph Theory 6

2.2 Computer Science 15

2.2.1 Conventions for Algorithms 15

2.2.2 Algorithmic Complexity 16

2.2.3 Stack and Queue 17

2.2.4 Graph Traversal Methods 17

2.3 Statistics 23

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4 Molecular Biology 28

3 Connectedness 32

4 Bipartiteness 36

5 Distance 39
5.1 Distance in Graphs without Weights 39
5.2 Distance in Weighted Graphs 43
5.2.1 Dijkstra's Algorithm 43
5.2.2 Floyd-Warshall's Algorithm 46
5.3 Eccentricity, Radius, Diameter, Center and Median for Graphs without Weights 47
5.4 Distance and Eccentricity for a Subgraph 52

6 Blocks and Cut-Vertices 55
6.1 Cut-Vertices 55
6.2 Blocks 58

7 Maximal and Maximum Cliques 62
7.1 Maximal Cliques 63
7.1.1 Finding a Maximal Clique Containing a Vertex of Maximum Degree 63
7.1.2 Finding All Maximal Cliques 65
7.2 Maximum Cliques 69
7.3 Comparison 72

8 Modeling the Database 73
8.1 Deriving a Graph from the Data 73
8.2 Gene Expression Graph 75
8.2.1 Properties of weighted graphs 75
8.2.2 Gene expression graph and perfect correlation graph 79
8.3 Finding a Suitable Pair of Thresholds 80

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8.4 Other Parameters on the Gene Expression Graph 84

9 Implementation of the Algorithms 86

9.1 MATLAB Code 86

9.2 Sample Output 112

10 Analyzing the Results 117

11 Further Research 124

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

In this thesis, we analyze a specific database of gene expression levels of samples of cells from a mathematical point of view. This database is called StemBase [31], and is provided by the Center for Cancer Therapeutics at the Ottawa Health Research Institute (http://www.ohri.ca/programs/cancer_therapeutics/). StemBase can be accessed through http://www.stembase.ca. The goal of this thesis is to use graph theory to model this database, and provide a tool that can assist in designing experiments which could help to better understand the interactions between genes.

The database is modeled by a graph whose vertices represent the genes, and edges represent the relationship between the expression levels of the corresponding genes. We study some parameters on this graph that may be biologically relevant, such as distance, eccentricity, radius, diameter, etc. The focus is on structural properties of the graph; thus, we investigate the graph's bipartiteness, connected components, center, median, cut-vertices and blocks, and its maximal and maximum cliques.

The thesis is organized as follows. Chapter 2 provides a brief review of some definitions and results used in the thesis. These concepts and results arise from various parts of graph theory, computer science, statistics, and molecular biology. Chapters 3-7 contain a short study of the parameters mentioned above, and the theoretical development and analysis of the algorithms for computing them. Chapter 8 illustrates our approach to modeling the database of gene expression levels as a graph and applying the algorithms of previous chapters to this model. In this chapter, we develop

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the necessary terminology and results used for modeling the database. Chapter 9 includes the implementation of some of the algorithms in MATLAB. In this thesis we have limited ourselves to only a portion of the database, and implemented only those algorithms that are known to be polynomial. In Chapter 10 we analyze the results of running the code on the database and discuss their relevance. Since this project can be continued in various ways, such as considering other parameters and measures on the database, we include Chapter 11, which provides some suggestions and ideas for further research on this topic. We should mention that Chapters 2-7 contain material that is surveyed based on existing results. Chapters 8-11 describe original methods and results that we have found during this research.

The main goal of this research is to explore a novel approach to studying gene expression data. Since the method taken in this thesis is an original method, this research should be viewed as an example that presents the possibility and advantages of our approach, which may be used for further research in this area.

1.1 Background

There are several studies on mathematical analysis of gene expression data obtained from microarray analysis. Most of the research in this area focuses on partitioning genes into clusters; that is, identifying groups of genes that have similar expression patterns. There are various clustering methods. Since the structure of gene expression data is complicated, different clustering methods may yield very different results. There are three main approaches to clustering: partition-based approach, hierarchical approach, and pattern-based approach [25]. Each of these approaches comprises several methods. For example, agglomerative algorithms and divisive algorithms are examples of a hierarchical approach. The K-means algorithm, self-organizing map algorithm, model-based algorithms, and graph-theoretic algorithms are examples of a partition-based approach. We will give a brief description of each of these methods.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.1.1 Agglomerative Algorithms

These algorithms start from individual genes as clusters, and successively merge pairs of clusters. [16] used such a method for clustering to analyze gene expression data. One disadvantage of this method is that the initial structure of the clusters becomes distorted after each merge [42].

1.1.2 Divisive Algorithms

These algorithms start from one cluster containing all the genes and divide the cluster(s) until each cluster contains only one gene [2] or certain stop criteria are met [26]. One disadvantage with these algorithms is that it is difficult to set stop criteria when studying gene expression data.

1.1.3 Pattern-Based Algorithms

This approach is based on determining a subset of objects (here genes) that exhibit similar behavior across a subset of conditions [35]. This process, called biclustering, was first introduced in the seventies. [12] applied this to gene expression data. The main difficulty with this approach is that the algorithms for pattern-based clustering are heuristic and therefore can not guarantee to find desired results. [35] used this method with some combinatorial modifications and was able to give a polynomial time algorithm which works under certain restrictions on the original problem. [37] presented a general statistical framework that combines biclusters with the notion of a weighted bipartite graph and applies a greedy algorithm to identify dense biclusters.

1.1.4 K-means Algorithm

This algorithm partitions the data set into K clusters by minimizing the sum of the squared distances of genes from their cluster centers [36]. The main disadvantage of this algorithm is that it requires to specify the number of clusters, K, from the start; this number is usually unknown for gene expression data.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.1.5 Self-Organizing Map Algorithms

This method starts from seed nodes on a specific geometric structure (for example, a two dimensional grid) and moves the seed nodes towards randomly selected points until so-called convergence [34]. The draw-back of this method is the requirement of specifying the numbers and the geometry of nodes, which affects the results of the clustering.

1.1.6 Model-Based Algorithms

These methods are based on imposing a statistical approach on the gene expression data. Several papers describe the use of multivariate Gaussian distributions as a model. This Gaussian model is not a good choice if the gene expression data contains information about expression levels of specific genes during a series of time points, since the Gaussian model does not consider the inherent dependency of such information. [13] has described the main difficulties that may arise in considering causal models or conditional independence, and some solutions for these difficulties.

1.1.7 Graph-Based Algorithms

These algorithms model gene expression data as a graph. There are two major approaches in such a modeling. The first approach uses a bipartite graph such that the vertices in one part represent the genes, and the vertices in the other part represent some type of a pattern that the genes may be associated with. [6] and [3] are examples of such a use of bipartite graph.

The second approach is to use as a model a graph whose vertices represent the genes, and two vertices are adjacent if the corresponding genes are "similar". Various measures for determining similarity have been considered. In [37], the clustering algorithm is based on a probabilistic model combined with the notion of cliques in the graph. The authors developed a heuristic algorithm that identifies such special cliques. [33] employed so-called p-quasi complete linkage clustering to the gene expression data. In a p-quasi complete subgraph, any member of one group has edges to one p-th of all members of the same group. As mentioned in [33], a restricted

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

version of the p-quasi complete graph problem has been proven to be NP-complete. Therefore, the authors developed an approximation algorithm for it.

[30] explores the possibility of relating gene expression data to protein interaction data. For $0 < \gamma < 1$, a subset of k genes (proteins) form a γ-quasi cluster if each gene (protein) is similar to at least $\gamma(k-1)$ other genes (proteins). The authors try to find a subset of genes that is a γ_1-quasi cluster and the proteins that are the product of these genes form γ_2-quasi cluster, for certain values γ_1 and γ_2.

[41] described a new approach based on representing gene expression data as a minimum spanning tree and took advantage of the simple structure of a tree to develop clustering algorithms.

[27] builds a graph from gene expression data in such a way that each vertex is a gene, and two vertices x and y are adjacent if the distance between x and y in the distance matrix (constructed by using gene expression data) is at most equal to some given value of a threshold t (t can be any real number). The paper uses a special method called clique minimal separator decomposition on a data set of 40 genes to demonstrate the method.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

Preliminaries

In this chapter we have included the main definitions and results from different subjects of graph theory, computer science, statistics, and molecular biology that are required for this thesis. The material of this chapter serves as a quick introduction. For further studies the reader is referred to the standard textbooks on these subjects.

2.1 Graph Theory

Graph theory is formally a branch of combinatorics that has applications in several areas of science such as computer science, biology, chemistry, telecommunications, etc. In this section we will present some definitions and theorems from graph theory.

Definition 2.1 A *graph* $G = (\mathrm{V(G)}, \mathrm{E(G)})$ is a structure consisting of two finite sets V(G) and E(G) such that V(G) is nonempty and we have

$$\mathrm{E(G)} \subseteq \{\{u, v\} : u, v \in \mathrm{V(G)}, u \neq v\}.$$

The elements of V(G) are called *vertices* and the elements of E(G) are called *edges*.

We denote the number of vertices, $|V(\mathrm{G})|$, by n and the number of edges, $|E(\mathrm{G})|$, by m.

For a graph G, if $e = \{u, v\}$ is an edge, then u and v are called its ends, e is said to join u and v, and u and v are called *adjacent* or *neighbors*. By $u \sim_{\mathrm{G}} v$ we mean

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that vertices u and v are adjacent in G. If vertices u and v are two ends of an edge e, then e is said to be *incident* with v.

Remark 2.2 In some graph theory literature, what we defined as a graph is referred to as a simple finite graph.

Remark 2.3 According to Definition 2.1, the two ends of an edge have to be distinct. If we define an edge to be a pair of not necessarily distinct vertices, then we may have an edge on one vertex. Such an edge is referred to as a *loop*.

Definition 2.4 The *degree* of a vertex v in a graph G, denoted by $\deg(v)$ is the number of edges in G that are incident with v. By our definition of a graph, this is equal to the number of neighbors of v.

The set of neighbors of a vertex v in a graph G is denoted by $N_{\mathrm{G}}(v)$ or simply by $N(v)$.

The number $\delta(\mathrm{G}) = \min\{\deg(v) | v \in \mathrm{V(G)}\}$ is the *minimum degree* of G, and the number $\Delta(\mathrm{G}) = \max\{\deg(v) | v \in \mathrm{V(G)}\}$ is the *maximum degree* of G.

Theorem 2.5 *For any graph G,*

$$2m = \sum_{v \in \mathrm{V(G)}} \deg(v).$$

PROOF. When we sum all the vertex degrees in G, we count every edge exactly twice, once from each of its ends. □

Definition 2.6 The *adjacency matrix* of a graph G with vertex set $\mathrm{V(G)} = \{v_1, ..., v_n\}$, denoted by $A(\mathrm{G})$, is an $n \times n$ matrix such that each entry a_{ij} in the matrix is 1 if there is an edge joining v_i and v_j, and 0 otherwise.

Definition 2.7 Let G be a graph. A graph H is called a *subgraph* of a graph G, denoted by $\mathrm{H} \subseteq \mathrm{G}$, if $\mathrm{V(H)} \subseteq \mathrm{V(G)}$ and $\mathrm{E(H)} \subseteq \mathrm{E(G)}$.

Let V' be a nonempty subset of V(G). The subgraph of G whose vertex set is V' and whose edge set is the set of elements of E(G) that have both ends in V' is called

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the subgraph of G *induced* by V′ and is denoted by G[V′]. The graph H is called an *induced subgraph* of G if H = G[V′] for some V′ ⊆ V(G).

Suppose E′ ⊆ E(G) and E′ is nonempty. The subgraph of G whose vertex set is the set of ends of the edges in E′ and whose edge set is E′ is called the subgraph of G *induced* by E′ and is denoted by G[E′].

A *spanning subgraph* of G is a subgraph H of G such that V(H) = V(G).

Definition 2.8 If v is a vertex of a graph G, then by the *vertex-deletion subgraph* $G - v$ we mean the subgraph of G induced by the vertex set $V(G) - \{v\}$. That is,

$$V(G - v) = V(G) - \{v\} \quad \text{and} \quad E(G - v) = \{e \in E(G) : v \text{ is not an end of } e\}.$$

Definition 2.9 If e is an edge of a graph G, then by the *edge-deletion subgraph* $G - e$ we mean the subgraph obtained from G by deleting the edge e. That is,

$$V(G - e) = V(G) \quad \text{and} \quad E(G - e) = E(G) - \{e\}.$$

Definition 2.10 The *union* of two graphs G_1 and G_2, denoted by $G_1 \cup G_2$, is the graph with vertex set $V(G_1) \cup V(G_2)$ and edge set $E(G_1) \cup E(G_2)$.

Definition 2.11 A *weighted graph* is a graph G together with a function $w : E(G) \to \mathbb{R}$. If $e \in E(G)$, then $w(e)$ is called the *weight* of the edge e.

Definition 2.12 A *walk* in a graph G is a finite non-empty alternating sequence $W = v_0e_1v_1e_2v_2...e_kv_k$ of vertices and edges such that v_{i-1} and v_i are the ends of e_i for $1 \leq i \leq k$.

A walk from a vertex v_0 to a vertex v_k is also called a $(v_0, v_k) - $ *walk*. A walk $v_0e_1v_1e_2v_2...e_kv_k$ is called *closed* if $v_0 = v_k$. If $W_1 = v_0e_1v_1...e_iv_i$ and $W_2 = v_ie_{i+1}...e_kv_k$ are walks, then by $W_1 + W_2$ we mean the walk obtained by concatenating W_1 and W_2 at v_i, that is, $v_0e_1v_1...e_kv_k$. A $(v_i, v_j)-$*section* of a walk $W = v_0e_1v_1e_2v_2...e_kv_k$ is a walk $v_ie_{i+1}...e_jv_j$ that is a subsequence of consecutive terms of W.

Since in our definition of a graph there is at most one edge between any pair of vertices, a walk $v_0e_1v_1e_2v_2...e_kv_k$ can be determined by the sequence $v_0v_1v_2...v_k$ of its vertices, and may be denoted by this sequence.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2.13 If the vertices of a walk are pairwise distinct, then the walk is called a *path*. A (v_0, v_r)-*path* $P = v_0v_1...v_r$ in a graph G is thus a sequence of $r+1$ distinct vertices starting with v_0 and ending with v_r such that for each i, $0 \leq i < r$, the vertices v_i and v_{i+1} are adjacent in G. The *length* of the path $P = v_0v_1...v_r$, denoted by $l(P)$, is equal to r.

Lemma 2.14 For any two vertices x, y in a graph G, there is an $(x, y)-$path in G if and only if there is an $(x, y)-$walk in G.

PROOF. ($\Rightarrow$) By definition, any path is a walk. Therefore, if there is an $(x, y)-$path in G, then there is an $(x, y)-$walk.

($\Leftarrow$) Let $W = xv_1v_2...v_sy$ be an $(x, y)-$walk in G that is not a path. Suppose that v_i and v_j are two vertices in W such that $v_i = v_j$, and suppose $i \leq j$. By deleting the sequence $v_{i+1}...v_j$ from W we have a new sequence $P = xv_1...v_iv_{j+1}...v_sy$ such that each two consecutive vertices are adjacent. Repeat the same process for P until you get a sequence Q such that there are no repeated vertices in Q. Therefore, Q is an $(x, y)-$path in G. □

Definition 2.15 If there is a path between any two vertices of a graph G, then G is *connected.* A graph that is not connected is called *disconnected.* A *connected component* C of a graph G is a maximal subgraph of G that is connected. The number of connected components of a graph G is denoted by $\omega(\mathrm{G})$. If there is a path P from a vertex u to a vertex v in a graph G, then we say u is *connected to* v via path P. Such a binary relation on the vertex set of a graph is called *connection.*

Connection is an equivalence relation on the vertex set V(G). So there is a partition of V(G) into the equivalence classes $\mathrm{V}_1, \mathrm{V}_2, ..., \mathrm{V}_\omega$. The connected components of G are then precisely the induced subgraphs $\mathrm{G}[\mathrm{V}_1], \mathrm{G}[\mathrm{V}_2], ..., \mathrm{G}[\mathrm{V}_\omega]$.

Definition 2.16 A closed walk $W = v_0v_1v_2...v_k$ such that $v_0 = v_k$ and $v_0, ..., v_{k-1}$ are pairwise distinct is called a *cycle.* A graph that does not have any cycles is called *acyclic.* The *length* of a cycle $W = v_0v_1v_2...v_k$ is equal to k.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Remark 2.17 It should be mentioned that paths and cycles in a graph G can be viewed as subgraphs of G. We shall use each of these terms in both meanings, and it should be clear from the context whether we mean a sequence of vertices or a subgraph.

Definition 2.18 A *tree* is a connected acyclic graph. A *rooted tree* is a tree with a distinguished vertex called the *root*.

Definition 2.19 A *forest* is an acyclic graph.

The connected components of a forest are trees.

Lemma 2.20 [7] There is a unique path between any two vertices in a tree.

PROOF. By contradiction. Let T be a tree and u and v be two vertices of T. By definition, T is connected. Therefore, there is at least one (u, v)–path in T, say P. Now suppose there is another path Q from u to v. Since $P \neq Q$, there exists an edge $e = xy$ such that e is in P, but not in Q. Now $P \cup Q - \{e\}$ is connected. So, it contains a (x, y)–path R. But Ryx is a cycle in T. This contradicts with T being a tree. Therefore, there is a unique path between any two vertices in T. □

Definition 2.21 In a rooted tree, a vertex v that immediately precedes vertex w on the unique path from the root to w is called the *parent* of w, and w is called the *child* of v.

In a rooted tree a vertex w is called a *descendant* of a vertex v, and v is called an *ancestor* of w, if v is on the path from the root to w.

Example 2.22 Figure 1 shows a disconnected graph G of 16 vertices and 17 edges. G has two connected components. The sequence *nlknmon* is a closed walk in G and *kmon* is a path of length 3. Component 2 has a cycle *kmnlk* of length 4, and component 1 is a tree.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

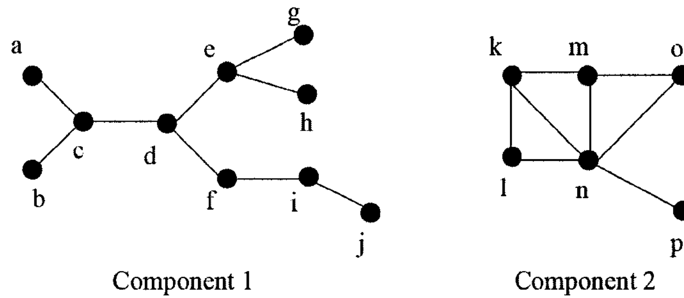

Figure 1

Definition 2.23 A *complete* graph K_n is a graph on n vertices such that there is an edge between every pair of vertices.

From the above definition, we can conclude that K_n has $\binom{n}{2}$ edges and each vertex is of degree $n-1$.

Definition 2.24 A *cut-edge* of a graph G is an edge e of G such that $\omega(G) < \omega(G-e)$.

Definition 2.25 A *cut-vertex* of a graph G is a vertex v of G with the property that E(G) can be partitioned into two nonempty subsets E_1 and E_2 such that v is the only vertex $G[E_1]$ and $G[E_2]$ have in common.

Remark 2.26 If G is a (loopless) graph, then v is a cut-vertex of G if and only if $G - v$ has more connected components than G; that is, $\omega(G - v) > \omega(G)$.

Let G be a connected (loopless) graph. Then v is a cut-vertex of G if and only if $G - v$ is disconnected.

Definition 2.27 A connected graph that has no cut-vertices is called a *block*. A *block of a graph* G is a maximal subgraph of G that is a block.

A block H of a graph G has no vertices that are cut-vertices of H, but it may contain vertices that are cut-vertices of G.

Clearly, complete graphs are blocks and have no cut-vertices. Every induced subgraph of a complete graph is a block.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Example 2.28 In Figure 2 vertex d is a cut-vertex of G, and edge cd is a cut-edge of G. The subgraph H of G induced by the set of vertices $\{d, e, f, g\}$ is a block of G.

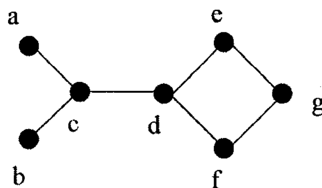

Figure 2

Theorem 2.29 [21] Let B_1 and B_2 be two blocks of a graph G. Then B_1 and B_2 have at most one vertex in common.

Proof. By contradiction. Suppose that B_1 and B_2 are two distinct blocks with two vertices x and y, $x \neq y$, in common. We will show that $B_1 \cup B_2$ is a block. Suppose not. Then $B_1 \cup B_2$ has a cut-vertex c. There are two cases:

Case 1: Suppose $c \in \{x, y\}$. Without loss of generality we assume that $c = x$. We show that $(B_1 \cup B_2) - x$ is connected. Choose any two vertices $v, w \in (B_1 \cup B_2) - x$. Since B_1 is a block, it has no cut-vertices.Therefore, $B_1 - x$ is connected. Similarly, B_2 is a block, and so $B_2 - x$ is connected. First assume that both vertices v and w belong to the same block, say B_1. Then, since $B_1 - x$ is connected, it contains a path between v and w. Now assume that v and w belong to different blocks, say $v \in B_1$ and $w \in B_2$. Again, since $B_1 - x$ is connected, there is a path P between v and y in $B_1 - x$. Similarly, $B_2 - x$ is connected, and there is a path Q between vertex y and vertex w in $B_2 - x$. Therefore, $P \cup Q$ is a $(v, w)-$walk in $(B_1 \cup B_2) - x$. So, there is a $(v, w)-$path in $(B_1 \cup B_2) - x$. Therefore, vertex $x = c$ is not a cut-vertex of $B_1 \cup B_2$.

Case 2: Without loss of generality suppose $c \in B_1 - \{x, y\}$. Then we show that $(B_1 \cup B_2) - c$ is connected. As we mentioned before, $B_1 - c$ is connected, and from the assumption we know that B_2 is a block, and hence it is connected. Therefore, $(B_1 \cup B_2) - c = (B_1 - c) \cup B_2$ is the union of two connected graphs with two vertices in common, so it is connected. That means $B_1 \cup B_2$ does not have a cut-vertex.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

So, in both cases $B_1 \cup B_2$ has no cut-vertices, and hence it is a block. This contradicts the maximality of blocks B_1 and B_2. Therefore, B_1 and B_2 have at most one vertex in common. □

Definition 2.30 A *clique* of a graph G is a set of vertices $C \subseteq \mathrm{V(G)}$ such that $\mathrm{G}[C]$ is complete.

By the *size* of a clique we mean the number of its vertices. A *maximum clique* of a graph G is a clique with maximum size. A clique C of a graph G is *maximal* if there is no clique D of G such that $C \subseteq D$ and $C \neq D$.

Example 2.31 In Figure 3, $C_1 = \{e, f, g\}$ is a clique of size 3 of G, and $C_2 = \{a, b, c\}$ is a maximal clique. $C_3 = \{d, e, f, g, h\}$ is a maximum clique of G.

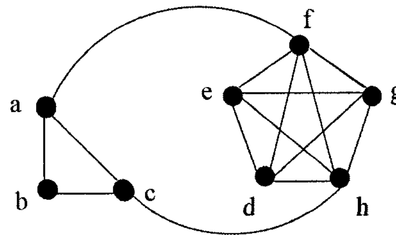

Figure 3

Definition 2.32 A graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ is called *bipartite* if we can partition its vertex set V into two sets V_1 and V_2 such that every edge of G has one end in V_1 and the other end in V_2.

Example 2.33 Figure 4 shows a bipartite graph G with bipartition $(\mathrm{V}_1, \mathrm{V}_2)$ such that $\mathrm{V}_1 = \{a, b, c\}$ and $\mathrm{V}_2 = \{d, e\}$.

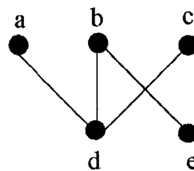

Figure 4

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 2.34 A graph G is bipartite if and only if it contains no cycles of odd length.

PROOF. ($\Rightarrow$) Suppose that G is bipartite with parts V_1 and V_2. We will show that every cycle of G has even length. Let $C = v_0v_1...v_kv_0$ be a cycle of G and suppose that $v_0 \in V_1$. Since v_0v_1 is an edge and G is bipartite, $v_1 \in V_2$. Similarly, $v_2 \in V_1$, and in general $v_{2i} \in V_1$ and $v_{2i+1} \in V_2$. Since $v_0 \in V_1$, we have $v_k \in V_2$. Hence, $k = 2i + 1$ for some i and therefore C is of even length.

($\Leftarrow$) Assume that G has no odd cycles; we will prove that it is bipartite, that is, we can find a partition of the vertex set V into two parts V_1 and V_2 such that each edge joins a vertex from V_1 to a vertex from V_2. Without loss of generality, we may assume that G is connected. If G is not connected, we use the following for each connected component of G.

Let v be a fixed vertex of G, and for any vertex u let $d(u)$ be the length of a shortest path from v to u. Let V_1 be the set of the vertices u such that $d(u)$ is even, and V_2 be the set of vertices such that $d(u)$ is odd. We claim that (V_1, V_2) is a bipartition of G. Suppose not; that is, suppose u and w are two vertices from the same part, say V_1, such that $u \sim w$. Let P be a shortest $(v, u)-$path and Q be a shortest $(v, w)-$path. Let x be the last vertex that P and Q have in common. Since P and Q are shortest $(v, u)-$ and $(v, w)-$path, respectively, the $(v, x)-$sections of P and Q are shortest $(v, x)-$paths, and hence, they have the same length. Since $u, w \in V_1$, P and Q have even lengths, and therefore the lengths of the $(x, u)-$section of P and $(x, w)-$section of Q have the same parity. Now, if we start from x and go along the $(x, u)-$section of P and then go through the edge uw and then go backward through the $(x, w)-$section of Q, we will get a cycle of odd length. This is because the lengths of $(x, u)-$section of P and $(x, w)-$section of Q have the same parity. Therefore, we have an odd cycle, but this is a contradiction. The proof is similar for the case that u and w lie in V_2. So, no two vertices of the same part can be adjacent. Therefore, (V_1, V_2) is a bipartition of G. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Computer Science

There are many connections between graph theory and computer science. Algorithms play a strong role in graph theory research. Since a major part of this thesis contains various algorithms, we first mention the conventions that we will use for algorithms.

We give a short introduction to the notion of complexity and NP-completeness for algorithms. This will give us a way to measure (or compute) the time it takes for the algorithm to return the answer. It also gives us a tool to compare different algorithms for the same parameter.

We will briefly introduce the two most frequently used data structures, stack and queue. Both are restricted versions of the linear or ordered list data structure, so they can be implemented using arrays or linked lists.

Many theoretical and applied problems in graph theory require traversing a graph in a particular way. We will discuss two of the most famous graph traversals methods, namely the Depth-First Search and the Breadth-First Search. These methods are widely used in the future chapters for various algorithms.

2.2.1 Conventions for Algorithms

Throughout this thesis, the algorithms are written in a pseudo code similar to programming languages such as Pascal and C.

At the beginning of a section containing an algorithm, the algorithm's name (if there is one) and its purpose are described. Then the input and output of the algorithm are mentioned. Whenever an algorithm is used (called) in another algorithm later in the thesis, we will give a specific name to the first algorithm and put its input(s) and output(s) in parentheses, separated by commas. Upon calling of the first algorithm, the second algorithm provides the necessary input and receives the output. This method is very similar to the usual calling of functions in C.

Each algorithm starts with "begin" and ends with "end.". If anywhere during the algorithm we use "stop", it means that one should terminate the algorithm.

Whenever we use the letter "M" as a value assigned to a variable, it means that the variable is equal to infinity for that algorithm. It should be mentioned that in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an implementation of such an algorithm, "M" can be substituted by a very large number which would not occur for that specific variable throughout the algorithm, and therefore, it would signify infinity.

By "$a := b$" we mean assigning b to the variable a. A phrase such as "$a = b$" returns true if a is equal to b and false otherwise.

Comments within algorithms are contained between two % signs.

2.2.2 Algorithmic Complexity

Definition 2.35 Let f and g be functions $\mathbb{N} \rightarrow \mathbb{R}$. The function f is in the family $O(g(n))$ (Big-O of $g(n)$) if there is an integer n_0 and a positive real number c such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$.

Definition 2.36 An algorithm is of complexity $O(f(n))$ if its running time (as a function on the input size n) is in $O(f(n))$ for some function $f(n)$. If $f(n)$ is a polynomial, then the algorithm is called a *polynomial-time* algorithm.

Definition 2.37 [21] A *decision problem* is a problem that requires only a "yes" or "no" answer regarding whether some element of its domain has a particular property.

A decision problem belongs to the *class P* if there is a polynomial-time algorithm to solve the problem.

A decision problem belongs to the *class NP* is there is a way to provide evidence of correctness of a "yes" answer so that it can be confirmed by a polynomial-time algorithm.

There is a large collection of problems (called NP-complete) for which no polynomial-time algorithms have been found despite considerable effort over the last several decades. It remains open whether there exist polynomial-time algorithms for these problems. It turns out that if one can find a polynomial-time algorithm for any one of these problems, then polynomial-time algorithms can be found for all of them. Here we provide a brief introduction to this topic. A precise, detailed description of this class of problems can be found in standard references.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2.38 A decision problem R is *polynomially reducible* to a decision problem Q if there is a polynomial-time transformation of each instance I_R of problem R to an instance I_Q of problem Q, such that instances I_R and I_Q have the same answer (yes or no).

A decision problem is NP-*hard* if every problem in class NP is polynomially reducible to it.

An NP-hard problem R is NP-*complete* if R is in class NP.

It can be shown that the class of NP-complete problems is non-empty.

2.2.3 Stack and Queue

A *stack* is a list of elements in which insertions and deletions are permitted only at one end, called the *top*. The other end is called the *bottom*. That means that it is possible to remove elements from a stack only in reverse order from the insertion of elements into the stack. Therefore, the structure of a stack is usually referred to as Last-In-First-Out (LIFO). The operation for inserting or adding an element into the stack is often called *pushing*. The operation for removing or deleting an element from the stack is called *popping*.

A *queue* is a linear list in which additions and deletions take place at different ends. The end at which we can insert a new element is called the *rear*, and the end at which we can delete or remove an element is called the *front*. A queue data structure is referred to as First-In-First-Out (FIFO). The operation for deleting an element from a queue is sometimes referred to as *dequeuing*.

2.2.4 Graph Traversal Methods

In this section we discuss two famous methods for traversing a graph (that is, systematically visiting all vertices of a graph). First we discuss Depth-First Search. We study two algorithms for performing a Depth-First Search on a graph. The first one is a recursive algorithm and the second one is a non-recursive algorithm that uses a stack. Breadth-First Search is the second graph traversal method that we will discuss. An algorithm that performs a Breadth-First Search on a graph will be described.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Depth-First Search

Assume that we are in an adventure maze and our goal is to visit all the chambers in the maze systematically. Depth-First-Search, abbreviated DFS, is a powerful method to achieve our goal. To perform a DFS we start at some chamber and then we go along the passages to other chambers. If we reach a chamber that has not been visited before, we leave a mark in that chamber to know that we have now visited it. When we are at a chamber we continue our adventure along the passages to an unmarked chamber, as long as this is possible. Once we get to a chamber C such that all of its neighboring chambers are marked (they all have been visited before), we continue the search from the chamber that was visited immediately before the first visit to C. By proceeding in this manner we will eventually visit all of the chambers. We can represent this maze by a graph G whose vertices are the chambers and the edges are the passages between the chambers. So DFS is a technique to visit all vertices of the graph G.

Method 2.39 Let G be a graph with $V(G) = \{v_1, v_2, ..., v_n\}$. In DFS the vertex that is being currently visited is called the *active vertex.* Begin the DFS by choosing a first vertex to visit, say v_1. So v_1 is the active vertex and label it 1. Then select a vertex adjacent to v_1 that has the smallest subscript (and has not been visited before) and label it 2, and this vertex becomes our new active vertex. Add the edge v_1v_2 to the set F. In general, suppose that we are currently visiting vertex v. The next active vertex is the vertex with the smallest subscript that is adjacent to v and has not been visited before. This vertex is our new active vertex; label it with the next available label and the edge between v and the new active vertex will be added to the set F. If all the neighbors of v have been visited, then we go back to the vertex visited just before the first visit of v and this vertex becomes the new active vertex. We repeat this procedure until there is no vertex in that component of G that has not been visited. If G has other vertices that have not been visited (which means that G is disconnected), then choose the first such vertex as the new active vertex and repeat the above procedure starting from this vertex.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Notation 2.40 The label assigned by DFS to a vertex v of the graph G is called the *depth-first search index* and is denoted by $\text{dfi}(v)$.

At the beginning $\text{dfi}(v) = 0$ for all $v \in V(G)$, but when the DFS algorithm terminates, the depth-first search index for each vertex v gives us the order in which v was visited for the first time while performing the DFS. The set F contains a subset of E(G). We claim that the subgraph of G induced by F does not have any cycles. Suppose not; so let u be the first vertex of a cycle $C = uvv_1v_2..v_nwu$ that we visit during the DFS. Since C is a cycle, v and w are neighbours of u on the cycle C. Assume that the DFS chooses to visit vertex v after u, and continue DFS to v_1, v_2,...,v_n, and w. When the DFS visits w, since u has already been visited, the DFS must go back through all the vertices v_n,...,v_1, and v. Therefore, it is impossible to have a cycle C in the set F of the edges that the DFS visits. So the subgraph induced by F is acyclic, and therefore, it is a forest. This forest is called a *depth-first search forest.* If G is connected, then F is a spanning tree, called a *depth-first search tree.* Assume that F is a DFS forest. Then each component of F is a rooted tree such that the root is the vertex v with the minimum $\text{dfi}(v)$ in that component, i.e. the root is the first vertex visited in that tree. Since F is a forest, the original graph G may contain edges that are not edges of F.

Algorithm 2.41 : *A Depth-First Search of a graph* G, *recursive method*

Input: Graph G

Output: Values of dfi for all vertices of G

procedure DFS(v)

begin

  $\text{dfi}(v) := i$

  $i := i + 1$

  for all vertices $v' \in N(v)$ **do**

    if $\text{dfi}(v') = 0$ **then**

      DFS(v')

end

begin

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$i := 1$

for all $v \in$ V(G) **do**

dfi(v) := 0

while for some u, dfi(u) = 0 **do**

DFS(u)

output dfi(v) for all vertices $v \in$ V(G)

end.

We will show that the complexity of the DFS algorithm is $O(max\{n, m\})$. The time spent by DFS(v) is proportional to deg(v), since the "for" loop is executed for all of the neighbors of v. Since $\sum_{w \in V(G)} \deg(w) = 2e$, the complexity of all calls of DFS is $O(m)$. Complexity for the initiation dfi(v) = 0 for all $v \in$ V(G) is $O(n)$ and also the "while" loop is of $O(n)$. The output line takes $O(m)$, hence the total algorithm has complexity $O(max\{n, m\})$.

Algorithm 2.41 uses a recursive method to implement a DFS. The next algorithm is a non-recursive algorithm to perform Depth-First Search on a graph by using a stack.

Algorithm 2.42 : *A Depth-First Search of a graph G, stack method*

Input: Graph G

Output: Values of dfi for all vertices of G

begin

for all vertices $v \in$ V(G) **do**

dfi(v) := 0

$i := 0$

create an empty stack S

for all $r \in$ V(G) **do**

if dfi(r) = 0 **then**

begin

push r to S

while $S \neq \emptyset$ **do**

begin

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            pop a vertex from S and call it v
            if dfi(v) = 0 then
            begin
              i := i + 1
              dfi(v) := i
              for all vertices w ∈ N(v) do
                if dfi(w) = 0 then
                  push w on S
            end
          end
        end
      output dfi(v) for all vertices v ∈ V(G)
    end.
```

Definition 2.43 Let G be a graph and F be a DFS forest of G. Each edge of G that is not an edge of F is called a *back edge.*

Clearly, each back edge is between two vertices in the same component of G, and since F is a spanning subgraph of G and each component of F is a spanning tree of a component of G, each back edge joins two vertices in a component of F, that is, in a rooted tree.

Theorem 2.44 Let G be a connected graph and T be a DFS tree of G, and let $e = xy$ be a back edge such that $\text{dfi}(x) < \text{dfi}(y)$. Then vertex x is an ancestor of vertex y in the tree T.

PROOF. Consider the first time that the DFS visits vertex x. By the assumption, the edge e is a back edge, this means that e never became a tree edge. Since $\text{dfi}(x) < \text{dfi}(y)$, y is visited after the first visit of x. In G we have $x \sim y$, but we know that $e = xy$ is not an edge of the DFS tree. This means that at the vertex x DFS chose a neighbor of x other than y to get visited after x, and by continuing along a walk through other vertices DFS eventually visited y. Since all the neighbours of each vertex v get visited before the last visit of v, y is visited by the DFS before the last

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

visit of x. So y is in the subtree rooted at x. Therefore x in an ancestor of y and y is a descendant of x. $\square$

Breadth-First Search

When a graph is traversed by visiting all the adjacent vertices of a vertex first, the traversal is called Breadth-First Search, abbreviated by BFS. This method starts at a vertex u and identifies all the vertices adjacent to it. After visiting all of the vertices adjacent to u, the algorithm visits all vertices adjacent to the neighborhoods of u that have not been visited before. This process continues until all vertices of the graph are visited. Upon termination of the algorithm, each vertex v acquires a *Breadth-First Search Index*, denoted by $\text{bfi}(v)$, which indicates the order in which the vertex v was visited. Since vertex u is visited first, we have $\text{bfi}(u) = 1$.

Algorithm 2.45 : *A Breadth-First Search of a graph* G

Input: Graph G

Output: Values of bfi for all vertices of G

begin

  for all vertices $v \in V(G)$ **do**

    $\text{bfi}(v) := 0$

  $i := 0$

  initialize the empty queue Q

  while there is a vertex u such that $\text{bfi}(u) = 0$ **do**

  begin

    add u to Q

    while Q is not empty **do**

    begin

      dequeue a vertex from Q and call it w

      if $\text{bfi}(w) = 0$ **then**

      begin

        $i := i + 1$

        $\text{bfi}(w) := i$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            for all vertices v ∈ N(w) do
                if bfi(v) = 0 then
                    add v to Q
        end
    end
end
output bfi(v) for all vertices v ∈ V(G)
end.
```

We will show that the complexity of the BFS algorithm is $O(max\{n, m\})$. The first "for" is of $O(n)$, since it runs on all vertices of the graph. The time spent by the second "for" is proportional to deg(v), since the "for" loop is executed for all of the neighbors of v. The check on the neighborhood happens for all vertices in the graph, and since $\sum_{w \in V(G)} \deg(w) = 2e$, the complexity of this process is $O(m)$. The output line takes $O(n)$, therefore, Algorithm 2.45 is of complexity $O(max\{n, m\})$.

2.3 Statistics

In this section we will present the definitions and results from statistics that will be used further in this thesis. Since the data that is used for this research comes from laboratory experiments, the use of statistical methods and results for its analysis is necessary. The data in StemBase comes from several samples of tissues, and therefore, we first describe sample mean, sample variance, and sample standard deviation.

Definition 2.46 Suppose that $x_1, x_2, ..., x_n$ are observations in a sample. The *sample mean*, denoted by $\bar{x}$, is defined as

$$\bar{x} = \sum_{i=1}^{n} \frac{x_i}{n} = \frac{x_1 + x_2 + ... + x_n}{n}.$$

Definition 2.47 The *sample variance*, denoted by s^2, is defined as

$$s^2 = \sum_{i=1}^{n} \frac{(x_i - \bar{x})^2}{n - 1}.$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The *sample standard deviation*, denoted by s, is equal to the positive square root of the sample variance, that is,

$$s = \sqrt{s^2}.$$

Definition 2.48 The set S of all possible outcomes of a statistical experiment is called a *sample space.*

Definition 2.49 A *random variable* X is a function from the sample space to the set of real numbers. We denote the range of X by $\text{Rng}(X)$.

A *discrete* random variable is a random variable with a finite or countably infinite range.

A *continuous* random variable is a random variable such that its range is not finite or countably infinite; such a range can be an interval or a union of intervals.

Definition 2.50 [39] A function f from $\text{Rng}(X)$ to $[0, 1]$ is called a *probability function* or *probability mass function* or *probability distribution* of the discrete random variable X if

1.$f(x) \geq 0$ for all $x \in \text{Rng}(X)$,

2.$\sum_x f(x) = 1$, and

3.$f(x) = P(X = x)$, where $P(X = x)$ is the probability that the random variable X takes the value x.

Definition 2.51 [39] The function $f(x)$ from $\text{Rng}(X)$ to $\mathbb{R}$ is called a *probability density function* for the continuous random variable X, if

1.$f(x) \geq 0$, for all $x \in \text{Rng}(X)$,

2.$\int_{-\infty}^{+\infty} f(x)dx = 1$, and

3.$P(a < X < b) = \int_a^b f(x)dx$.

Definition 2.52 The *mean* or *expected value* of a random variable X, denoted by μ or $E(X)$, is defined as

$$\mu = E(X) = \sum_x xf(x)$$

if X is discrete, and

$$\mu = E(X) = \int_{-\infty}^{+\infty} xf(x)dx$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

if X is continuous.

The *variance* of X, denoted by σ^2 or $V(X)$, is defined as

$$\sigma^2 = V(X) = E(X-\mu)^2 = \sum_x (x-\mu)^2 f(x)$$

if X is discrete, and

$$\sigma^2 = V(X) = E(X-\mu)^2 = \int_{-\infty}^{+\infty} (x-\mu)^2 f(x)dx$$

if X is continuous.

The *standard deviation* of X is denoted by σ and it is equal to the square root of the variance.

If we want to specify the mean, variance, or standard deviation of a random variable X, then we will write μ_X, σ_X^2, and σ_X, respectively.

The mean or expected value of a random variable is a way to describe where the probability function is centered. However, it does not provide us with information about the variability in the distribution. To have a better understanding about a distribution we can use variance. Variance is the most important measure of variability of a random variable X.

Lemma 2.53 The variance σ^2 of a random variable X is equal to

$$E(X^2) - \mu^2.$$

PROOF. Assume X is a discrete random variable. Then

$$\sigma^2 = E(X-\mu)^2 = \sum_x (x-\mu)^2 f(x) = \sum_x (x^2 - 2\mu x + \mu^2) f(x)$$

$$= \sum_x x^2 f(x) - 2\mu \sum_x x f(x) + \mu^2 \sum_x f(x).$$

Since by definition we have $\mu = \sum_x x f(x)$ and $\sum_x f(x) = 1$, we have

$$\sigma^2 = \sum_x x^2 f(x) - \mu^2 = E(X^2) - \mu^2.$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the case that X is a continuous random variable, the same proof is valid by substituting $\sum$ by $\int$. □

It is often useful to have a measurement for the relationship between two random variables. In order to be able to define and compute such a relationship we define joint probability distribution of two random variables, and expected value of a function of two random variables.

Definition 2.54 [39] The function f from $(\text{Rng}(X), \text{Rng}(Y))$ to $[0, 1]$ is called a *joint probability distribution* of the discrete random variables X and Y if

1. $f(x, y) \geq 0$ for all $(x, y) \in (\text{Rng}(X), \text{Rng}(Y))$,
2. $\sum_y \sum_x f(x, y) = 1$, and
3. $f(x, y) = P(X = x, Y = y)$.

Definition 2.55 [39] The function f from $(\text{Rng}(X), \text{Rng}(Y))$ to $\mathbb{R}$ is called a *joint density function* of the continuous random variables X and Y if

1. $f(x, y) \geq 0$ for all $(x, y) \in (\text{Rng}(X), \text{Rng}(Y))$,
2. $\int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y)dxdy = 1$, and
3. $P((X, Y) \in A) = \int \int_A f(x, y)dxdy$, for any region A in the xy plane.

Definition 2.56 Let h be any function of two random variables X and Y defined on a probability space with joint probability distribution f. The *mean* or *expected value* of a function h of these two random variables is defined as

$$E(h(X, Y)) = \sum_{x \in \text{Rng}(X)} \sum_{y \in \text{Rng}(Y)} h(x, y)f(x, y)$$

if X and Y are discrete, and

$$E(h(X, Y)) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} h(x, y)f(x, y)dxdy$$

if X and Y are continuous.

Remark 2.57 [28] $E(h(X, Y))$ can be thought of as the weighted average of $h(x, y)$ for each point in the range of (X, Y). The value of $E(h(X, Y))$ represents the average value of $h(X, Y)$ that is expected in a long sequence of repeated trials of the random experiment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2.58 The *covariance* between two random variables X and Y, denoted as $cov(X, Y)$ or σ_{XY} is defined to be

$$\sigma_{XY} = E((X - \mu_X)(Y - \mu_Y)).$$

Remark 2.59 [28] Covariance is a measure of linear association between the random variables. If the relationship between the random variables is nonlinear, then the covariance might not be sensitive to the relationship, and therefore, in that case the covariance is not a good measure of the relationship between the random variables.

Remark 2.60 [10] If large values of X tend to be observed with large values of Y and small values of X with small values of Y, then σ_{XY} will be positive. This is because if $X > \mu_X$, then $Y > \mu_Y$ is likely to be true. Therefore, the product $(X - \mu_X)(Y - \mu_Y)$ will be positive, and $\sigma_{XY} = E((X - \mu_X)(Y - \mu_Y)) > 0$. If large values of X are observed with small values of Y and small values of X with large values of Y, then σ_{XY} will be negative because when $X > \mu_X$, Y will tend to be less than μ_Y and vice versa. Therefore, $(X - \mu_X)(Y - \mu_Y)$ will tend to be negative, and $\sigma_{XY} = E((X - \mu_X)(Y - \mu_Y)) < 0$. This shows that the sign of covariance between two random variables can give us some information about the relationship between the two random variables.

One disadvantage of covariance is that the value of covariance does not indicate anything regarding the strength of the relationship. Therefore, we introduce a common statistical parameter called correlation.

Definition 2.61 The *correlation* of two random variables X and Y, denoted by ρ_{XY}, is

$$\rho_{XY} = \frac{cov(X, Y)}{\sqrt{V(X)V(Y)}} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y}.$$

Since σ_X and σ_Y are both positive, the denominator in the above definition is always positive. Therefore, if the covariance between X and Y is positive, negative, or zero, then the correlation of X and Y is positive, negative, or zero, respectively.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lemma 2.62 [10] For any two random variables X and Y the following hold.

1. $-1 \leq \rho_{XY} \leq 1$.
2. $|\rho_{XY}| = 1$ if and only if there exist numbers $a \neq 0$ and b such that $P(Y = aX + b) = 1$. If $\rho_{XY} = 1$, then $a > 0$, and if $\rho_{XY} = -1$, then $a < 0$.

By the above lemma, the correlation is always between -1 and 1. When the correlation of two random variables is -1 or 1, the correlation shows a perfect linear relationship between the two random variables (for more details look at [10]). In general, the closer $|\rho_{X,Y}|$ gets to 1, the stronger the linear relationship between X and Y. However, it should be mentioned that if the correlation of two random variables is zero, we can not deduce that there is no relationship between these two random variables. The reason is that the correlation only measures the existence of a linear relationship, and the two random variables might have a non-linear relationship.

Lemma 2.63 Let X, Y,Z be three random variables on the same sample space. Assume $|\rho_{XY}| = |\rho_{YZ}| = 1$. Then

$$\text{sign}(\rho_{XZ}) = \text{sign}(\rho_{XY})\text{sign}(\rho_{YZ}).$$

PROOF. Since $|\rho_{XY}| = 1$, by Lemma 2.62 we have $Y = aX + b$, for some $a \neq 0$ and b. Also, if $\rho_{XY} = 1$, then $a > 0$, and if $\rho_{XY} = -1$, then $a < 0$; that is, $\text{sign}(a) = \text{sign}(\rho_{XY})$. Similarly, since $|\rho_{YZ}| = 1$ we have $Z = a'Y + b'$, where $\text{sign}(a') = \text{sign}(\rho_{YZ})$. Hence, $Z = a'(aX + b) + b' = a'aX + a'b + b'$. Since a, a', b and b' are real numbers, and $a, a' \neq 0$, by the second statement of Lemma 2.62 we can conclude that $|\rho_{XZ}| = 1$, and $\text{sign}(\rho_{XZ}) = \text{sign}(a'a) = \text{sign}(a')\text{sign}(a) = \text{sign}(\rho_{XY})\text{sign}(\rho_{YZ})$. □

2.4 Molecular Biology

The expression of genes and the genetic information in a cell is a complicated process. Here, we just mention the main parts of a gene and their function in basic terms.

Deoxyribonucleic acid (DNA) is a nucleic acid that contains the genetic instructions for the development and functioning of living organisms. DNA carries the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

genetic information of a cell. DNA molecules are made up of a long, linear arrangement of subunits known as *nucleotides.* Each nucleotide has three parts : a phosphate group, a sugar, and a base that contains nitrogen. In DNA, the sugar is deoxyribose, which gives its name to DNA. There are four types of nucleotides that differ only in their base. These four types are adenine, thymine, guanine, and cytosine, represented by A, T, G, and C. Each DNA is a double stranded molecule such that its two strands are wound around each other to form a double helix. In DNA, the bases of one strand are paired with the bases of the other strand. In this structure, A pairs with T, and C with G, and they are locked in the interior of the double helix. Therefore, DNA has a stable structure that suits for storing the valuable information of a cell [4].

The *genome* of an organism is its whole hereditary information and is usually encoded in the DNA. The average mammal genome contains about 5×10^9 base pairs of DNA [40].

A *gene* is a locatable region of genomic sequence, corresponding to a unit of inheritance, which is associated with regulatory regions, transcribed regions and/or other functional sequence regions [29]. Genes, made from long strands of DNA, are units of information within the *chromosome* that can be inherited. Chromosomes are structures made of DNA that bear the genes of a cell. Each gene is responsible for an inherited property of an organism. Sometimes, two or more genes are involved in a certain property of a higher organism such as height. Therefore, the interaction of genes with each other forms the development and behavior of a cell.

DNA molecules in a cell carry the original genetic information, and thus, they are protected within the *nucleus.* Nucleus is an internal part of a cell that contains the genetic information. Only the cells of the higher organisms contain nucleus. Nucleus contains the chromosomes. Nucleus is separated from the rest of the cell by *nuclear membrane.* Whenever the genes need to communicate with the rest of the cell, a copy of the DNA is made. This copy carries the information from the nucleus to the rest of the cell. *Ribonucleic acid,* abbreviated by RNA, is another nucleic acid that can carry genetic information. RNA is normally single stranded. When the genetic information is carried from the genes to the rest of the cell, a particular type of RNA is used. *Messenger RNA* (shortly mRNA) is a type of RNA that DNA uses as a messenger to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

send the genetic information. mRNA can be viewed as a working copy of a particular sequence of DNA [40].

The process of transferring genetic information from DNA to mRNA is called *transcription*. When a gene is active, the coding sequence is copied in the process of transcription, producing an mRNA copy of the gene's information. This mRNA can then direct the synthesis of proteins in the cell via the genetic code. Proteins perform cellular functions and define cellular identity.

During transcription certain parts of the genome are copied. There might be several copies of any given section of the genome. Different parts of the genome are transcribed to different extents, and this process is regulated in each cell. Therefore, in different cells, or in the same cell at different times, different sets of genes might be transcribed. For example, two cells of a human body that have the same genetic information may transcribe different sets of genes, and therefore perform in different ways, for instance, one cell might act as a neuron and the other may become a skin cell [40]. Therefore, expressing different sets of genes is the cause of difference in performance of various cells. Not all genes are expressed in all cells all the time. Different cell types express different but overlapping subsets of genes. Usually for any two different cell types about less than half of the genes are expressed in both of them. A specific cell type can be defined by the expression of about 100-200 genes that are expressed in a certain way that gives the cell its unique identity [4].

Transcriptional regulation is the mechanism that coordinates the expression of DNA with the needs of various life processes such as development, gestation and metabolism. A *transcription factor* is a protein that acts as a regulator of gene expression. Different transcription factors regulate expression of different sets of genes, and thus, control the outcome of gene expression that distinguishes cells from each other [32].

Hence, studying gene expression in different cells reveals information that helps in better understanding of cellular functions and characteristics.

Microarray technology is a powerful technique that can be used for measuring gene expression levels for cells in different tissues of different organisms. Loosely speaking, expression microarrays measure the relative mRNA level over populations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of cells in multiple samples. The main assumption of gene expression microarrays is that the level of a given mRNA in a cell is positively correlated with gene expression levels [44]. DNA microarrays simultaneously measure gene expression and mRNA levels from genes [31].

A *gene expression profile* is the result of DNA microarray analyses, which give the breakdown of the switching on and off of certain genes. It answers the question of what genes are expressed in a particular cell type of an organism, at a particular time, under particular conditions. For instance, we can have a comparison of gene expression between normal and diseased (e.g., cancerous) cells.

According to [22], gene expression profiles specific for disease subgroups is the most important application of microarray analysis in medical science. One possible example of this application is using the results of microarray analysis for gene expression profiles to predict responses to specific treatments for different patients. Also, gene expression profiling helps in building gene models that can be used in finding treatment strategies for different diseases.

One reason that the expression levels of two genes might be highly correlated is that they are both regulated by the same transcription factors. Therefore, if we want to understand how various cells differ in transcriptional regulation, one approach would be to study their gene expression levels [17].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

Connectedness

One of the main questions about a graph is whether a given graph G is connected or not, and if it is not connected, finding the connected components of G is of interest. Answering these questions would give us a first impression of the structure of the graph, which can then be used for further study. The algorithms that are described in this chapter are used to determine whether a graph is connected or not.

If we have the adjacency matrix A of a graph, we can answer the question of connectivity by trying different permutations of rows and the corresponding columns of A, and then checking to see if the matrix obtained by simultaneously permuting the rows and columns of A is in a block-diagonal form. Clearly, this is not an efficient way, since it may involve $n!$ permutations where n is the number of vertices of the graph. Algorithm 3.1 below answers the question of connectivity in a more efficient way. This algorithm uses Depth-First Search method described in 2.2.4 in order to find the connected components of a graph. The second algorithm in this chapter, Algorithm 3.2, uses Breadth-First Search as a way to find the connected components.

Algorithm 3.1 [21]: *To find the connected components of a graph* G *by using DFS.*

Input: Graph G

Output: An array called component such that the value of component[v] is the same for all vertices v in one component.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
begin
  i := 0
  for all vertices v ∈ V(G) do
    visited[v]:=false
  create an empty stack S
  for all vertices r ∈ V(G) do
    if visited[v]=false then
    begin
      push r to S
      i := i + 1
      while S ≠ ∅ do
      begin
        pop a vertex from S and call it v
        if visited[v]=false then
        begin
          visited[v]:=true
          component[v] := i
          for all vertices w ∈ N(v) do
            if visited[w]=false then
              push w on S
        end
      end
    end
  output component[v] for all vertices v of G
end.
```

As we mentioned before, the above algorithm uses the Depth-First Search (DFS) method to find the connected components. Therefore, its complexity is equal to $O(max\{n, m\})$.

Breadth-First Search can also be used for finding the connected components of a graph. The next algorithm finds the connected components of a given graph by using Breadth-First Search.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 3.2 : *To find the connected components of a graph* G *by using BFS.*

Input: Graph G

Output: An array called component such that the value of component[v] is the same for all vertices v in one component.

```
begin
    create an empty queue Q
    i := 1
    for all vertices v ∈ V(G) do
        visited[v]:=false
    for all vertices v ∈ V(G) do
        if visited[v]=false then
        begin
            visited[v]:=true
            component[v] := i
            add v to Q
            while Q is non-empty do
            begin
                dequeue first vertex from Q and call it y
                for all vertices w ∈ N(y) do
                    if visited[w]=false then
                        begin
                            visited[w]:=true
                            component[w]:=i
                            add w to Q
                        end
            end
            i := i + 1
        end
    output component[v] for all vertices v of G
    end.
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since Algorithm 3.2 is BFS with the addition of detecting the connected components, the complexity of it is same as the complexity of BFS; that is $O(\max\{n, m\})$.

Once we determine the connected components of a graph we can continue with other algorithms that find out different parameters in a graph such as distance, eccentricity, radius, diameter, blocks, cut-vertices, etcetera. Chapters 4-7 discuss various algorithms for determining these parameters. In these chapters we assume that all the graphs we are dealing with are connected (unless we mention that the graph is disconnected). Clearly, if we have a disconnected graph, then the algorithms discussed in the following chapters can be applied to each connected component of the graph.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

Bipartiteness

Bipartite graphs have a special structure that is of interest in many applications. Also, many computationally hard problems are polynomial for bipartite graphs. Therefore, one of the primary questions about a graph is whether a given graph is bipartite. In this chapter we will discuss an algorithm that, for a given graph G, determines whether it is bipartite.

One might think of an algorithm to test for bipartiteness which follows the definition of a bipartite graph. That is, an algorithm that constructs every possible partition of the vertex set V into parts V_1 and V_2, and tests whether or not the graph is bipartite relative to this partitioning. Such an algorithm would have to test an exponential number of partitions, which makes it inefficient.

Similarly, despite of its mathematical appeal, the statement of Theorem 2.34 does not provide an efficient way to test whether a graph is bipartite. An algorithm based on Theorem 2.34 that checks whether every cycle has even length would be very inefficient. This is because of the large number of cycles that have to be tested, since the number of cycles is typically exponential in the number of vertices in a graph. However, the proof of Theorem 2.34 is the basis of an efficient algorithm for determining whether a graph is bipartite or not.

We now describe an efficient algorithm to determine whether a graph is bipartite. The idea is to start from an arbitrary vertex v and put it into part V_1, then proceed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to the neighbors of v and assign them to part V_2, and so on. Obviously, the graph is not bipartite if and only if some vertex is forced into both parts. So the algorithm traverses the vertices of the graph and labels each vertex by 0,1, or 2 according to whether it is still unvisited, belongs to part 1, or part 2. If there is an edge between two vertices of the same part, then the graph is not bipartite and the algorithm stops. The algorithm takes $O(n+m)$ time. Assuming G is connected, $O(n+m)$ simplifies to $O(m)$. Note that for a given bipartite graph G, we can assign two colors black and white to the vertices of G according to the part they belong to. That is, all vertices in the same part should have the same color and each edge would have one of its ends of color white and the other end of color black.

Algorithm 4.1 : *To determine whether a graph is bipartite.*

Input: Graph G

Output: "Yes" if G is bipartite, "no" if it is not.

begin

  choose an arbitrary vertex s

  for each vertex $u \in V(G) - s$ **do**

    part[u]:=0

  part[s]:=1

  initialize queue Q as empty

  add s to the queue Q

  while Q is not empty **do**

  begin

    dequeue the first vertex from Q and call it u

    for each vertex $v \in N(u)$ **do**

    begin

      if part[u]=part[v] **then**

        Output "The graph is not bipartite" and stop.

      else

        if part[v]=0 **then**

        begin

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

part[v] :=3 - part[u]
add v to the queue Q
end
end
end
Output "The graph is bipartite".
end.

We should mention here that Algorithm 4.1 is essentially Breadth-First Search on the graph with the addition of the necessary commands to partition the graph into two parts, if possible.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

Distance

This chapter studies the concept of distance in graphs. Distance plays an important role in some of the graph theory problems that have extensive real world applications. Finding a shortest path between two vertices, or finding the center, radius, and diameter of a graph are examples of problems that deal with distance. First, we study the notion of distance in graphs without weights. We present an algorithm that finds the distance between all pairs of vertices of a graph, and prove its correctness. In the second section of this chapter, we give the definition of distance in weighted graphs and we discuss the famous Dijkstra Algorithm that finds distance in weighted graphs. The next part of this chapter is dedicated to introducing the following parameters: eccentricity, radius, diameter, center, and median. After defining these parameters for a graph without weights, we will discuss algorithms that can be used to find these parameters. Finally, we define distance of a vertex from a subgraph and eccentricity of a subgraph, and we present algorithms to compute these parameters. Throughout this chapter, we assume G is a connected graph.

5.1 Distance in Graphs without Weights

Definition 5.1 Let G be a graph, and u and v two vertices of G. The *distance* between u and v, denoted by $\text{dist}(u, v)$, is the length of a shortest (u, v)-path in G.

It is easy to see (Theorem 5.2) that distance is metric.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 5.2 Let G be a graph. Then we have the following:

(i) For all vertices $u, v \in$ V(G) we have $\text{dist}(u, v) \geq 0$, and $\text{dist}(u, v) = 0$ if and only if $u = v$.

(ii) For all vertices $u, v \in$ V(G) we have $\text{dist}(u, v) = \text{dist}(v, u)$.

(iii) (Triangle Inequality) For any three vertices $u, v, w \in$ V(G) we have

$$\text{dist}(u, v) + \text{dist}(v, w) \geq \text{dist}(u, w).$$

PROOF. (i) By definition, the length of a path is a non-negative number. So for any two vertices u, v we have $\text{dist}(u, v) \geq 0$. The length of a $(u, v)-$path is zero if and only if it has no edges, that is, if and only if $u = v$.

(ii) Any $(u, v)-$path gives rise to a $(v, u)-$path of the same length in a natural way, that is, by reversing the order of vertices in it.

(iii) Let $P = uv_1v_2...v_kv$ be a shortest (u, v)-path and $Q = vw_1w_2...w_kw$ be a shortest (v, w)-path. Then $uv_1v_2...v_kvw_1w_2...w_kw$ is a (u, w)-walk in G. Since the length of any shortest (u, w)-path in G is less than or equal to the length of any (u, w)-walk, we have $\text{dist}(u, v) + \text{dist}(v, w) \geq \text{dist}(u, w)$. □

One of the parameters that we are interested in is the distance of each vertex from a distinguished vertex. The following algorithm calculates the distance of all of the vertices of G from a fixed vertex u. In this algorithm M is a number greater than the maximum distance between any two vertices in the graph. For computational purposes we can choose M to be $|\text{V(G)}|$ (recall that G is assumed to be connected). The array l contains the distance from u for each vertex v. Initially, we let $l(v) = M$ for all $v \in \text{V(G)} - \{u\}$. We start from u, and all of the vertices that are adjacent to u (except u itself) will be of distance one from u, so for these vertices in the array we have $l(v) = 1$. Then all the vertices that are adjacent to these vertices and $l(v) = M$ for them (they have not previously been visited) will be of distance two, and so on. This process continues until all vertices of G are visited.

Algorithm 5.3 : *To find* dist(u, v) *for a fixed vertex* u *in a connected graph* G *and every vertex* v *of* G

Input: Graph G and vertex u

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Output: Array l of distances of vertices of G from the vertex u
Distance(u,l)
begin
   for all vertices v of G do
      l(v) := M
   l(u) := 0
   add u to queue Q
   while Q is non-empty do
   begin
      delete a vertex x from Q
      for every vertex y ∈ N(x) do
         if l(y) = M then
         begin
            l(y) := l(x) + 1
            add y to Q
         end
   end
   output (v, l(v)) for all vertices v of G
end.
```

Remark 5.4 Algorithm 5.3 is in fact a Breadth-First Search in the graph.

When Algorithm 5.3 terminates, $l(v)$ is the distance of a vertex v from the vertex u. The first "for" requires $O(n)$ basic operations. Within the "while" loop, the algorithm checks all neighbors of each vertex v, therefore, the loop takes time proportional to $\sum_{v \in V(G)} \deg(v)$, which is of the order $O(m)$. The output line is of the order $O(n)$. Hence Algorithm 5.3 has complexity of $O(\max\{n, m\})$. Since G is connected, this simplifies to $O(m)$.

It can be seen immediately that Algorithm 5.3 calculates an upper bound for the distance between vertex u and any other vertex v. This is clear from Theorem 5.2 part(iii), the triangle inequality. We need to show that the distances that we get from executing Algorithm 5.3 are in fact distances in the graph G.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Lemma 5.5 If x and y are any two vertices in G such that x is (added and) deleted from the queue Q before y, then $l(x) \leq l(y)$.

PROOF. By strong induction. Let $v_1, v_2, ..., v_n$ be the order in which the vertices were added to and therefore deleted from Q. It suffices to prove that $l(v_i) \geq l(v_j)$ for all $j < i$, for $i = 2, ..., n$. For the basis of the induction let $i = 2$. So, there are two vertices v_1 and v_2 in the queue such that v_1 is the first vertex that was added to the queue and v_2 is added after v_1. Therefore, v_1 is adjacent to v_2, and we have $l(v_2) = l(v_1) + 1 > l(v_1)$.

Now assume that for some $i \in \{2, ..., n\}$, the statement is true for all vertices $v_1, v_2, ..., v_i$; that is, $l(v_i) \geq l(v_j)$ for all $j < i$. It suffices to show $l(v_{i+1}) \geq l(v_i)$. From the algorithm we know that v_i was added as a neighbour of a vertex, say v_k, and v_{i+1} was added as a neighbour of a vertex, say $v_{k'}$ (possibly $k = k'$, but necessarily $k \leq k' \leq i$). So we have $l(v_i) = l(v_k) + 1$ and $l(v_{i+1}) = l(v_{k'}) + 1$. By induction hypothesis, since $k \leq k' \leq i$, we have $l(v_k) \leq l(v_{k'})$. Therefore, $l(v_i) \leq l(v_{i+1})$. Hence, by the principle of induction, $l(v_i) \geq (v_j)$ for all vertices v_i and v_j such that $j < i \leq n$; that is, for all vertices v_i and v_j such that v_j is deleted before v_i. □

Theorem 5.6 When Algorithm 5.3 terminates, the values of $l(v)$ for vertices $v \in V(\mathrm{G})$ are equal to $\mathrm{dist}(u, v)$ in the graph G.

PROOF. By contradiction. Assume that vertex $w \in V(\mathrm{G})$ is the closest vertex to u such that $l(w) \geq \mathrm{dist}(u, w)$. So for all vertices that appear before w on a shortest path from u, the value of l is in fact their distance from u. Let v be the last vertex before w on a shortest (u, w)-path. Therefore we have $\mathrm{dist}(u, w) = \mathrm{dist}(u, v) + 1$. Since v appears before w on a shortest path, we have $l(v) = \mathrm{dist}(u, v)$. Let y be the vertex that was responsible for adding w to the Q, so $l(w) = l(y) + 1$. Now we have

$$l(y) + 1 = l(w) > \mathrm{dist}(u, w) = \mathrm{dist}(u, v) + 1 = l(v) + 1$$

Therefore, we have $l(y) > l(v)$. So, according to Lemma 5.5, vertex v must have been added to the queue Q before vertex y. Thus, w should have been added to Q as a neighbour of v rather than y. This is a contradiction. Therefore, since $l(w)$ can not be less than $\mathrm{dist}(u, w)$, we conclude that $l(w) = \mathrm{dist}(u, w)$. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.2 Distance in Weighted Graphs

For a connected weighted graph G with weight function $w : \mathrm{E(G)} \to \mathbb{R}$ and $v_0, v_k \in \mathrm{V(G)}$ let $P = v_0v_1...v_k$ be a (v_0, v_k)-path. By the *length* of P in the weighted graph G we mean $w(P) = \sum_{i=1}^{k} w(v_{i-1}v_i)$. Therefore, a shortest (v_0, v_k)-path is a $(v_0, v_k)-$path P such that $w(P)$ is minimum. By the *distance* from a vertex u to a vertex v in a weighted graph G, $\mathrm{dist}(u, v)$, we mean the length of a shortest (u, v)-path in G.

Let G be a weighted graph and H be the graph obtained from G by removing the edge weights. Then $\mathrm{dist}(u, v)$ in G can be different from the $\mathrm{dist}(u, v)$ in H because a shortest (u, v)-path in G need not be a shortest (u, v)-path in H (and vice-versa).

If in a graph G the weights are all equal to 1, then the distance between a vertex u and any other vertex in G can be found by the same algorithm that finds the distance in graphs without weights (See Algorithm 5.3). But, if the weights are different from 1, then we can not use Algorithm 5.3 to find distance in weighted graphs. A famous algorithm that finds distance (length of a shortest path) in a weighted graph is due to Edsger Dijkstra. The Dijkstra algorithm assumes that all edge weights are nonnegative. Dijkstra's algorithm finds the length of a shortest path from a given vertex u to every other vertex in a weighted connected graph. If the graph has negative weights, then we can use another algorithm which is known as Floyd-Warshall's algorithm.

5.2.1 Dijkstra's Algorithm

Dijkstra's algorithm was first published in 1959. During the algorithm, each vertex v has an associated label $L(v)$, which is an upper bound for $\mathrm{dist}(u, v)$. We take the convention that $w(uv) = M$ if $u \not\sim v$. Initially $L(u) = 0$ and $L(v) = M$ for all vertices $v \neq u$. When Dijkstra's algorithm terminates, the value of the label $L(v)$ for each vertex v is $\mathrm{dist}(u, v)$. At each step the algorithm constructs a set $T \subseteq \mathrm{V(G)}$ such that a shortest path from u to each vertex $v \in T$ passes only through vertices in T. That means that at every stage for vertices v in the set T we have $L(v) = \mathrm{dist}(u, v)$. Among all the vertices that are not in T, the algorithm finds a vertex v' that has the minimum label, that is, for all the vertices not in T, we have $L(v') \leq L(v)$. Then, for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

all vertices v not in T, if $L(v') + w(v'v)$ is less than $L(v)$, the algorithm substitutes $L(v)$ by $L(v) + w(v'v)$. The algorithm terminates when the set T contains all of the vertices of G.

Algorithm 5.7 *Dijkstra's Algorithm : To find the length of a shortest path from vertex u to every other vertex.*

Input: Connected weighted graph G with non-negative weights, and vertex u

Output: Array L containing the length of a shortest path from u to all other vertices of G

```
begin
    for all vertices v ≠ u do
        L(v) := w(uv)
    L(u) := 0
    T := {u}
    while T ≠ V(G) do
    begin
        find a vertex v' ∉ T such that for all v ∉ T we have L(v') ≤ L(v)
        T := T ∪ {v'}
        for all v ∉ T do
            if (L(v') + w(v'v)) < L(v) then L(v) := L(v') + w(v'v)
    end
    output L(v) for all v ∈ V(G)
end.
```

Since the Dijkstra algorithm is more complicated than the other algorithms that we discussed before, we will show its correctness. To help with the proof, we introduce the notion of a Dijkstra tree, constructed as follows.

Let S be an empty set. Starting at the vertex u, at each iteration of Dijkstra's algorithm, we add an edge to the set S that is not already in S. The way we choose this edge is by Dijkstra algorithm. Note that at each step, for all $v \notin T$, $L(v) = \min\{L(y) + w(yv) : y \in T\}$. Therefore, at each iteration, an edge $e = xv'$ is added to the set S such that $L(v') = \min\{L(v) : v \notin T\}$ and $L(v') = L(x) + w(xv')$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for $x \in T$. Note that one end of e is already an end on an edge in S, and the other end of e is as close as possible to y. Hence, the set S induces a subgraph that is a tree. Such a tree is called a *Dijkstra tree.*

Theorem 5.8 Let G be a connected graph and T_j be the Dijkstra tree after the j-th iteration of Dijkstra's algorithm for $j = 0, 1, ..., |\text{V(G)}| - 1$. Then for each vertex v in T_j, the unique (u, v)-path in T_j is a shortest (u, v)-path in G.

PROOF. By induction. Clearly, the statement is true for T_0, the tree containing just vertex u. Assume that for some j, $0 \leq j < |\text{V(G)}| - 1$, the theorem is true. We need to show that T_{j+1} satisfies the property. Assume that $v' \notin \text{V}(T_j)$ such that for all vertices s not in T_j we have $L(v') \leq L(s)$. Therefore, Dijkstra's algorithm selects the vertex v' to be added to T_j. Let $x \in \text{V}(T_j)$ be such that the edge $e = xv'$ is added in the $j+1$ iteration to T_j to form T_{j+1}. Let Q be the (u, v')-path in T_{j+1} that is formed by adding the edge xv' to the unique (u, x)-path in T_j. Notice that $l(Q) = L(v')$.

Since v' is the only new vertex in T_{j+1}, it is enough to show that the (u, v')-path Q in T_{j+1} is a shortest (u, v')-path in G.

Let R be any (u, v')-path in G. We will show that $l(R) \geq l(Q)$.

Let v be the first vertex in R such that v has a neighbor z on R that is not a vertex of T_j, and let $f = vz$. Let K be the portion of R from z to v'. Note that within Algorithm 5.7 at each step, for all vertices p not in the current tree, we have $L(p) = \min\{L(q) + w(pq) : q \text{ is in the current tree}\}$. Hence $L(v) + w(vz) \geq L(z)$. Dijkstra's algorithm selected the edge e to be added in the $j+1$ iteration; therefore, $L(z) \geq L(v')$. Hence, $L(x) + w(xv') = L(v') \leq L(z) \leq L(v) + w(vz)$. Therefore,

$$l(R) = L(v) + w(vz) + l(K) \geq L(x) + w(xv') + l(K).$$

Since the weights are non-negative, we have

$$l(R) \geq L(x) + w(xv') = l(Q).$$

Therefore, the unique (u, v')-path Q in T_{j+1} is a shortest (u, v')-path in G. The theorem follows by induction. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In order to determine the complexity of the Dijkstra algorithm, note that the computations that take place within the "while" loop, totalled over all iterations, need $n(n-1)/2$ additions and $n(n-1)$ comparisons. The decision for determining whether a vertex belongs to T or not, can be done in $O(n^2)$. Therefore, the algorithm is of order $O(n^2)$.

If we need to find the distance between every pair of vertices in a weighted graph, we can run Dijkstra's algorithm for all vertices of the graph, instead of just one vertex u. Therefore, the complexity increases by a factor of n to $O(n^3)$.

5.2.2 Floyd-Warshall's Algorithm

This algorithm was introduced independently by R.W.Floyd and S.Warshall in 1962. The algorithm finds the distance between all pairs of vertices in a weighted graph. Initially, for each pair of adjacent vertices v_i and v_j, the distance is the weight of the edge v_iv_j, and for all other pairs of vertices, the distance is set as infinity. At each iteration k, the algorithm finds the shortest distance from v_i to v_j via the vertices $v_1, v_2, ..., v_k$. Therefore, at the termination of the algorithm, the distance between every pair of vertices has been found.

Algorithm 5.9 [19] *Floyd Warshall's Algorithm : To find the length of a shortest path between every two pair of vertices in a graph.*

Input: Connected weighted graph G

Output: dist(i, j) for all vertices i, j of G

begin

  for $i := 1$ to n **do**

    $d(i,i) := 0$

  for $i := 1$ to n **do**

    for $j := 1$ to n **do**

      if $v_i \sim v_j$ **then**

        $d(i,j) := w(v_iv_j)$

      else

        $d(i,j) := M$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                d(i, j) := min{d(i, j), d(i, k) + d(k, j)}
    output d(i, j) for all v_i, v_j ∈ V(G)
end.
```

The Floyd-Warshall algorithm has complexity of $O(n^3)$. This is because of the three nested loops, each of complexity $O(n)$.

5.3 Eccentricity, Radius, Diameter, Center and Median for Graphs without Weights

The goal of this section is to study important graph parameters that are related to distance. We define the parameters that we are interested in, and for each one of them we give a polynomial algorithm to compute it.

An important parameter in our research is eccentricity. Eccentricity of each vertex v in a graph gives us the distance from v to the farthest vertex from v in the graph.

Definition 5.10 Define eccentricity of a vertex v of a graph G to be

$$\mathrm{ecc}(v) = \max\{\mathrm{dist}(v, u) : u \in \mathrm{V(G)}\}.$$

Once we apply Algorithm 5.3, we can find the eccentricity $\mathrm{ecc}(u)$ of a vertex u by the following algorithm.

Algorithm 5.11 : *To find eccentricity of a single vertex of a graph*

```
Input: Graph G and vertex u ∈ V(G)
Output: Eccentricity e of a vertex u
Eccentricity (u, e)
begin
    e := 0
    Distance(u, l)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for all vertices v of G **do**
if $l(v) > e$ **then**
$e := l(v)$
output e
end.

Since Algorithm 5.3 has complexity of $O(\max\{n, m\})$, Algorithm 5.11 is of $O(\max\{n, m\})$ too. Since we are assuming G is connected, Algorithm 5.11 has complexity $O(m)$.

Diameter and radius of a graph are the next parameters we are interested in.

Radius equals to the eccentricity of a vertex v such that the maximum distance from v to other vertices in the graph is minimum. Therefore, radius is the minimum of the eccentricities of all vertices in a graph. Diameter is the largest distance that occurs in a graph. If a graph is disconnected, then radius and diameter are equal to infinity.

Definition 5.12 The *diameter* of a graph G, denoted by diam(G), is the maximum distance between two vertices of G.

Definition 5.13 The *radius* of the graph G, denoted by rad(G), is defined as

$$\mathrm{rad}(G) = \min\{\mathrm{ecc}(v) : v \in V(G)\}.$$

The following algorithm finds the diameter and radius of a graph G by using the distance and eccentricity algorithms.

Algorithm 5.14 :*To find the diameter and radius of a graph* G.
Input: Graph G
Output: Radius and diameter of G
% l is the array of distances, ecc is the array of eccentricities, and n is the number of vertices of G. *%*
Rad-Diam(rad,diam)
begin
for $i := 1$ to n **do**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        ecc(i) := 0
    for i := 1 to n do
    begin
        Eccentricity(v_i, e)
        ecc(i) := e
    end
    rad:=M
    diam:= 0
    for i := 1 to n do
    begin
      if ecc(v_i) < rad then
          rad:=ecc(v_i)
      if ecc(v_i) > diam then
          diam:=ecc(v_i)
    end
    output(rad,diam)
end.
```

In order to determine the complexity of Algorithm 5.14 we notice that the Algorithm needs to compute eccentricity of all vertices of the graph. Therefore, it takes time proportional to $n \cdot O(\max\{n, m\})$. For a connected graph the complexity would simplify to $O(n \cdot m)$.

Another concept related to the ones above is the center of a graph.

Definition 5.15 The *center* of a graph G, denoted by $C(\mathrm{G})$, is the subgraph induced by those vertices of G whose eccentricity equals the radius of G.

That means a vertex is in the center of G if its greatest distance from any other vertex is as small as possible. The center of a graph is an important parameter in many applications. For example, in problems that deal with facility locations, a graph can be used as a model; vertices would then represent different locations and edges the roads between the locations. Then, the center of the graph contains the ideal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

locations for emergency facilities such as a police station or fire station. Here, the eccentricity of a vertex v represents the response time from v to a vertex farthest from v. Clearly, the goal for an emergency facility is to minimize the response time. Therefore, the vertices in the center of a graph would give us a good set of candidates for emergency facilities [9].

Algorithm 5.16 *: To find the center of a graph* G *by using Algorithms* 5.11 *and* 5.14

Input: Graph G
Output: Vertex set of the center of G
Center(ecc, radius, center)
begin
  for $i := 1$ to n **do**
    Eccentricity(v_i, e)
    $ecc(v_i) := e$
  end
  Rad-Diam(rad,diam)
  initialize the set C as empty
  for $i := 1$ to n **do**
    if $ecc(v_i) =$ rad **then**
      add v_i to the set C
  output the set C
end.

Since Algorithm 5.14 is of $O(n \cdot m)$, the complexity of the above algorithm is equal to $O(n \cdot m)$, too.

Definition 5.17 A graph G is called *self-centered* if every vertex of G is in the center.

From the above definition it is clear that for a self-centered graph, the radius is equal to the diameter. All complete graphs are self-centered.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Algorithm 5.18 : *To determine if a graph* G *is self-centered by using Algorithm* 5.14

Input: Graph G

Output: True if and only if G is self-centered, false otherwise

```
Self-Center(self-centered)
begin
    self-centered:=false
    Rad-Diam(rad,diam)
    if rad=diam then
        self-centered:=true
    output(self-centered)
end.
```

Since it takes $O(n \cdot m)$ to compute radius and diameter of a graph, the complexity of the above algorithm is equal to $O(n \cdot m)$, too.

Median is another parameter that we are interested in.

Definition 5.19 The *total distance* of a vertex v in a graph G is the sum of the distances from v to each vertex of G.

Definition 5.20 The *median* $M(\mathrm{G})$ of a graph G is the set of vertices of G with minimum total distance.

The following algorithm computes the median of a graph.

Algorithm 5.21 : *To find the median of a connected graph* G *by using Algorithm* 5.3

Input: Connected graph G

Output: Median of G

```
Median(M)
begin
    initialize the set M as empty
    for all vertices u in V(G) do
    begin
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Distance(u, l)
        total-distance(u) := 0
        for all vertices v ∈ V(G) do
            total-distance(u) := total-distance(u) + l(v)
    end
    min-distance:= (n − 1)^2
    for all vertices v ∈ V(G) do
        if total-distance(v)< min-distance then
            min-distance:= total-distance(v)
    for all vertices v ∈ V(G) do
        if total-distance(v)= min-distance then
            add v to set M
    output the set M
end.
```

Example 5.22 Figure 5 shows a graph G with vertices a, b, c, d, e, f, g. For this graph we have the following. $\text{ecc}(a) = \text{ecc}(b) = \text{ecc}(g) = 4$, $\text{ecc}(c) = \text{ecc}(e) = \text{ecc}(f) = 3$, and $\text{ecc}(d) = 2$. Therefore, the radius of G is 2 and the diameter is 4. The subgraph induced by the vertex d is the center of G, and the median of G is $\{d\}$.

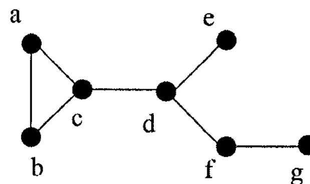

Figure 5

5.4 Distance and Eccentricity for a Subgraph

In the previous sections, we have defined distance between a pair of vertices and eccentricity of a single vertex in a graph. Similarly, distance of a vertex from a subgraph and eccentricity of a subgraph can be defined.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 5.23 Let G be a graph and H a subgraph of G. For any vertex $v \in V(G)$, the *distance from* v *to* H, denoted by $\text{dist}(v, H)$, is the minimum distance from v to a vertex in H.

The *eccentricity of* H, denoted by ecc(H), is the distance to H from a vertex farthest from H, so

$$\text{ecc}(H) = \max\{\text{dist}(v, H) | v \in V(G)\}.$$

In order to compute $\text{dist}(v, H)$ for a subgraph $H \subseteq G$ and a vertex $v \in V(G)$, we use Algorithm 5.3 to find the distance between the vertex v and every vertex in H. The next algorithm calculates $\text{dist}(v, H)$ for a subgraph $H \subseteq G$.

Algorithm 5.24 : *To find* $\text{dist}(v, H)$ *for a fixed vertex* v *in a connected graph* G *and a subgraph* $H \subseteq G$

Input: Graphs G and H and vertex v
Output: $\text{dist}(v, H)$
Distance-from-subgraph(v, *min*)
begin
  Distance(v, l) for the graph G
  $min := M$
  for every vertex $u \in V(H)$ **do**
    if $l(u) < min$ **then**
      $min := l(u)$
  Output *min*
end.

Since Algorithm 5.24 uses Algorithm 5.3, its complexity is equal to $O(m)$.

The next algorithm finds the eccentricity of a subgraph H of G.

Algorithm 5.25 : *To find* ecc(H) *for a connected subgraph* H *of* G

Input: Graphs G and H
Output: ecc(H)
Eccentricity-subgraph(*ecc*)
begin

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    for all vertices v of G do
        Distance-from-subgraph (v, H)
    ecc := 0
    for all vertices v of G do
        if dist(v, H) > ecc then
            ecc := dist(v, H)
    Output ecc
end.
```

Algorithm 5.25 calls Algorithm 5.24 for all vertices of the graph, therefore, the complexity of Algorithm 5.25 is $O(n \cdot m)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Blocks and Cut-Vertices

This chapter studies a method for finding cut-vertices and blocks in a connected graph. If our graph is not connected, then we can apply the algorithms described in this chapter to each connected component in order to find the blocks and cut-vertices.

The first section of this chapter is about finding cut-vertices in a graph. We discuss some important theorems that can help us characterize cut-vertices. Finding cut-vertices of a graph is in close connection with finding blocks of that graph. In the second section of this chapter we describe a method that finds both cut-vertices and blocks of a graph by using theorems and results from the first section. An algorithm for finding blocks and cut-vertices of a graph is described.

6.1 Cut-Vertices

In order to find blocks in a graph, we need to first determine the cut-vertices. There are two major results that characterize cut-vertices in a graph. Clearly, if a vertex is a cut-vertex of a graph, then it is a cut-vertex of some component of the graph and vice-versa. Hence, we can focus on finding cut-vertices in a connected graph.

Lemma 6.1 [21] A vertex v in a loopless connected graph is a cut-vertex if and only if there exist two distinct vertices u and w, both different from v, such that every (u, w)-path contains v.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PROOF. ($\Rightarrow$) Let v be a cut-vertex of G. Then by Remark 2.26, G $- v$ is disconnected, and therefore, there are vertices u and w in different components of G $- v$. So v must be on every path from u to w in G.

($\Leftarrow$) If u and w are two distinct vertices of G such that every (u, w)-path in G contains v, then in the graph G $- v$ there is no path between u and w. Therefore, u and w are in two different components of G $- v$, and G $- v$ is disconnected. So v is a cut-vertex of G. □

Theorem 6.2 [21] Let T be a DFS tree of a connected graph G with $|V(G)| > 1$, and let r be the root of T. Then r is a cut-vertex of G if and only if r has more than one child in T.

PROOF. ($\Rightarrow$) Assume r is a cut-vertex of G and suppose that r has just one child v in T. Then all other vertices are descendants of v. So the subtree rooted at v is a spanning tree of G $- r$, and G $- r$ is connected. This contradicts the assumption that r is a cut-vertex of G (see Remark 2.26). Hence r must have more than one child in T.

($\Leftarrow$) Assume that the root r has two children x and y. We want to show that r is a cut-vertex of G. We will show that every (x, y)-path contains r. Since x and y are both children of r, neither x nor y is an ancestor of the other. Also, no descendant of x is an ancestor or descendant of any descendant of y in T. Then by Theorem 2.44 there is no back edge joining a vertex in the subtree rooted at x to any vertex in the subtree rooted at y. Therefore, every (x, y)-path in G must go through r. Hence by Lemma 6.1, r is a cut vertex of G. □

The next theorem categorizes cut-vertices that are not a root of a DFS tree.

Theorem 6.3 [21] Let T be a DFS tree of G and v be a vertex of T other than the root r. Then v is a cut-vertex of G if and only if v has a child w such that no descendant of w is joined to an ancestor of v by a back edge.

PROOF. ($\Rightarrow$) Assume v is a cut-vertex of G and suppose that every child w of v has a descendant joined to an ancestor of v by a back edge. Then we claim that any two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vertices in the subtree rooted at v are joined by a path that does not go through v. Let x and y be two descendants of v. Then either x is a child of v, say w_1, or x is a descendant of a child w_1 of v. Similarly, y is a child of v, say w_2, or y is a descendant of a child w_2 of v. By the assumption, w_1 has a descendant d_1 that is joined to an ancestor of v, say a_1, by a back edge. Similarly, w_2 has a descendant d_2 joined to an ancestor of v, say a_2, by a back edge. Then the concatenation of the (x, w_1)-path, the (w_1, d_1)-path, edge d_1a_1, (a_1, a_2)-path, edge a_2d_2, (d_2, w_2)-path, and the (w_2, y)-path forms an (x, y)-walk that does not contain v. Therefore, there exists an $(x, y)-$path that does not contain v (Lemma 2.14).

Similarly, we will show that there is a path between any ancestor of v (not equal to v) and any descendant of v (not equal to v) that does not contain v. Let a_1 be an arbitrary ancestor of v and d be an arbitrary descendant of v. Then either d is a child of v, say w, or d is a descendant of a child w of v. By the assumption, w has a descendant d_1 that is joined to an ancestor of v, say a_2, by a back edge. Then the concatenation of the (d, w)-path, (w, d_1)-path, edge d_1a_2, and (a_2, a_1)-path is a (d, a_1)-walk that does not contain v. Therefore, there is a (d, a_1)-path that does not contain v.

Now, there is path from a_1 to the root r. So there is a (d, r)-walk that avoids v. From the root r, there is a path to any vertex in any subtree of r. Hence, there is a walk from d to any vertex in a different subtree of r that avoids v. So by Lemma 6.1 v is not a cut-vertex.

($\Leftarrow$) Assume that v has a child w such that no descendant of w is joined to an ancestor of v by a back edge. Then every path in G between w and r must go through v. Hence, by Lemma 6.1 v is a cut-vertex of G. □

Definition 6.4 Define *low value* of a vertex w in a DFS tree T, denoted by low(w), to be the smaller of dfi(w) and the smallest dfi among all vertices joined by a back edge to some descendant of w (including w).

Now Theorem 6.3 can be stated as follows:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Theorem 6.5 Let G be a connected graph and T be a DFS tree of G. Then a non-root vertex v of T is a cut-vertex of G if and only if v has a child w such that $\text{low}(w) \geq \text{dfi}(v)$.

PROOF. Let T be a DFS tree of G and v a non-root vertex.

($\Rightarrow$) Assume v is a cut-vertex of G. Hence, by Theorem 6.3, v has a child w such that no descendant of w is joined to an ancestor of v by a back edge. We will show that $\text{low}(w) \geq \text{dfi}(v)$. Suppose not; that is, suppose $\text{low}(w) < \text{dfi}(v)$. Since w is a child of v, we have $\text{dfi}(v) < \text{dfi}(w)$. Hence, from the definition of $\text{low}(w)$ we can conclude that $\text{low}(w) = \text{dfi}(x)$ for some vertex x that is joined by a back edge to some descendant y of w. Now since $\text{low}(w) < \text{dfi}(v)$, we have $\text{dfi}(x) < \text{dfi}(v)$. Therefore, x is an ancestor of v (recall that G is connected). So, an ancestor x of v is joined to a descendant y of w by a back edge. This contradicts Theorem 6.3. Therefore, $\text{low}(w) \geq \text{dfi}(v)$.

($\Leftarrow$) Assume v has a child w such that $\text{low}(w) \geq \text{dfi}(v)$. We will prove that v is a cut-vertex. Suppose not. Then by Theorem 6.3, there exists a back edge between a descendant y of w and an ancestor x of v. Since x is an ancestor of v and y is a descendent of w, we have $\text{dfi}(x) < \text{dfi}(v) < \text{dfi}(y)$. From the definition of $\text{low}(w)$ we know that $\text{low}(w) \leq \text{dfi}(x)$. Therefore, $\text{low}(w) \leq \text{dfi}(x) < \text{dfi}(v)$. This is a contradiction with the assumption that $\text{low}(w) \geq \text{dfi}(v)$. Therefore, v is a cut-vertex of G. □

In the next section we combine the theorems we have discussed above with a recursive DFS algorithm. This combination will give us an algorithm that finds cut-vertices and blocks of a given graph.

6.2 Blocks

The next algorithm finds all blocks of a graph. As described above, Theorems 6.2 and 6.5 characterize cut-vertices of a DFS tree. In order to find the blocks of a graph we use a DFS algorithm as a basis and within it incorporate the notion of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

low(v). Since DFS is essentially a recursive procedure, we use the remark below in Algorithm 6.7 to combine evaluation of low(v) with the DFS algorithm.

Remark 6.6 To find the cut-vertices and blocks of a graph using Theorem 6.5 we need to compute low(v) for any vertex v. Consider the definition of low value low(v). Our recursive procedure for evaluating low(v) will be based on the following:

$$\text{low}(v) = \min(\{\text{dfi}(v)\} \cup \{\text{low}(y) \mid y \text{ is a child of } v\} \cup \{\text{dfi}(w) \mid vw \text{ is a back edge}\}).$$

Algorithm 6.7 uses the idea of DFS, combined with the necessary steps to find blocks. During the search, information will be computed and saved so that the edges can be divided into edge sets of different blocks as the search progresses. Algorithm 6.7 checks if a vertex in the DFS tree of a graph G is a cut-vertex each time the search backs up to it. Suppose that the search backs up from w to v, and that v is a non-root vertex such that no descendant of w is joined to an ancestor of v by a back edge. Then, by Theorem 6.3, v must be a cut-vertex. The subtree rooted at w, along with the edge vw, and all back edges leading from w, can be separated from the rest of the graph at v. However, this subgraph of G is not necessarily a block; it can be a union of several blocks. To ensure that the blocks are properly identified, we remove each one as soon as it is detected. Also, vertices that are farther than the root are tested for a cut-vertex property before the vertices closer to the root. This ensures that when a cut-vertex is found, the subtree rooted at that cut-vertex (along with the edges mentioned before) forms one block [5]. Therefore, we need to keep track of how far back in the tree one can get from each vertex by following tree edges and back edges. This is when we use the notion of low. During the algorithm, the vertices are numbered according to the order in which they are first visited, that is, using the depth-first search index, dfi. The depth-first search for blocks (DFSB) algorithm initializes low(v) to its maximum possible value, which is dfi(v). Then during the algorithm, each time a descendant w is found such that low(v) $\geq$ low(w), we update low(v). The algorithm must keep track of the edges traversed during the DFS so that those in one block can be identified. As we discussed, when a block is found, its edges are the edges most recently visited (after the previous cut-vertex). Therefore, the DFSB algorithm uses a stack for keeping the edges such that as soon as a block

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is found, its edges will be popped from the stack and output as a block. In the main body of the algorithm we first initialize the stack and then during the procedure DFSB, the edges are put in the stack.

Algorithm 6.7 [18] *: To find blocks of a graph G by using DFS*

Input: Connected graph G

Output: Edges of each block of G

Finding-blocks

```
procedure DFSB(v)
begin
    i := i + 1
    dfi(v) := i
    low(v) := i
    for all w ∈ N(v) do
    begin
    if dfi(w) < dfi(v) then
        push vw to S
        if dfi(w) = 0 then
        begin
            DFSB(w)
            if low(w) ≥ dfi(v) then
            begin
                add v to the set K
                pop and output the stack S up to and including vw
            end
            else
                low(v) := min(low(v), low(w))
        else
            low(v) := min(low(v), dfi(w))
    end
end
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

% main body of the algorithm %

begin

  $i := 0$

  initialize empty stack S as the edgestack

  initialize set of cut-vertices K as empty

  for all $v \in$ V(G) **do**

    dfi(v) := 0

  choose an arbitrary vertex v

  $DFSB(v)$

end.

We will show that the complexity of the algorithm for finding blocks is $O(\max\{n, m\})$. As we have described before, the complexity of the DFS algorithm is $O(\max\{n, m\})$. If a graph is connected, this would simplify to $O(m)$. The depth-first search for finding blocks is an extension to the DFS algorithm. The only difference is the usage of the stack. But the total time that is needed through the entire calls of the DFSB procedure to push and pop the edges from the stack is proportional to m. Hence this adds $O(m)$ to the complexity of the DFS. Therefore the complexity of the depth-first search for blocks in a connected graph is $O(m)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7

Maximal and Maximum Cliques

Finding complete subgraphs of a graph is one of the fundamental problems of graph theory, and has various applications. The problem of finding a maximum clique in a graph is NP-complete. (For further discussion on NP-completeness of this problem we refer the reader to [18].) However, there are approximation algorithms for the problem of finding a maximum clique in a graph. Such algorithms based on heuristic methods find a reasonably big maximal clique. These type of algorithms are based on the assumption that a large enough maximal clique would be very close to a maximum clique in the graph. Clearly, this assumption is not true for all graphs, because not all maximal cliques in a graph need to be maximum cliques, too. For example, consider Figure 6. The set $C = \{a, b\}$ forms a maximal clique which is not maximum. In this example, the maximum clique contains vertices c, d, and e. In fact, C does not have any vertex in common with the maximum clique of the graph. However, in a lot of cases, finding a maximal clique may lead us to the maximum clique.

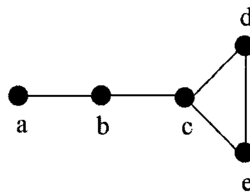

Figure 6

We start this chapter by studying the problem of finding maximal cliques in a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

graph. We discuss two algorithms for this problem. The first algorithm, because of the special way it is developed, will give us a big maximal clique (close to maximum) in many cases. The second algorithm will find all maximal cliques in a graph. However, we should mention that finding all maximal cliques of a graph is not of polynomial complexity, because the maximum number of maximal cliques of a graph can be exponential in the number of vertices of the graph [38].

In the second section of this chapter we study an algorithm for finding a maximum clique of a graph, and once again this algorithm is not a polynomial algorithm since the problem is NP-hard.

7.1 Maximal Cliques

7.1.1 Finding a Maximal Clique Containing a Vertex of Maximum Degree

One approach to approximate a maximum clique is to find a big maximal clique. The authors of [8] suggest an algorithm based on this approach, which we present below (Algorithm 7.1). For this algorithm we make the assumption that a big clique would contain a vertex of maximum degree. It should be mentioned that this is not necessarily the case for all graphs. However, it seems to be a good assumption in many cases. Algorithm 7.1 builds a maximal clique around a vertex of maximum degree. The algorithm initializes a clique C to contain just v_{max}, where v_{max} is a vertex of largest degree in G. The set S initially contains all of the neighbours of v_{max} in G. We start constructing a clique as the following. Each time we look at the subgraph induced by S, find a vertex of largest degree in G[S], and assign it to v_{max}. Since this vertex has been chosen from the neighbourhood of the previous v_{max}, it is adjacent to it. We add it to the clique C and delete it from S. Then we look at S and keep each vertex that is adjacent to v_{max}, and delete others. Therefore, the vertices that remain in S are adjacent to all of the vertices in the current clique C. Again, we find the vertex of largest degree in the subgraph induced by S, and repeat the above process. This continues until S becomes empty; that is, there are no more

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vertices in the graph that are not in the clique and are adjacent to all vertices of the current clique. So, at the termination of the algorithm, the clique can not be extended anymore, and hence, it is maximal.

Algorithm 7.1 : *To find a maximal clique of* G *that contains a vertex of maximum degree*

Input: graph G
Output: a maximal clique of G that contains a vertex of maximum degree
begin
  v_{max} := vertex of maximum degree in G
  $C := \{v_{max}\}$
  $S := N_G(v_{max})$
  let D be the sequence of degrees of vertices in G[S]
  while S is not empty **do**
  begin
    v_{max} := vertex of largest degree in G[S]
    $S := S - \{v_{max}\}$
    update D
    $C := C \cup \{v_{max}\}$
    for all vertices $u \in S$ **do**
      if $u \not\sim_{G[S]} v_{max}$ **then**
      begin
        $S := S - \{u\}$
        update D
      end
  end
  output C
end.

In order to find a vertex of maximum degree, we need to find the degrees of all vertices in a graph; that is, for each vertex v of a graph G we need to check up to $n-1$ vertices to find the neighborhood and therefore the degree of v. This requires at most

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$n(n-1)$ operations. So, finding a vertex of maximum degree takes $O(n^2)$ time. Also, we can perform update of the degree sequence D by deleting the degree of the vertex u that has been removed from S, and subtracting 1 from the degrees of all vertices adjacent to u in G$[S]$. Therefore, the upper-bound of $O(n^2)$ remains unchanged for each time that the "while" loop is performed. Since S is the set of neighbours of a vertex of maximum degree, Algorithm 7.1 has complexity of $O(n \cdot n^2) = O(n^3)$.

7.1.2 Finding All Maximal Cliques

The following algorithms, which deal with finding maximal cliques, are based on [38]. This approach starts with a clique and tries to extend it in all possible ways. This is done by building a set of candidate vertices, that is, vertices that are adjacent to all vertices of the current clique. Then all possible extensions of the clique with vertices of the candidate set are obtained by taking each candidate vertex, removing it from the set of candidates, and adding it to the clique. The set of candidate vertices is then updated and this procedure is performed recursively starting with the new set of candidates and the current clique.

In order to study this approach, some related lemmas are mentioned, and then Algorithm 7.6 is developed based on these lemmas.

Lemma 7.2 [38] Every maximum clique of a graph is maximal.

PROOF. Assume C is a maximum clique of G that is not maximal. Then there is a clique D in G such that $C \subseteq D$ and $C \neq D$. Therefore, clique D has more vertices than C. This contradicts the assumption that C is a maximum clique. Hence, C is maximal. □

Lemma 7.3 [38] Let C be a clique of the graph G and $B \subseteq C$. Then, B is also a clique of G.

PROOF. Since C is a clique, for all distinct vertices $u, v \in C$ we have $u \sim v$. Particularly, for all vertices $u, v \in B$ we have $u \sim v$. Therefore, B is a clique of G. □

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In this section we discuss a backtracking method that finds all maximal cliques of a graph. This method is based on the observation that once we have a clique C of a graph, it can be extended to a larger clique if there is a vertex not in C which is adjacent to all of the vertices of C. So let C be a clique of G and assume that $P \subseteq \mathrm{V(G)} - C$ is the set of candidate vertices that we can use to extend C, so $P = \{v \in \mathrm{V(G)} - C | u \sim v$ for all $u \in C\}$. For each vertex v in P we do the following process: remove v from P and add it to C, create a new set P from the old one by removing all those vertices that are not adjacent to v, and perform this procedure recursively on the new sets C and P.

The above procedure extends a clique C. The only problem is that we might obtain the same clique twice. To avoid this problem we fix an order on the vertices of G and we let the set P of candidate vertices includes only those vertices that are adjacent to all of the vertices of C and are greater than all the vertices in C, according to the order of the vertices; that is, $P = \{v \in \mathrm{V(G)} - C|$ order(v) >order(u) and $u \sim v$ for all $u \in C\}$.

As we discussed above, the clique of a graph can not be extended to a larger clique unless there is a vertex outside of the clique that is adjacent to all of the vertices in the clique.

Lemma 7.4 [38] A clique C of a graph G is maximal if and only if there is no vertex $w \in \mathrm{V(G)} - C$ such that $v \sim w$ for all vertices $v \in C$.

PROOF. Let C be a maximal clique of G and suppose that there exists a vertex $w \in \mathrm{V(G)} - C$ such that $v \sim w$ for all vertices $v \in C$. Now consider $C \cup \{w\}$. Then for all vertices $u, v \in C \cup \{w\}$ we have $u \sim v$. Therefore, $C \cup \{w\}$ is a clique, contradicting the maximality of C.

Conversely, let C be a clique of G with no vertex $w \in \mathrm{V(G)} - C$ such that $v \sim w$ for all vertices $v \in C$. We want to prove that C is maximal. Suppose not. So, there is a set $D \subseteq \mathrm{V(G)} - C$ such that the subgraph induced by the set of vertices $C \cup D$ is a maximal clique. Then, for all vertices $v, w \in C \cup D$, and in particular for all vertices $v \in C$ and $w \in D$, we have $v \sim w$. But this contradicts the assumption that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

there is no vertex $w \in \mathrm{V}(G) - C$ such that $v \sim w$ for all vertices $v \in C$. Therefore, C is maximal. □

By the above theorem we conclude that in an algorithm that finds maximal cliques of a graph, a necessary condition for a clique C to be maximal would be $P = \emptyset$, since if $P \neq \emptyset$, then we can extend the clique C to a larger clique by adding a vertex from P using the same process that we described before. Clearly, such a necessary condition is not sufficient since we might have already obtained a larger clique that contains C but the set P is still empty.

To solve this problem we use two sets of vertices P and S. Let $P \subseteq \mathrm{V(G)} - C$ be the set of candidate vertices that we can use to extend the clique C to a larger clique that has not been obtained before, and let S be the set of vertices such that all extensions of the clique C with vertices from S have already been obtained; that is,

$S = \{u \in \mathrm{V(G)} - C - P : C \cup \{u\}$ is a clique that has been obtained earlier.$\}$

Now we extend a clique C to a larger one by taking a vertex $v \in P$, removing it from P and adding it to C, creating new sets P and S from the old ones by removing those vertices that are not adjacent to v, recursively performing this procedure on the new sets C, P, and S, and then removing v from C and adding it to S. During this procedure, a clique C is maximal if and only if P and S are both empty.

Lemma 7.5 [38] A clique C is maximal if and only if the sets P and S are empty.

PROOF. Let C be a maximal clique and suppose $P \neq \emptyset$. So, there is a vertex $w \in P$ such that $w \sim v$ for all vertices $v \in C$. Then the set $C \cup \{w\}$ is a clique of G that is larger than C. This contradicts the maximality of C. Therefore, the set P should be empty.

Suppose $S \neq \emptyset$; that is, there is a vertex $w \in S$ such that the clique $C \cup \{w\}$ has already been obtained. This again contradicts the maximality of C, and therefore S is empty.

Conversely, suppose that C is a clique of G such that both sets P and S are empty. We want to prove that C is maximal. Suppose not; that is, there is a clique D such that $C \subseteq D$ and $C \neq D$. Let w be a vertex of D that is not in C. Since D is a clique,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

we have $v \sim w$ for all vertices v of D, particularly $v \sim w$ for all vertices v of C, which means that $w \in P$ or $w \in S$, contradicting the assumption that P and S are empty. Hence, C is a maximal clique. □

The following algorithm finds maximal cliques of a graph G by starting from an initially empty clique C and extending it according to the previous lemmas. The extension of a clique C to a larger one is done by the procedure Maximal-Clique, which recursively adds to C a vertex $w \in \mathrm{V(G)} - C$ that is adjacent to all vertices in C. For each extension of the clique C by adding a vertex $v \in P$, a new set PP of candidate vertices will be created from those vertices of P that are adjacent to v. Similarly, a new set SS is created, containing those vertices of S that are adjacent to v.

Algorithm 7.6 :*Finding all maximal cliques of a graph*

Input: Connected graph G
Output: The set L of all maximal cliques of G
Maximal-Clique
Procedure Next-Maximal-Clique(G, C, P, S, L)
begin
  if P and S are both empty **then**
  begin
    add C to L
  end
  else
  begin
    for each vertex v of P **do**
    begin
      delete v from P
      for each vertex $w \in N(v)$ **do**
      begin
        if $w \in P$ **then** add w to the set PP
        if $w \in S$ **then** add w to the set SS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                end
                add v to the set C
                Next-Maximal-Clique(G, C, PP, SS, L)
                delete v from C
                add v to S
            end
        end
    end
    % main body of the algorithm %
    begin
        initialize the sets C and S as empty
        initialize the set of cliques L as empty
        P := V(G)
        Next-Maximal-Clique(G, C, P, S, L)
        Output L
    end.
```

7.2 Maximum Cliques

The problem of determining whether a graph G of n vertices has a clique with at least k vertices is an NP-complete problem (see [38]), whence, all known algorithms for this problem take exponential time in the number of vertices of the graph.

If we already know all maximal cliques of a graph G, then the maximum cliques of G can be found by choosing the maximal cliques of largest size (greatest number of vertices). With the help of this simple observation and the following lemmas, we will have Algorithm 7.9 which finds all maximum cliques of a given graph.

The next theorem gives an upper bound on the size of a maximum clique of a graph.

Lemma 7.7 [38] If C is a maximum clique of the graph G, then $|C| \leq \Delta(\mathrm{G}) + 1$, where $\Delta(\mathrm{G})$ is the maximum degree of G.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PROOF. By contradiction. Let C be a maximum clique of G and suppose that $|C| > \Delta(\mathrm{G}) + 1$. Since C is a clique, for all vertices $u, v \in C$ we have $u \sim v$, which means that $\deg(v) \geq |C| - 1 > \Delta(\mathrm{G}) + 1 - 1 = \Delta(\mathrm{G})$ for all vertices $v \in C$. But this is a contradiction, since $\Delta(\mathrm{G})$ is the maximum degree of G. Hence, $|C| \leq \Delta(\mathrm{G}) + 1$.

□

Lemma 7.8 [38] If C is a maximal clique of the graph G and $D \subseteq C$, then $|C| \leq |D| + |P|$, where P is the set of all candidate vertices with which clique D can be extended.

PROOF. By Lemma 7.3 we know that D is also a clique of G. Since P is the set of all candidate vertices for extending D to a bigger clique, at most $|P|$ vertices can be added to D to obtain a larger clique. Therefore, $|C| \leq |D| + |P|$. □

We use the above lemmas to develop an algorithm for finding maximum cliques of a graph. The algorithm is similar to Algorithm 7.6. Suppose that at some point in the algorithm C_{max} is the largest clique found so far and C is the clique we are trying to extend. Since we want to find maximum cliques, if $|C| = \Delta(\mathrm{G}) + 1$, then by Lemma 7.7 C is a maximum clique. Also, if P is a set of all candidate vertices for C, then if $|C_{max}| > |C| + |P|$, by Lemma 7.8 we do not try to extend C since it will not give us a clique larger than C_{max}. Considering these two additions, the algorithm for finding all maximum cliques is as follows. The algorithm starts with an empty clique C and extends it in all possible ways with a recursive procedure Next-Maximum-Clique. This procedure extends a clique C of G by adding a vertex v from the set P of candidate vertices. Then it creates a new candidate set PP by including only those vertices of P that are adjacent to v. Similarly, the algorithm would create a new set of non-candidate vertices SS from the previous set of non-candidate vertices S, by including only those vertices of S that are adjacent to v. At each stage before making an extension of the current clique, the algorithm checks for the conditions described in Lemmas 7.7 and 7.8. That means that if the size of the current largest clique, m, is less than or equal to the size of the current clique C plus the size of the new candidate set PP, and if C has not yet reached the upper

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

bound of $\Delta(G)+1$, then the algorithm extends C. The algorithm keeps the size of the largest clique found so far in m. The set L contains the maximal cliques found so far whose size is equal to m. Every time a new maximal clique C is found such that $|C| > m$, the algorithm changes m to be equal to $|C|$, deletes from L all cliques of size less than $|C|$, and adds C to L.

Algorithm 7.9 :*Finding all maximum cliques of a graph*

```
Input: Connected graph G and maximum degree Δ(G)
Output: All maximum cliques of G
Maximum-Clique
Procedure Next-Maximum-Clique(G, C, P, S, L, m)
begin
   if P and S are both empty and |C| = m then
      add C to L
   if m < |C| then
   begin
      m := |C|
      delete those cliques from L whose sizes are less than m
   end
   if |C| < Δ + 1 then
   begin
      for each vertex v of P do
      begin
         delete v from P
        for each vertex w ∈ N(v) do
         begin
            if w ∈ P then add w to the set PP
            if w ∈ S then add w to the set SS
         end
         add v to the set C
            if (m ≤ (|C| + |PP|)) then
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                Next-Maximum-Clique(G, C, PP, SS, L, m)
            delete v from C
            add v to S
        end
    end
end
% main body of the algorithm %
begin
    initialize the sets C,S,L as empty
    P := V(G)
    m := 0
    Next-Maximum-Clique(G, C, P, S, L, m)
end.
```

7.3 Comparison

Algorithm 7.1 finds a maximal clique containing a vertex of maximum degree of a given graph. This algorithm is a polynomial time algorithm. Algorithm 7.6 finds all maximal cliques of a given graph. This algorithm is of exponential time. Algorithm 7.9, which is based on Algorithm 7.6, finds all maximum cliques of a graph. Similarly, this algorithm is of exponential time, and therefore, it is very time-consuming and is not a good practical choice. So, even though Algorithm 7.1 does not guarantee that it can find a maximum clique in all cases, it has the advantage of finding a possibly very large maximal clique and possibly close to maximum clique in a polynomial time. Therefore, despite the theoretical value of Algorithms 7.6 and 7.9, in most applications Algorithm 7.1 seems to be a better choice, especially when the graph has a large number of vertices and edges.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 8

Modeling the Database

This chapter will describe the definitions and tools that we use to model the database of gene expression levels. This database was derived from StemBase [31]. StemBase is a database that can be accessed at http://www.stembase.ca. It contains data on Affymetrix DNA microarray experiments. It currently has more than 180 samples which are mostly human and mouse stem cell samples and their derivatives [43]. The data used for this thesis can be accessed under the chip name MOE430.

The notions, method and theorems described in this chapter are mostly original results.

8.1 Deriving a Graph from the Data

We will describe our approach for constructing a graph that models the database of gene expression levels.

The database is derived from laboratory experiments over various cell samples. The expression level of each gene in each sample has been calculated. The expression level of a gene can be viewed as a random variable defined over the set of samples.

Originally, the information on 45,135 genes over the set of samples were taken from StemBase. Since dealing with such a large amount of information was not possible at this point, we took the following approach.

For each gene, in each sample, the average expression level over multiple replicates

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is calculated. The biological replicates recreate the experiment several times. This gives a sense of biological variability [44]. Dr. Scott Findlay and his team calculated the average gene expression level over usually 3, sometimes 2 or 1, replicates. Also, for each gene, its variance in expression level over the set of samples was found.

To make the set of genes more manageable, the 45,135 genes are ranked on the basis of their variance in expression levels over the set of samples. According to this ranking, genes are assigned indices 1,2,...,45135, with gene 1 being the gene that has the largest variance in expression level and gene 45135 being the gene with the least variance in expression level over the set of samples. These were done by Dr. Scott Findlay and his team.

In order to understand the function of a cell and the significance of the expression of a gene in a cell, the genes with higher variance are of particular interest for us. The genes with higher variance are the ones that are expressed differently in various cells, and therefore, they will be more informative. The genes with low variance are expressed at a similar level in different cells, and therefore, their study will not reveal much information [17]. Hence, the top 256 genes according to this ranking were chosen. It should be mentioned that the decision for choosing 256 genes is based on the technological constraints of this research. Later on, in our graph, vertex 1 corresponds to gene 1 of this ranking.

Finally, the correlation matrix for the top 256 genes was calculated. For this purpose, as we mentioned before, gene expression level can be viewed as a random variable and therefore, correlation as defined in Section 2.3 can be calculated between two expression levels. The correlation matrix is a 256×256 matrix that contains the correlation between each pair of gene expression levels. This matrix is a symmetric matrix, since the correlation between expression level of gene i and gene j is equal to the correlation between expression level of gene j and gene i. This matrix was provided to us by Dr. Findlay. We use this matrix to construct a graph G.

Thus, we have a symmetric matrix $A = (a_{ij})$ of size 256×256 such that each entry a_{ij}, for $1 \leq i, j \leq 256$, represents the correlation between the expression level of gene i and gene j. For all $1 \leq i, j \leq 256$, the value of a_{ij} is a real number between -1 and 1 (see Lemma 2.62). We consider the matrix A to be the adjacency matrix of a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

weighted graph of 256 vertices, and the weight of the edge $e = v_i v_j$, $w(e)$, is equal to the entry a_{ij} of the matrix A. We should mention that in matrix A, the values of a_{ii} for $1 \leq i \leq 256$ are equal to 1. That is, the correlation between the expression level of each gene and itself is equal to 1. In the graph G, this would correspond to having a loop on each vertex. Clearly, these loops will not affect parameters such as the number of connected components of the graph. Hence, having the values $a_{ii} = 1$ will not cause any problems in our modeling. For convenience, we shall consider the graph G to be loopless, therefore, the diagonal values of the corresponding adjacency matrix are equal to zero.

8.2 Gene Expression Graph

In this section we will describe our terminology and results that have not been discussed before. First, we study some properties of weighted graphs. Then, we define the notion of a gene expression graph and perfect correlation graph, and we obtain the results that relate these two notions to each other.

8.2.1 Properties of weighted graphs

Our model for the database of gene expression levels is based on a weighted graph. Here, we introduce the new terms on weighted graphs that we will use later to prove our results.

Definition 8.1 In a weighted graph G an edge is called a *negative (positive) edge* if its weight is negative (positive).

Definition 8.2 A complete weighted graph G has a *multiplicative sign property* if for any three vertices x,y, and z in G we have the following:

$$\text{sign}(w(xz)) = \text{sign}(w(xy))\text{sign}(w(yz)).$$

Theorem 8.3 Let G be a complete weighted graph. Then G has a multiplicative sign property if and only if G does not contain any cycles with an odd number of negative edges.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PROOF. ($\Rightarrow$) Assume that G is a weighted complete graph and it does not contain any cycles with an odd number of negative edges. Then G does not contain any cycle of length 3 with an odd number of negative edges. Let $C = xyz$ be a cycle of length 3. Since C does not have an odd number of negative edges, there are two possible cases for the signs of edges xy, yz, and xz. The first possibility is that two edges have negative signs and one edge has a positive sign. In this case, the signs of the edges satisfy a multiplicative sign property. The second case is when all three edges have positive signs; and therefore, a multiplicative sign property is satisfied. Hence, G has a multiplicative sign property.

($\Leftarrow$) By induction on the length of a cycle. Consider a cycle of length 3, say $v_1v_2v_3$ in G. If $w(v_1v_2)$ is negative and $w(v_2v_3)$ is positive, then by the multiplicative sign property, $w(v_1v_3)$ must be negative. Similarly, if $w(v_1v_2)$ is positive and $w(v_2v_3)$ is also positive, then $w(v_1v_3)$ is positive, and if $w(v_1v_2)$ and $w(v_2v_3)$ are both negative, then $w(v_1v_3)$ must be positive, according to the assumption that G is a complete graph with the multiplicative sign property. Therefore, in any case, the cycle of length 3 has an even number of negative edges. This is the basis of our induction.

Now let G be a complete weighted graph with the multiplicative sign property such that no cycle of length less than $k \geq 4$ has an odd number of negative edges (induction hypothesis). Let $C = v_1v_2...v_{k-1}v_kv_1$ be a cycle of length k in G. We will show that C does not have an odd number of negative edges. Take an arbitrary negative edge of C, say v_1v_k.

If $v_{k-1}v_k$ is a negative edge, then by deleting v_k and adding the edge v_1v_{k-1} we have a cycle C_1 of length $k-1$. Since v_1v_k and $v_{k-1}v_k$ have negative weights, by the multiplicative sign property, v_1v_{k-1} must have a positive weight. By the induction hypothesis, C_1 does not have an odd number of negative edges. Therefore, along the path $v_1, ..., v_{k-1}$ we have an even number of negative edges. Hence, the cycle C has two more edges of negative weights, v_1v_k and $v_{k-1}v_k$. Therefore, C does not have an odd number of negative edges.

Now assume that $v_{k-1}v_k$ is a positive edge. Since v_1v_k is negative, by the multiplicative sign property we conclude that v_1v_{k-1} is also a negative edge. Now consider the cycle $C_2 = v_1v_2...v_{k-1}v_1$. By the induction hypothesis, C_2 has an even number of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

negative edges. Therefore, the path $P = v_1v_2...v_{k-1}$ has an odd number of negative edges. Since C has one more negative edge than P, we conclude that C has an even number of negative edges.

By induction, G does not have any cycle with an odd number of negative edges.

$\square$

Definition 8.4 Let G be a weighted graph with weight function $w : \mathrm{E(G)} \to [-1, 1]$, and τ^+ and τ^- be two real numbers in the interval $[0, 1]$. We define the following subgraphs of G induced by specific sets of edges:

$\mathrm{G}(\tau^+) = \mathrm{G}[E(\tau^+)]$, where $E(\tau^+) = \{e \in E(\mathrm{G}) : w(e) \geq \tau^+\}$;

$\mathrm{G}(\tau^-) = \mathrm{G}[E(\tau^-)]$, where $E(\tau^-) = \{e \in E(\mathrm{G}) : w(e) \leq -\tau^-\}$; and

$\mathrm{G}(\tau^+, \tau^-) = \mathrm{G}[E(\tau^+, \tau^-)]$, where $E(\tau^+, \tau^-) = \{e \in E(\mathrm{G}) : w(e) \geq \tau^+$ or $w(e) \leq -\tau^-\}$.

Definition 8.5 Let G be a weighted graph and H a subgraph of G. Then we define the following:

$$\mathcal{O}(\mathrm{H}) = \begin{cases} 0 & \text{if H has an even number of negative edges} \\ 1 & \text{if H has an odd number of negative edges} \end{cases}$$

Lemma 8.6 Let G be a weighted graph with no cycle with an odd number of negative edges, and let $u, v \in \mathrm{V(G)}$. Then for any two $(u, v)-$paths P and Q we have $\mathcal{O}(P) = \mathcal{O}(Q)$.

Proof. By strong induction on $l(P) + l(Q)$. Assume P and Q are two paths with the same ends, and $l(P) = l(Q) = 1$. Then $P = Q$, and therefore the statement of the lemma is true. This serves as the basis of our induction. Now suppose that for some $k \geq 2$ and every pair of ends x, y, we have $\mathcal{O}(P) = \mathcal{O}(Q)$ for all $(x, y)-$paths P and Q such that $2 \leq l(P) + l(Q) \leq k$.

Let x be the first vertex after u that P and Q have in common. There are three cases.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Case 1. If $x = v$, then $P \cup Q$ is a cycle. Since by assumption, no cycle has an odd number of negative edges, we conclude that the number of negative edges in P must have the same parity as the number of negative edges in Q. Therefore, $\mathcal{O}(P) = \mathcal{O}(Q)$.

Case 2. If $x \neq v$ but $x \sim u$, then by the induction hypothesis $\mathcal{O}(P') = \mathcal{O}(Q')$, where P' and Q' are the $(x, v)-$sections of P and Q. Hence, whether the edge ux is positive or not, $\mathcal{O}(P) = \mathcal{O}(Q)$.

Case 3. If $x \neq v$ and $x \not\sim u$, then let P'' and Q'' be the $(u, x)-$sections of P and Q, and let P' and Q' be the $(x, v)-$sections of P and Q. By induction hypothesis, $\mathcal{O}(P') = \mathcal{O}(Q')$. Since $P'' \cup Q''$ is a cycle, by assumption of the lemma, it does not have an odd number of negative edges; therefore, $\mathcal{O}(P'') = \mathcal{O}(Q'')$. Now we have $\mathcal{O}(P') = \mathcal{O}(Q')$ and $\mathcal{O}(P'') = \mathcal{O}(Q'')$. Therefore, $\mathcal{O}(P) = \mathcal{O}(Q)$.

Hence by the principle of induction we have $\mathcal{O}(P) = \mathcal{O}(Q)$. □

Corollary 8.7 Let G be a connected weighted graph that has no cycle with an odd number of negative edges. Then G is a spanning subgraph of a complete weighted graph with a multiplicative sign property.

PROOF. Let u and v be two non-adjacent vertices of G. We construct a weighted graph $G + uv$ as follows. Since G has no cycle with an odd number of negative edges, by Lemma 8.6 we know that all $(u, v)-$paths in G have the same value of $\mathcal{O}$. Let P be any $(u, v)-$path in G. Let $w(uv) = 1$ if $\mathcal{O}(P) = 0$ and $w(uv) = -1$ if $\mathcal{O}(P) = 1$. Consider any cycle C in the graph $\text{G} + uv$. If C does not contain the edge uv, then C does not have an odd number of negative edges. If C contains the edge uv, then $C = Qvu$ for some $(u, v)-$path Q. If $\mathcal{O}(Q) = 1$, then by Lemma 8.6, $\mathcal{P} = 1$ and hence by assigning a negative value to $w(uv)$, we can conclude that Qvu has an even number of negative edges. Similarly, if $\mathcal{O}(Q) = 0$, then by Lemma 8.6, $\mathcal{P} = 0$ and therefore by assigning a positive value to $w(uv)$ we can conclude that Qvu has an even number of negative edges. Hence, C does not have an odd number of negative edges.

Therefore, $\text{G}+uv$ is a connected weighted graph with no cycle with an odd number of negative edges. We continue this process until we have a weighted graph H such

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that all pairs of vertices u and v are adjacent in H. Therefore, G is a spanning subgraph of a complete weighted graph H such that there are no cycles with an odd number of negative edges in H. By Theorem 8.3, H has a multiplicative sign property. Hence, G is a spanning subgraph of a complete weighted graph with a multiplicative sign property. □

8.2.2 Gene expression graph and perfect correlation graph

We define a gene expression graph as a way to model the information in the database. Then, we construct a mathematical structure that is based on the properties of correlation. This structure, called a perfect correlation graph, is used to obtain results which are the bases for the analysis of our database.

Definition 8.8 We call a complete weighted graph G a *gene expression graph* if each vertex v_i represents gene i and the weight of an edge v_iv_j is equal to the correlation between the expression level of gene i and the expression level of gene j, calculated based on laboratory experiences.

Definition 8.9 We call a complete weighted graph G a *correlation graph* if each vertex v_i represents a random variable i, and the weight of an edge v_iv_j is equal to the correlation between random variables i and j.

Corollary 8.10 Let G be a correlation graph. Then G has a multiplicative sign property if all the weights are equal to 1 in absolute value.

PROOF. Let G be a correlation graph, and let x,y, and z be three vertices of G that represent random variables X,Y, and Z, respectively. Hence $w(xy) = \rho_{XY}$, $w(yz) = \rho_{YZ}$ and $w(xz) = \rho_{XZ}$. If $|\rho_{XY}| = |\rho_{YZ}| = |\rho_{XZ}| = 1$, then by Lemma 2.63 we have $\text{sign}(\rho_{XZ}) = \text{sign}(\rho_{XY})\text{sign}(\rho_{YZ})$. That means $\text{sign}(w(xz)) = \text{sign}(w(xy))\text{sign}(w(yz))$. Therefore, G has a multiplicative sign property. □

Definition 8.11 A correlation graph is said to be a *perfect correlation graph* if it has a multiplicative sign property.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Remark 8.12 We should mention that the word "perfect" in the term "perfect correlation graph" should not be confused with perfect graphs. For a perfect correlation graph, the word "perfect" refers to the multiplicative sign property, and therefore to having an absolute value of 1 for all the weights (correlations).

Definition 8.13 Let G be a complete weighted graph, and $\tau^+, \tau^- \in [0, 1]$. The pair (τ^+, τ^-) is called a *suitable pair of thresholds* for G if $G(\tau^+, \tau^-)$ is a spanning subgraph of a perfect correlation graph.

Lemma 8.14 Let G be a gene expression graph, and $\tau^+, \tau^- \in [0, 1]$. Then (τ^+, τ^-) is a suitable pair of thresholds for G if and only if $G(\tau^+, \tau^-)$ has no cycle with an odd number of negative edges.

PROOF. ($\Rightarrow$) Assume (τ^+, τ^-) is a suitable pair of thresholds for G. Then, according to Definition 8.13, we know that $G(\tau^+, \tau^-)$ is a spanning subgraph of a perfect correlation graph, say H. By Theorem 8.3, the perfect correlation graph H does not have any cycle with an odd number of negative edges. Therefore, no subgraph of H has a cycle with an odd number of negative edges. Since $G(\tau^+, \tau^-)$ is a subgraph of H, we conclude that $G(\tau^+, \tau^-)$ has no cycle with an odd number of negative edges.

($\Leftarrow$) Assume $G(\tau^+, \tau^-)$ has no cycle with an odd number of negative edges. Then by Corollary 8.7, $G(\tau^+, \tau^-)$ is a spanning subgraph of a complete graph H with the multiplicative sign property. Since G is a gene expression graph, the weight of an edge is the correlation of the random variables corresponding to its ends. Therefore, H is a correlation graph with the multiplicative sign property, and hence it is a perfect correlation graph. Therefore, $G(\tau^+, \tau^-)$ is a spanning subgraph of a perfect correlation graph. By definition, (τ^+, τ^-) is a suitable pair of thresholds for G. □

8.3 Finding a Suitable Pair of Thresholds

Assume G is a gene expression graph, and let $\tau^+, \tau^- \in [0, 1]$. Consider the graph $G(\tau^+, \tau^-)$. According to the definition, $G(\tau^+, \tau^-)$ does not have an edge e such that $-\tau^- < w(e) < \tau^+$. The goal is to find a suitable pair of thresholds (τ^+, τ^-). Let

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$\tau^+ = \tau_0$ and $\tau^- = \tau_0'$, and let (τ_0, τ_0') be the pair of numbers that we want to test for suitability. If we tried to use Lemma 8.14 directly, we could take the following approach. First, we would find all cycles of $G(\tau_0, \tau_0')$, and then check to see if the number of negative edges in every cycle is even; if so, (τ_0, τ_0') is a suitable pair. This requires finding all cycles of a graph, which is generally an exponential problem, and therefore, this is not a good approach. Theorem 8.16 gives us an equivalent set of conditions that must be satisfied for a pair of thresholds to be suitable. These conditions can be verified in polynomial time.

Definition 8.15 Let G be a graph and H a spanning subgraph of G. Define the *quotient graph* G/H to be a graph such that

1. $V(G/H) = \{$connected components of H$\}$, and

2. if $H_1, H_2 \in V(G/H)$, then $H_1 \sim H_2$ in G/H if and only if there exist vertices $v_1 \in V(H_1)$ and $v_2 \in V(H_2)$ such that $v_1 \sim_G v_2$.

Theorem 8.16 Let G be a gene expression graph and $\tau^+, \tau^- \in [0,1]$. Then, (τ^+, τ^-) is a suitable pair for G if and only if both of the following conditions are satisfied:

(i) There is no edge of $G(\tau^-)$ with both ends in the same connected component of $G(\tau^+)$.

(ii) $G(\tau^+, \tau^-)/G(\tau^+)$ is bipartite.

Proof. ($\Rightarrow$) Assume (τ^+, τ^-) is a suitable pair of thresholds for G.

(i) Suppose there is a negative edge, say uv, in a connected component H of $G(\tau^+)$. Since H is a connected component of $G(\tau^+)$, there is a $(u,v)-$path P in H such that all of the edges of P have positive weights. Now $P \cup uv$ makes a cycle in $G(\tau^+, \tau^-)$ with exactly one negative edge. Therefore, there is a cycle in $G(\tau^+, \tau^-)$ with an odd number of negative edges. This contradicts Corollary 8.14. So, there is no negative edge in connected components of $G(\tau^+)$.

(ii) Suppose that the quotient graph $G(\tau^+, \tau^-)/G(\tau^+)$ is not bipartite. Then by Theorem 2.34, $G(\tau^+, \tau^-)/G(\tau^+)$ has an odd cycle. Let $C = H_1 a_1 H_2 a_2 ... H_k a_k H_1$ be an odd cycle of $G(\tau^+, \tau^-)/G(\tau^+)$, where H_i is a vertex and a_i is an edge of $G(\tau^+, \tau^-)/G(\tau^+)$ for $0 < i \leq k$. Then by definition, for any $a_i = H_i H_{i+1}$ there exist

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

vertices $v_{i'} \in H_i$ and $v_{i+1} \in H_{i+1}$ such that $v_{i'}v_{i+1}$ is an edge of $G(\tau^+, \tau^-)$. Note that the edges $v_{i'}v_{i+1}$ all have negative weights. So the odd cycle $C = H_1a_1H_2a_2...H_ka_kH_1$ of $G(\tau^+, \tau^-)/G(\tau^+)$ gives rise to a cycle $Q = v_1P_1v_{1'}v_2P_2v_{2'}...v_kP_kv_{k'}v_1$ in $G(\tau^+, \tau^-)$, where P_i is a $(v_i, v_{i'})$−path within each connected component H_i (see Figure 7).

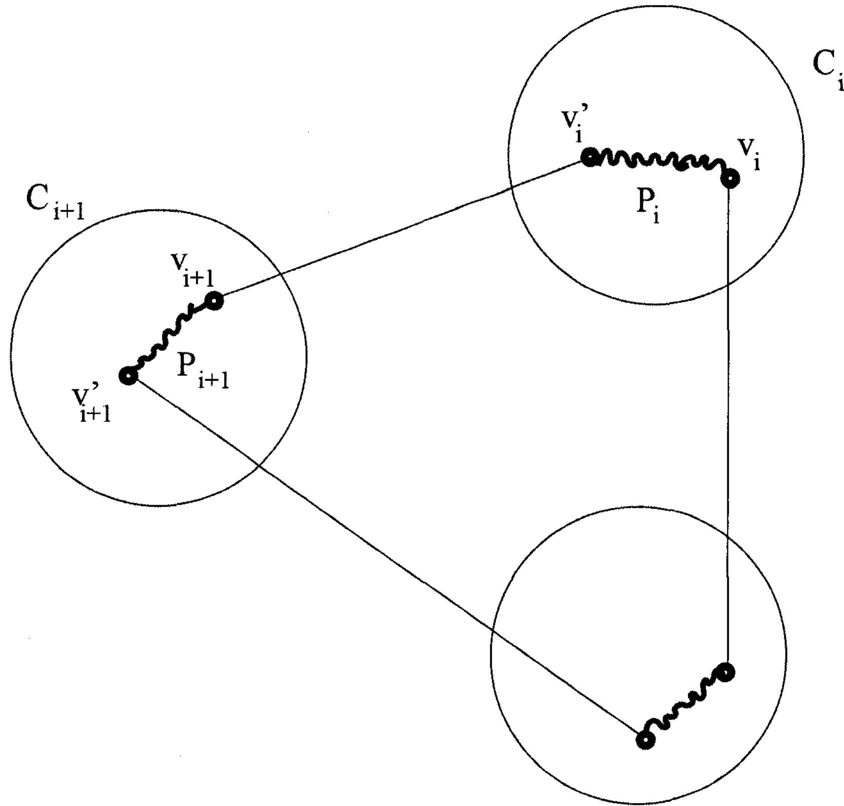

Figure 7

In the cycle Q all the edges of the paths P_i are positive, and all edges $v_{i'}v_{i+1}$ are negative. Since C is an odd cycle, there is an odd number of edges in Q in the form $v_{i'}v_{i+1}$. Therefore, Q has an odd number of negative edges. Hence, $G(\tau^+, \tau^-)$ has a cycle with an odd number of negative edges. But this contradicts the assumption that (τ^+, τ^-) is a suitable pair. Therefore, the quotient graph $G(\tau^+, \tau^-)/G(\tau^+)$ must be bipartite.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

($\Leftarrow$) Assume that the two conditions of the theorem hold. We want to prove that (τ^+, τ^-) is a suitable pair of thresholds for G. Suppose not; therefore, by Corollary 8.14, $G(\tau^+, \tau^-)$ has a cycle C with an odd number of negative edges. Since by assumption the connected components of $G(\tau^+)$ contain no negative edges, the negative edges of C can not be within any connected component of $G(\tau^+)$. Therefore, a negative edge of C is an edge that is between two connected components of $G(\tau^+)$. Hence, the negative edges of C correspond to edges of the quotient graph $G(\tau^+, \tau^-)/G(\tau^+)$. Since C has an odd number of negative edges, the quotient graph $G(\tau^+, \tau^-)/G(\tau^+)$ has a closed walk with an odd number of edges. This contradicts the assumption that $G(\tau^+, \tau^-)/G(\tau^+)$ is bipartite. Hence, $G(\tau^+, \tau^-)$ has no cycle with an odd number of negative edges. By Corollary 8.14, (τ^+, τ^-) is a suitable pair of thresholds for G. $\square$

Remark 8.17 Assume G is a gene expression graph and (τ^+, τ^-) is a suitable pair for G. Then all pairs of (τ_i, τ_j) such that $\tau_i \geq \tau^+$ and $\tau_j \geq \tau^-$ are also suitable.The reason is that if (τ^+, τ^-) is a suitable pair, then by Lemma 8.14, $G(\tau^+, \tau^-)$ has no cycle with an odd number of negative edges. Therefore, any subgraph $G(\tau_i, \tau_j)$ of $G(\tau^+, \tau^-)$ does not have a cycle with an odd number of negative edges, and therefore, (τ_i, τ_j) is also a suitable pair of thresholds for G.

Definition 8.18 For any two pairs of thresholds (a, b) and (c, d), we say that (a, b) is *smaller* than (c, d) if and only if we have $a < c$, or $a = c$ and $b \leq d$. (Note that this is the lexicographic order for pairs of numbers.)

Remark 8.19 Note that for our choice of order for the pairs of thresholds in the interval $[0, 1]$ there exists a smallest pair of thresholds that satisfies the conditions of Theorem 8.16.

Remark 8.20 If the two conditions of Theorem 8.16 are satisfied for $G(\tau^+, \tau^-)$, then they are satisfied for every connected component of $G(\tau^+, \tau^-)$. Conversely, if conditions of Theorem 8.16 are satisfied for every connected component of $G(\tau^+, \tau^-)$, then they are satisfied for $G(\tau^+, \tau^-)$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Method 8.21 We start from $(\tau^+, \tau^-) = (0, 0)$ and increase τ^+ and τ^- until we find the first pair (τ^+, τ^-) that satisfies both conditions. Suppose we wish to check a pair of thresholds (τ^+, τ^-) for suitability. We take the following approach. First, we find the connected components $C_1, C_2, .., C_p$ of $G(\tau^+, \tau^-)$. Then, within each connected component C_i of $G(\tau^+, \tau^-)$, we find the connected components $L_1, L_2, ..., L_q$ of G(τ^+). We check the first condition of Theorem 8.16 for each connected component L_j. This means that for each edge $e = vw$ in L_j, if $w(e)$ is negative, then (τ^+, τ^-) is not a suitable pair of thresholds. If there is no negative edge within a connected component of G(τ_k), then we continue our check for the second condition of Theorem 8.16. That is, we find out whether $G(\tau^+, \tau^-)$/G(τ^+) (restricted to V(L_j)) is bipartite or not. If it is bipartite, then the second condition of Theorem 8.16 is also satisfied. If conditions of Theorem 8.16 are satisfied for all connected components C_i of the graph G(τ^+, τ^-), then we have found a suitable pair of thresholds (τ^+, τ^-) for G. We continue the same process for another pair of thresholds until we find a suitable pair. We initialize (τ^+, τ^-) to $(0, 0)$ and take the method of bisection within the interval $[(0, 0), (1, 1)]$ to find a suitable pair of thresholds that is very close to a smallest suitable pair of thresholds. The step size taken in this process is 0.0001.

This process involves finding connected components of a graph and checking for bipartiteness of a graph. Both of these two problems have been discussed in previous chapters. In Chapter 3 we discussed various polynomial algorithms for finding connected components of a given graph. In Chapter 4 we studied a polynomial algorithm for determining bipartiteness of a graph. Therefore, there exists a polynomial algorithm which can check both conditions of Theorem 8.16 for a given pair of thresholds (τ^+, τ^-) and a given gene expression graph G.

8.4 Other Parameters on the Gene Expression Graph

Besides finding suitable thresholds for the gene expression graph, we considered some other parameters discussed in the previous chapters. After the connected components of $G(\tau^+, \tau^-)$ have been identified, we can focus on each connected component.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The idea is to investigate parameters such as degrees, distance (for graphs without weights), center, radius, diameter, and median. Therefore, after finding connected components of $G(\tau^+, \tau^-)$, we look into each connected component C_i in detail. For each connected component, we first do a study of degrees. We find the degrees of all vertices, maximum degree, minimum degree, and average degree for each connected component. Then, we calculate the radius and diameter of the connected component, and we find the vertices in the center and median of the connected component. Also, for a given vertex v in a connected component C_i, we find distances of all other vertices in C_i from v. Our goal is to obtain as much information as possible about $G(\tau^+, \tau^-)$ and its individual connected components. These parameters and algorithms for computing them have all been studied in Chapters 3-5. We used the algorithms mentioned in Chapters 3-5 for writing a code that computes these parameters.

We should mention the relevance of other parameters studied in the previous chapters. In Chapter 5 we discussed the notion of distance for weighted graphs, and presented algorithms for computing distance between vertices in a weighted graph. In our analysis of distance in the graph $\mathrm{G}(\tau^+, \tau^-)$, we could, alternatively, use this notion of weighted distance; however, the weights used for computing distance would have to correspond to the (absolute values) of the inverses of correlations (or something similar). (The idea is that the larger the correlation in absolute value, the smaller the distance.) Chapter 6 contains results and an algorithm for finding blocks in a graph. A block is in some sense a more strongly connected part of a connected component. Therefore, in our model a block can be viewed as a set of genes that have a possibly stronger relationship with each other. In Chapter 7 we discussed algorithms for finding maximal and maximum cliques. A clique in our model is a set of genes (vertices) in $G(\tau^+, \tau^-)$, any two of which are directly strongly correlated. Computing parameters such as distance in the weighted graph, blocks and cut-vertices, and maximal and maximum cliques may reveal more information about the gene expression graph. In previous chapters we have studied the theory and algorithms for finding these parameters, however, computing them is beyond the scope of this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 9

Implementation of the Algorithms

In this chapter we include the MATLAB code for finding the parameters discussed in Chapter 8. We will also give an example of the output of running the code.

9.1 MATLAB Code

The code written here is in MATLAB version 6.5. One of the reasons for choosing MATLAB was because of the useful and easy structure it provides for dealing with matrices. The code is as follows. The comments are separated by % from the actual code.

```
% This program finds different parameters for the graph G of%
% gene expression levels, based on the matrix of%
% gene expression levels as an input. The program receives%
% two thresholds ( real numbers between 0 and 1) as input%
%
% The program performs the following : %

% 1. Finding the connected components of G(t+,t-)%
% 2. For each connected component C of G(t+,t-)%
%    it checks the two conditions of suitability of thresholds%
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
%    based on Theorem 8.16
%      I) the connected components of G(t+) have no negative edges%
%      II) G(t+,t-)/G(t+) is bipartite %
%    If both of the conditions are satisfied for all connected%
%    components of G(t+,t-), then the pair of thresholds is suitable%
% 2. Receiving the index of one of the connected components C%
%    from the user to perform the analysis %
% 3. Finding degrees, eccentricity, radius, diameter, median, and
%    center for C %
% 5. Finding the distance of vertices of C from a fixed vertex u%
     entered by the user %

% The algorithms, theorems, and methods used in this program are : %

% Algorithm 3.2 :finding connected components by using BFS %
% Algorithm 4.1 :determining whether a graph is bipartite or not %
% Algorithm 5.3 :finding the distances from a given vertex %
%                in graphs without weights %
% Algorithm 5.10 :Floyd-Warshall's Algorithm for finding distance %
%                 between all pairs of vertices, used with weights %
%                 equal to 1 %
% Algorithm 5.12 :finding eccentricity of the vertices %
% Algorithm 5.15 :finding the diameter and radius of a graph %
% Algorithm 5.17 :finding the center of a graph %
% Algorithm 5.22 :finding the median of a graph %
% Theorem 8.16 : determining whether the two conditions for thresholds %
                 to be suitable are satisfied%
% Method 8.21 : finding suitable thresholds %

function thesis=main()
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
a=adjecny_matrix; % this is the matrix of the gene expression levels%

rownumber = length(a(:,1)); % number of row (columns) of a;%
                             % equal to the number of genes%
disp(sprintf('Welcome to the graph-theoretical analysis of
              gene expression graph.'));

% Reading the threshold from the user

threshold_pos=input('Please enter the positive threshold
                     for this test: ');
threshold_neg_1=input('Please enter the negative threshold
                     for this test: ');
threshold_neg= - threshold_neg_1;

disp([' Finding  the connected components of G(t+,t-) and
determining if the pair of thresholds is suitable '])

% Constructing G(t+,t-) : the subgraph with edges whose%
% weights are bigger than t+ or smaller than t-%

gt_plus_minus = zeros(rownumber);
% a square zero matrix of size rownumber which acts as%
% the adjacency matrix of G(t+,t-)%

for i=1:rownumber
    for j=1:rownumber
        if (a(i,j)>threshold_pos)
            gt_plus_minus(i,j) = a(i,j);
        end
        if (a(i,j)==threshold_pos)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            gt_plus_minus(i,j) = a(i,j);
        end
        if (a(i,j)==threshold_neg)
            gt_plus_minus(i,j) = a(i,j);
        end
        if (a(i,j)<threshold_neg)
            gt_plus_minus(i,j) = a(i,j);
        end
    end
end

% finding the connected components of G(t+,t-)%

check_matrix_gen = ones(1,rownumber);
% vector with size rownumber and all entries equal to 1.%
% It will be used to check if a vertex has been visited or not.%
% After a vertex i gets visited, check_matrix_gen(i)=0.%

components_gen =zeros(rownumber);
% each row i of this matrix will be filled with%
% the vertices found in component i.%

que_gen = zeros(1,rownumber);
% a vector of size rownumber (maximum size of the queue %
% during BFS), which will act as a queue %

connecteds_gen =0;
% number of connected components in G(t+,t-)%

for k=1:rownumber   % for all vertices k in the graph%
    if(check_matrix_gen(k)~=0)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
% if k has not been visited before%
% then it becomes the root of a new component%
    connecteds_gen = connecteds_gen +1;
    % a new component is found%
    index_comp_gen =1;
    % shows where to enter a new vertex i%
    % in the component matrix, when i is found%
    quecount_gen = 1;
    % keeps track of the length of the matrix%
    que_gen(1)= k;
    % adding to the queue%
    components_gen(connecteds_gen,1)=k;
    % adding vertex k to component connecteds_gen%
    check_matrix_gen(k)=0;
    % mark the vertex k as visited%

    % performing BFS%
    while (quecount_gen>0) %while queue is not empty%
        index_gen=que_gen(quecount_gen);
        % dequeue a vertex and put it in index%
        que_gen(quecount_gen)=0;
        % delete that vertex from the queue%
        quecount_gen = quecount_gen -1;
        % adjusting the length of the queue%
        for i=1:rownumber
            % finding neighbors of index%
            if(gt_plus_minus(index_gen,i)~=0)
            % if index and i are adjacent in G(t+,t-)%
                if(check_matrix_gen(i)~=0)
                    % if i has not been visited before%
                    check_matrix_gen(i)=0;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    % mark i as visited%
                    index_comp_gen = index_comp_gen+1;
                    components_gen(connecteds_gen,index_comp_gen)=i;
                    % adding i to the list of vertices%
                    % in the component connecteds_gen%
                    quecount_gen = quecount_gen + 1;
                    % adjusting the length of the queue%
                    que_gen(quecount_gen)=i;
                    % adding i to the queue%
                end
            end
        end
    end
end
end

size_comp=zeros(1,connecteds_gen);
% a vector such that size_comp(i) gives the number of vertices%
% in the connected component i%
i=1;
j=1;
while components_gen(i,j)~=0
    while components_gen(i,j)~=0
        size_comp(i)=size_comp(i)+1;
        j=j+1;
    end
    j=1;
    i=i+1;
end
disp(sprintf('The thresholds used for this test are : %g , %g'
              ,threshold_pos,threshold_neg_1));
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
disp(sprintf('The number of connected components of G(t+,t-) is: %g'
              ,connecteds_gen));
disp(sprintf('The sizes of the connected components are: '));
disp(size_comp);

see=input('Do you want to see the vertices in each connected
           component? If yes, please enter 1, otherwise
           please enter 0 : ');
if (see==1)
% printing vertices in each connected component%
    for i=1:connecteds_gen
        j=1;
        disp(sprintf('Component %g of G(t+,t-) includes:',i));
        temp = [0];
        while(components_gen(i,j)~=0)
            temp(j) = components_gen(i,j);
            j = j+1;
            if(j == rownumber+1)
                break
            end
        end
        disp(sprintf(' %g',temp));
    end
end %if see==1

%Check for suitability of the thresholds within each connected component%

suitability=1;
%If the thresholds are not suitable then suitability is zero%

for C=1:connecteds_gen
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
% each time s is the connected component C we are currently%
% studying for suitability of thresholds%
if (suitability==1)
    %while the thresholds are suitable so far, we check for the%
    %suitability of the thresholds in the next component%
    size=size_comp(C);
    %size of the connected component C%
    if (size~=1)
        % if size of the connected component is bigger than 1,%
        % then we check for the conditions; if the size is one,%
        % the conditions are automatically satisfied%

        % Constructing the adjacency matrix of the subgraph C%
        h= zeros(size); % the adjacency matrix of subgraph C%
        for j=1:size
            for k=1:size
              h(j,k)=
                 gt_plus_minus(components_gen(C,j),components_gen(C,k));
            end
        end
        disp(sprintf('Connected components of G(t+)
                      within component %g',C));

        %Constructing G(t+) in C%
        gt_plus_in_C=zeros(size);
        for i=1:size
            for j=1:size
                if (h(i,j)>threshold_pos)
                    gt_plus_in_C(i,j) = h(i,j);
                end
                if (h(i,j)==threshold_pos)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            gt_plus_in_C(i,j) = h(i,j);
        end
    end
end

%Constructing G(t-) in C%
gt_minus_in_C = zeros(size);
for i=1:size
    for j=1:size
        if (h(i,j) < threshold_neg)
            gt_minus_in_C(i,j) = h(i,j);
        end
        if (h(i,j)==threshold_neg)
            gt_plus_in_C(i,j) = h(i,j);
        end
    end
end

array_orig=zeros(1,size);
% this vector gives the label of each vertex in the original%
% graph, for each vertex in the connected component%

for i=1:size
    array_orig(i)=components_gen(C,i);
end

% finding the connected components of G(t_plus)%
% within C whose sizes are larger than 1%

check_matrix = ones(1,size);
components = zeros(size);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
que = zeros(1,size);
connecteds =0;
for k=1:size
    if(check_matrix(k)~=0)
        connecteds = connecteds +1;
        index_comp =1;
        quecount = 1;
        que(1)= k;
        components(connecteds,1)=k;
        check_matrix(k)=0;
        while (quecount>0)
            index=que(quecount);
            que(quecount)=0;
            quecount = quecount -1;
            for i=1:size
                if(gt_plus_in_C(index,i)~=0)
                    if(check_matrix(i)~=0)
                        check_matrix(i)=0;
                        index_comp = index_comp+1;
                        components(connecteds,index_comp)=i;
                        quecount = quecount + 1;
                        que(quecount)=i;
                    end
                end
            end
        end
    end
end
disp(sprintf('The number of connected components of G(t+)
              in connected component %g of G(t+,t-) is:
              %g',C,connecteds));
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
for i=1:connecteds
    j=1;
    disp(sprintf('Component %g of G(t+) includes:',i));
    temp = [0];
    while(components(i,j)~=0)
        temp(j) = components(i,j);
        j = j+1;
        if(j == size+1)
            break
        end
    end
    disp(sprintf(' %g',array_orig(temp)));
end

% finding the sizes of connected components of G(t+)%
sub_size=zeros(1,connecteds);
for i=1:size
    for j=1:size
        if (components(i,j)~=0)
            sub_size(i)=sub_size(i)+1;
        end
    end
end
disp(sprintf('The sizes of the connected components
               of G(t+) are '));
disp(sub_size);

%Checking for the first condition of Theorem 8.16%
for i=1:connecteds
    j=2;
    while(components(i,j)~=0 && j<size+1)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            for k=1:j-1
                if(suitability ==0)
                % if the condition is not satisfied%
                    break   % stop%
                end
                temp1 = components(i,k);
                temp2 = components(i,j);
                if(gt_minus_in_C(temp1,temp2)~=0)
                % if a connected component of G(t+)%
                % contains a negative edge, that is,%
                % an edge of G(t-)%

                    suitability = 0;
                    % first condition is not satisfied%
                end
            end
            j = j +1;
        end
    end
    if (suitability==1)
        %if the first condition is satisfied,
        %check for the second condition

        %constructing the quotient graph restricted to C%
        % the graph whose vertices are the connected%
        % components of G(t+) in C%

        g = gt_plus_in_C + gt_minus_in_C;

       quotient_graphG = zeros(connecteds);
       % adjacency matrix of the quotient graph%
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
% filling the adjacency matrix of the quotient graph %
% with the appropriate entries. Two vertices i and j %
% in the quotient graph G(t+,t-)/G(t+) are adjacent iff%
% their corresponding connected components i and j of %
% G(t+) have two vertices v_i in i and v_j in j such %
% that v_i is adjacent to v_j in G(t+,t-)%

 for index1=1:connecteds
        for index2=index1+1:connecteds
            j1=1;
            while(components(index1,j1)~=0)
                temp1 = components(index1,j1);
                % a vertex in connected component%
                % index 1 of G(t+)%
                j2=1;
                while(components(index2,j2)~=0)
                    temp2 = components(index2,j2);
                    % a vertex in connected %
                    % component index 2%

                    if (g(temp1,temp2)~=0)
                    % if two vertices temp1 and %
                    % temp2 are adjacent in G(t+,t-)%
                        quotient_graphG(index1,index2)=1;
                        % there is an edge in the quotient%
                        % graph between vertices index1 %
                        % and index 2%

                        quotient_graphG(index2,index1)=1;
                        % the adjacency matrix of the %
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                    % quotient graph is symmetric %
                  end
                  j2 = j2+1;
                  if(j2 == size+1)
                    break
                  end
                  j1 = j1+1;
                  if(j1 == size+1)
                    break
                  end
              end
          end
      end
end

%checking the second condition of Theorem 8.16 : %
% if the quotient graph is bipartite%

component_length = length(quotient_graphG(1,:));
% number of vertices in the quotient graph, equal to%
% the number of connected components of G(t+)%

check_color = zeros(1,component_length);
% a zero vector of length equal to number of vertices%
% of the quotient graph. each entry i becomes 1 or 2,%
% depending on the color it receives.%

que = zeros(1,component_length);
% a zero vector of length the number of vertices;%
% queue for the BFS%
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
for i=1:component_length
% i is the vertex of the quotient graph that is%
% currently being visited%
    if(suitability ==0)
    % if the first condition is not satisfied,%
    % we do not need to check the second condition.%

        break
    end

    if (check_color(i)==0)
    % if vertex i has not been visited before%
        check_color(i)=1;
        % color i with color 1%
        quecount = 1;
        % current length of the queue%
        que(1)= i;
        % add i to the queue%
    end
    while (quecount>0)
    % while the queue is not empty%
        if(suitability ==0)
        % if there has been a discrepancy in %
        % the bipartite coloring%
            break
        end
        index=que(quecount);
        % dequeue a vertex and put it in index%

        que(quecount)=0;
        % deleting index from the queue%
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
quecount = quecount - 1;
% adjusting the length of the queue%

for k=1:component_length
    if(quotient_graphG(index,k)~=0)
    % if index and k are adjacent in%
    % the quotient graph%
        if(check_color(k)==check_color(index))
        % if two adjacent vertices k and index%
        % are of the same color, then the %
        % graph is not bipartite%
            quecount = 0;
            % empty the whole queue%
            que = zeros(1,component_length);
            suitability = 0;
            % the second condition is not %
            % satisfied%
            break
        end
        if(check_color(k)==0)
        % if k has not been visited before%
            quecount = quecount + 1;
            que(quecount)=k;
            % add k to the queue%

            % color k with the appropriate color%
            % other than the color of index%
            if(check_color(index)==1)
                check_color(k)=2;
            else
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
                                                check_color(k)=1;
                                            end
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
end
disp(['  ']);

if (suitability==1)
% if both conditions are satisfied for %
% all connected components of G(t+,t-)%
    disp(sprintf('According to the first condition,
                  for a suitable pair of thresholds
                  there should not be any edge of G(t-)
                  whose both ends are in the same
                  connected component of G(t+). '));
    disp(['The second condition checks if the quotient
           graph G(t)/G(t+)
           is bipartite or not']);
    disp(sprintf('Both conditions are satisfied
                  in each connected component
                  of G(t+,t-).'));
    disp(sprintf('A suitable pair of thresholds is found.'));
end

if (suitability~=1)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    disp(sprintf('According to the first condition,
                  for a suitable pair of thresholds
                  there should not be any edge of G(t-)
                  whose both ends are in the same
                  connected component of G(t+). '));
    disp(['The second condition checks if the quotient
           graph G(t)/G(t+)
           is bipartite or not']);
    disp(sprintf('The conditions for suitability are
                  not satisfied in all connected
                  components of G(t+,t-).'));
    disp(sprintf('The pair of thresholds is not suitable.'));
end
disp(['  ']);

again= input('If you want to continue to analysis of a
              specific connected component,
              please enter 1, otherwise please enter 0 : ');

if (again==1)
    disp(sprintf('We now analyze a chosen connected
                  component of G(t+,t-).'));
    M=input('Please enter the connected component for
             which you want to perform the analysis ');

    % Analysis of degrees, radius, diameter, center and median for M%

    size=size_comp(M);  % size of the connected component M

    % Constructing the adjacency matrix of the subgraph M
    h= zeros(size); % the adjacency matrix of subgraph M
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
for j=1:size
    for k=1:size
        h(j,k)=gt_plus_minus(components_gen(M,j),components_gen(M,k));
    end
end

array_orig=zeros(1,size);
% this vector gives the label of each vertex of M%
% in the original graph%
for i=1:size
    array_orig(i)=components_gen(M,i);
end

% constructing the adjacency matrix of graph X.%
% X is a graph such that V(X)=V(C) and the edges of X%
% are the same as the edges of C, without weights.%

X = zeros(size);
for i=1:size
    for j=1:size
        if (h(i,j)~=0)
            X(i,j) = 1;
        end
    end
end
for i=1:size
    for j=1:size
        if (i==j)
            X(i,j)=0;
        end
    end
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
end
disp(['  ']);
disp(sprintf('Study of degrees, radius, diameter,
              center, and median'));

degrees= zeros(1,size);
% a vector that gives deg(i) for each vertex i%
for i=1:size
    for j=1:size
        degrees(i)= X(i,j)+degrees(i);
    end
end

% displaying the degrees of vertices in the %
% connected component M of G(t+)%
see=input('if you want to see the degrees of vertices
           in this connected component,
           please enter 1, otherwise enter 0 : ');
if (see==1)
    disp(sprintf(' The degrees of vertices in component %g are',M));
    for i=1:size
        disp(sprintf(' Degree of vertex %g is %g',
                     array_orig(i),degrees(i)));
    end
end

% finding maximum, minimum, and average degree in component M%

max_deg=0;
min_deg=256;
total_deg=0;
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
for i=1:size
    if degrees(i)>max_deg
        max_deg=degrees(i);
    end
    if degrees(i)<min_deg
        min_deg=degrees(i);
    end
    total_deg=total_deg+degrees(i);
end
average_deg=total_deg / size;
disp(sprintf('The minimum degree in component  %g is %g',M,min_deg));
disp(sprintf('The maximum degree in component  %g is %g',M,max_deg));
disp(sprintf('The average degree in component  %g is %g',
            M,average_deg));
disp(sprintf('  '));

% finding the distance between vertices of X (connected component C)

for i=1:size
    for j=1:size
        distance(i,j)=300;
    end
end
for i=1:size
    for j=1:size
        if (X(i,j)==1)
            distance(i,j)=1;
        end
    end
end
for i=1:size
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    distance(i,i)=0;
end

% Floyd-Warshall algorithm for finding distance%
% between all vertices of a graph :%

for k=1:size
    for i=1:size
        for j=1:size
            if ((distance(i,k)+distance(k,j))<distance(i,j))
                distance(i,j)=distance(i,k)+distance(k,j);
            end
        end
    end
end

% finding eccentricity of vertices of X%

eccentricity=zeros(1,size);
for i=1:size
    for j=1:size
        if (distance(i,j)> eccentricity(i))
            eccentricity(i)=distance(i,j);
        end
    end
end

% finding diameter and radius of X%
radius=300;
diameter=0;
for i=1:size
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    if (eccentricity(i)<radius)
        radius=eccentricity(i);
    end
    if (eccentricity(i)>diameter)
        diameter=eccentricity(i);
    end
end
disp(sprintf('The diameter of connected component %g is : %g'
              ,M,diameter));
disp(sprintf('The radius of connected component %g is : %g'
              ,M,radius));

% finding the center of the graph%
center=zeros(1,size);
j=1;
for i=1:size
    if (eccentricity(i)==radius)
        center(j)=i;
        j=j+1;
    end
end
disp([' The vertices in the center are : ']);
for i=1:j-1
    disp(sprintf('%g',array_orig(center(i))));
end

% finding the median of the graph%
total_distance=zeros(1,size);
for i=1:size
    for j=1:size
        total_distance(i)=total_distance(i)+ distance(i,j);
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
    end
end
min_distance=30000;
for i=1:size
    if total_distance(i) < min_distance
        min_distance=total_distance(i);
    end
end
median=zeros(1,size);
flag2=0;
for i=1:size
    if (total_distance(i)==min_distance)
        median(i)=array_orig(i);
        flag2=1;
    end
end
if (flag2==1)
    disp([' The vertices in the median are : ']);
    for i=1:size
        if (median(i)~=0)
            disp(median(i));
        end
    end
end

% finding distances of vertices in L from a fixed vertex%

choice= input('If you want to find the distance of vertices in
               this connected component from a fixed vertex,
               please enter 1, otherwise enter 0 : ');
if (choice==1)
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
v=input(' Please enter a vertex in this component from
          which to find the distances : ');
% first we check if v is in the connected component M%
i=1;
flag=0;
while flag==0 && i<size+1
    if array_orig(i)==v
        u=i;
        flag=1;
    end
    i=i+1;
end
if (flag==0)
% v is not in this connected component%
    disp(sprintf('You have entered an incorrect vertex.
                  Vertex %g is not in connected component %g'
                  ,v,M));
end

if (flag==1)
% if the given vertex v is in component C%
% find the distance of vertices in C from v by using BFS%
    distance_matrix = zeros(1,size);
    % distance_matrix(i) for each vertex i of C gives%
    % the distance of i from v%
    % initializing distance_matrix to a maximum%
    for i=1:size
        distance_matrix(i)=300;
    end

    % performing BFS to find distance%
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
que = zeros(1,size);
distance_matrix(u)=0;
quecount = 1;
que(quecount)=u;
while (quecount>0)
    x=que(quecount);
    quecount = quecount -1;
    for i=1:size
        if(X(x,i)~=0)
            if(distance_matrix(i)==300)
            % if i has not been visited before%
                distance_matrix(i)=distance_matrix(x)+1;
                % distance between v and i is found%
                quecount=quecount+1;
                que(quecount)=i;
                % add i to the queue%
            end
        end
    end
end
see=input('If you want to see the list of distances of
           vertices, please enter 1, otherwise enter 0 :');
if (see==1)
    disp(sprintf('The distance of each vertex in the connected
                  component %g from vertex %g ',C,v));
    for i=1:size
        disp(sprintf('distance of vertex %g, from vertex %g is'
                      ,array_orig(i),v));
        disp(sprintf(' %g',distance_matrix(i)));
    end
end
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
            % diagram for the distribution of distance%
            % from a fixed vertex%

            array_dist=zeros(1,size);
            % array_dist(k) gives the number of%
            % vertices of distance k from v%

            for i=1:size
                if (distance_matrix(i)~=0)
                    array_dist(distance_matrix(i))=
                        array_dist(distance_matrix(i))+1;
                end
            end
            figure
            plot(array_dist,'r');
            xlabel('distance');
            ylabel('number of vertices at distance x
                    from the chosen vertex');
        end
    end
end
disp([' Thank you for your attention! Bye!!'])
```

9.2 Sample Output

We will present the output of running the above code for thresholds $(0.7986, 0.8029)$. We should mention that in this example we chose not to see the vertices in each connected component of $G(\tau^+, \tau^-)$, and also, we chose not to see the list of degrees, and distances from a fixed vertex. If we would like to see the results of these three

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

options, the user should enter 1 when asked to, during running the code. In the case of choosing to find distances from a fixed vertex, the output would include a diagram showing the distribution of distances from the given fixed vertex. The sample output is as follows.

```
Welcome to the graph-theoretical analysis of gene expression levels!!
Please enter the positive threshold for this test: 0.7986
Please enter the negative threshold for this test: 0.8029

Finding the connected components of G(t+,t-),
and determining if the pair of thresholds is suitable

The thresholds used for this test are : 0.7986 , 0.8029
The number of connected components of G(t+,t-) is: 18
The sizes of the connected components are:
  238  1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

Do you want to see the vertices in each connected component?
 If yes,please enter 1, otherwise please enter 0 : 0

 Connected components of G(t+,t-)

The number of connected components of G(t+) in connected component 1
of G(t+,t-) is: 6

Component 1 of G(t+) includes:
 1 2 3 4 6 7 11 15 23 25 33 39 40 41 43 45 55 56 69 72 75 76 78 82 102
 113 114 115 127 131 144 145 151 155 158 168 173 174 178 189 196 202
 212 224 230 243 246 251 8 18 24 46 48 52 60 64 88 92 93 94 95 100 104
 117 134 139 159 163 166 176 214 220 227 252 32 89 148 170 188 195 238
 49 16 53 197 73 242 29 240 62 107 116 204 34 35 70 132 161 97 14 17
 19 30 44 50 147 153 164 185 237 200 10 85 118 205
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Component 2 of G(t+) includes:

58 152 28 31 63 79 98 136 167 181 199 203 236 182 137 211 12 37 160
254 9 13 21 36 38 77 96 106 112 146 162 177 192 219 234 138 142 218
119 226 244 122 232 141 84 128 59 110 103 241 233 229 101 215 143
206 123 248 198 165 74 213 222 71 111 231 129 169 5 65 201 210 225
99 156 22 154 191 249 42 86 209 253 26 27 51 54 67 87 91 105 109 124
171 172 183 193 239 245 247 250 149 187 256 186 228 126 20 66 217
130 157 207 47 190 223 120

Component 3 of G(t+) includes:

184 194

Component 4 of G(t+) includes:

179 83

Component 5 of G(t+) includes:

216

Component 6 of G(t+) includes:

90

The sizes of the connected components of G(t+) are

115 117 2 2 1 1

Connected components of G(t+) within component 3

The number of connected components of G(t+) in connected component 3 of G(t+,t-) is: 1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Component 1 of G(t+) includes:
 61 68

The sizes of the connected components of G(t+) are

     2

According to the first condition, for a suitable pair of thresholds
there should not be any edge of G(t-) between any two vertices of
the same component of G(t+).

The second condition checks if the quotient graph G(t)/G(t+) is
bipartite or not.

Both conditions are satisfied in all connected components of
G(t+,t-).

A suitable pair of thresholds is found.

If you want to continue the analysis of a specific connected
component, please enter 1, otherwise please enter 0 : 1

We now analyze a chosen connected component of G(t+,t-).

Please enter the connected component for which you want to perform
the analysis: 1

Study of degrees, radius, diameter, center, and median

If you want to see the degrees of vertices in this connected
component, enter 1, otherwise enter 0 : 0
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
The minimum degree in component  1 is 1

The maximum degree in component  1 is 98

The average degree in component 1 is 32.5042

The diameter of connected component 1 is : 13

The radius of connected component 1 is : 7

The vertices in the center are :

18 203 211 12 37 160 254

The vertices in the median are :

3

If you want to find the distance of vertices in this connected
component from a fixed vertex, please enter 1, otherwise enter 0 : 0

Thank you for your attention! Bye!!
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 10

Analyzing the Results

The goal of this thesis was to model the data of gene expression levels as a graph, and to analyze it using various graph-theoretical algorithms. In this chapter we will present the original results that we were able to find by using our method.

As mentioned in Chapter 8, we used a gene expression graph as a way to represent the data of gene expression levels obtained from laboratory experiments. If we consider gene expression level to be a random variable, then the equivalent mathematical structure that can represent a gene expression graph would be a correlation graph. We studied some of the properties of weighted graphs and obtained results related to the multiplicative sign property for weighted graphs. We defined a perfect correlation graph and used the results obtained for weighted graphs on perfect correlation graphs and gene expression graphs. The crucial link in this process is the notion of a suitable pair of thresholds. We defined a suitable pair (τ^+, τ^-) of thresholds for a gene expression graph G to be a pair (τ^+, τ^-) such that $G(\tau^+, \tau^-)$ is a subgraph of a perfect correlation graph. With this definition, we were able to find necessary and sufficient conditions for a pair of thresholds to be suitable for a gene expression graph (see Theorem 8.16).

Chapter 9 gives the code used to analyze the data. As a result, we were able to determine suitable thresholds for our gene expression graph. The smallest pair of suitable thresholds we could find is (0.7986, 0.8029). For any τ^+ larger than or equal to 0.7986 and any τ^- larger than or equal to 0.8029, $G(\tau^+, \tau^-)$ satisfies both

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conditions of Theorem 8.16, and hence, it is a subgraph of a perfect correlation graph.

This result can be interpreted as follows. The connected components of the graph $G(\tau^+, \tau^-)$ represent those clusters of genes that are highly correlated with respect to their expression levels. Therefore, the number of the connected components and their sizes correspond to the number and sizes of the clusters of genes.

According to [17], if the correlation between gene i and gene j is very large (and positive), there are two possible explanations:

1. gene i and gene j are transcribed by the same set of transcription factors,

or

2. an increase in expression level of gene i causes an increase in expression level of gene j, which indicates the existence of a causal relationship between gene i and gene j.

If the high correlation between expression levels of gene i and gene j is due to the fact that they are transcribed by the same set of transcription factors, then the number of clusters can be viewed as an indicator of the number of classes of transcription factors. We can conjecture that the genes in each cluster (connected component) are controlled by the same set of transcription factors.

After running the code for the suitable pair $(0.7986, 0.8029)$ on the data, we notice that there are 18 connected components in the graph $G(0.7986, 0.8029)$. Among these 18 connected components, there is one very large connected component C of size 238. All of the other vertices (genes) that are not present in this main cluster C, form connected components of size 1 and 2, and therefore, are not closely related to any of the genes in the main cluster C.

We should mention that although the absolute values for the suitable pair of thresholds that we found, $(0.7986, 0.8029)$, may seem very large, this is not unexpected. The reason is that the majority of the weights in the gene expression graph have a large absolute value. That is, the majority of the absolute values of the correlations of the gene expression levels are large. Figure 8 shows the distribution of the weights (correlations) for the edges in the data. In this plot, the $x-$values represent the edge weights (rounded to one decimal), and the $y-$values represent the number of edges in the original graph with the given weight. It can be seen that a large

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

number of edges have the absolute value of its weight larger than 0.7, and therefore, the values of the suitable pair of thresholds we found is not unexpected.

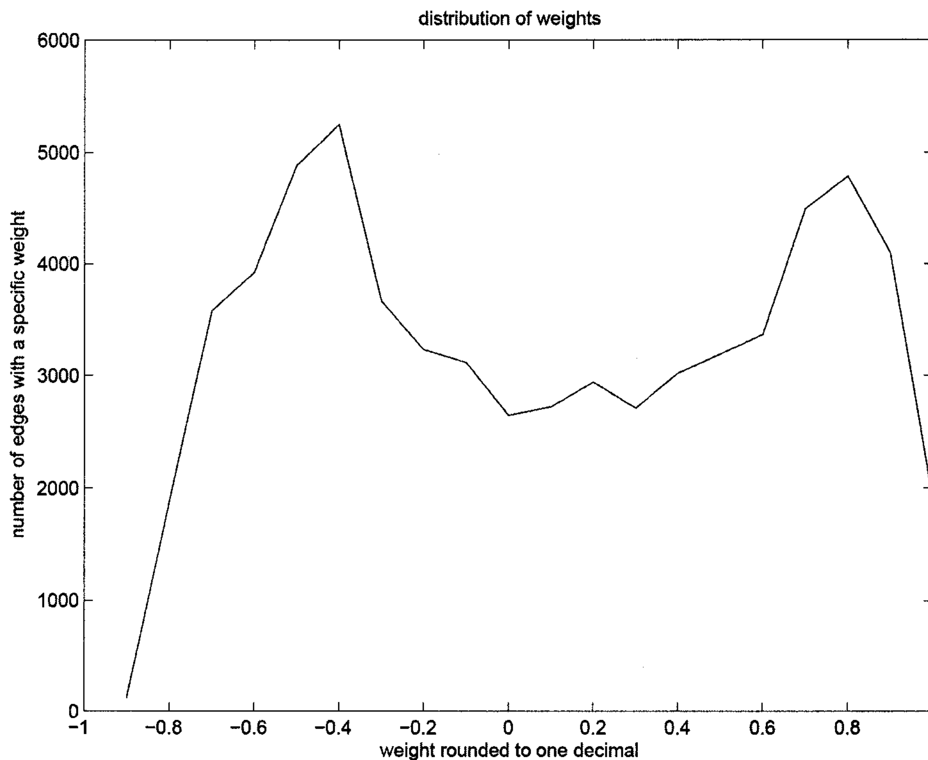

Figure 8

Let us focus on the main cluster C, to obtain more information about the graph $G(\tau^+, \tau^-)$. The main cluster consists of 6 connected components of the graph $G(\tau^+)$; C_1 and C_2 of sizes 115 and 117, respectively, C_3 and C_4 of size 2, and C_5 and C_6 of size 1. Thus, the picture that emerges after finding a suitable pair of thresholds is as follows: The main cluster C partitions into 2 large subclusters and 4 tiny subclusters. The genes in any of the 2 large subclusters have a very high positive correlation to each other, and all the genes in one subcluster have a high negative correlation to the genes in the other subclusters. This could mean, for example, that if the genes in cluster C_1 are highly expressed in a cell, then likely the genes in cluster C_2 will be at a low expression level. Figure 9 shows a snapshot of the structure of the graph

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$G(\tau^+, \tau^-)$.

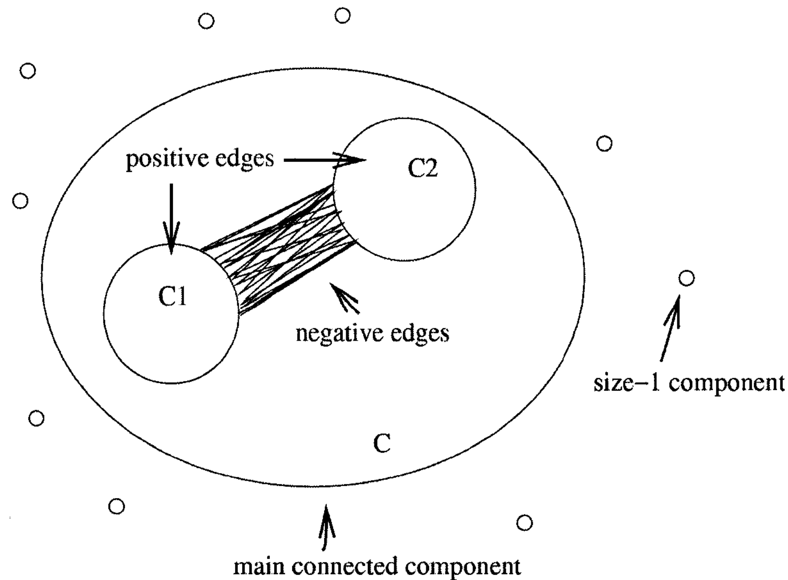

Figure 9

The analysis of other parameters on the graph reveals specific information about the clusters such as vertex degrees, radius, diameter, center, and median of each cluster. One of the parameters that we determined is the degree of each vertex in the cluster. The degree of a gene in the graph $G(\tau^+, \tau^-)$ is the number of the genes directly related to it with correlation greater than or equal to τ^+, or less than or equal to $-\tau^-$. Also, we obtained information about the density of each cluster. This was done by finding the average degree. The larger the degrees in a connected component, the more edges are present in it. Therefore, the average degree of each cluster gives us a measure of the density of each cluster. The average degree for the main cluster C the emerges at the suitable pair of thresholds $(0.7986, 0.8029)$ was equal to 32.5042, with the maximum degree of 98 and minimum degree of 1. The radius of C is 7 and its diameter is 13.

If we compare the radius, diameter, and size of the center of the main cluster C for different thresholds, we notice that as we get closer to a suitable pair of thresholds,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the radius and diameter do not change significantly, but there is a significant decrease in the size of the center. The size of the center for pairs of thresholds $(0.7985, 0.8029)$ and $(0.7986, 0.8028)$ are 75 and 34, respectively; whereas, the size of the center for the suitable pair $(0.7986, 0.8029)$ is only 7. The vertices in the center for the suitable pair of thresholds $(0.7986, 0.8029)$ are $12, 18, 37, 160, 203, 211$, and 254.

Using the algorithms developed in Chapter 5, we are able to find the distance of vertices in a connected component from a given vertex (provided by the user). Here, what we computed is the distance in the underlying graph of the gene expression graph, that is, the distance in the graph without weights. We could also compute distance in the weighted graph, but this was beyond the scope of this thesis. We hereby present an example where the chosen gene is in the center of the connected component C. Figure 10 shows the distribution of distances in the connected component C from vertex 203.

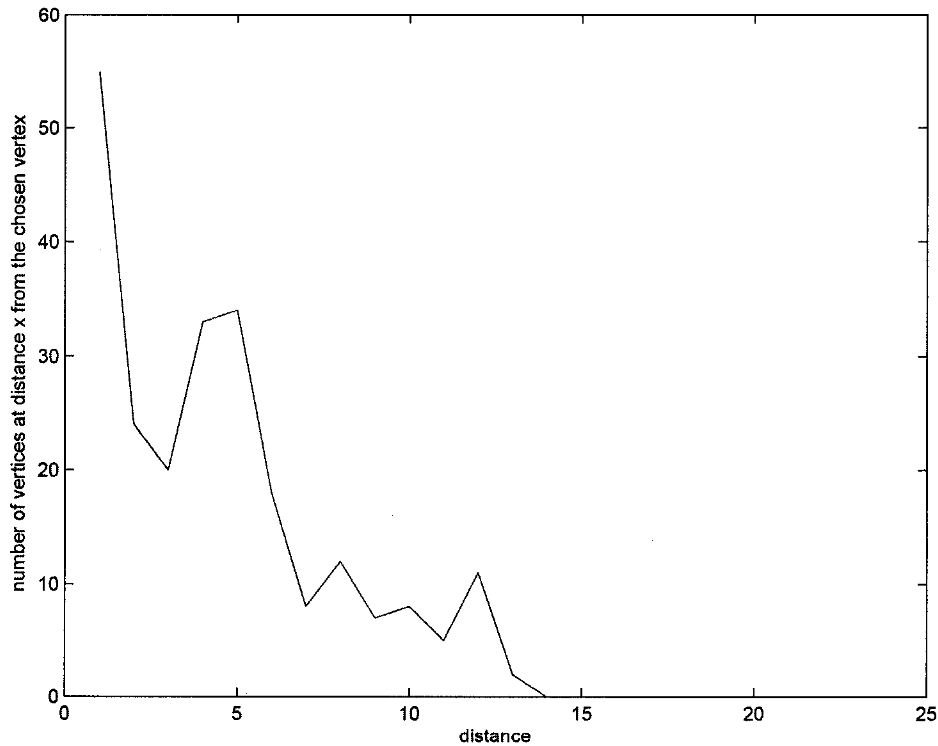

Figure 10

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For comparison, Figure 11 shows the distribution of distances in the connected component C from vertex 179, which is not in the center.

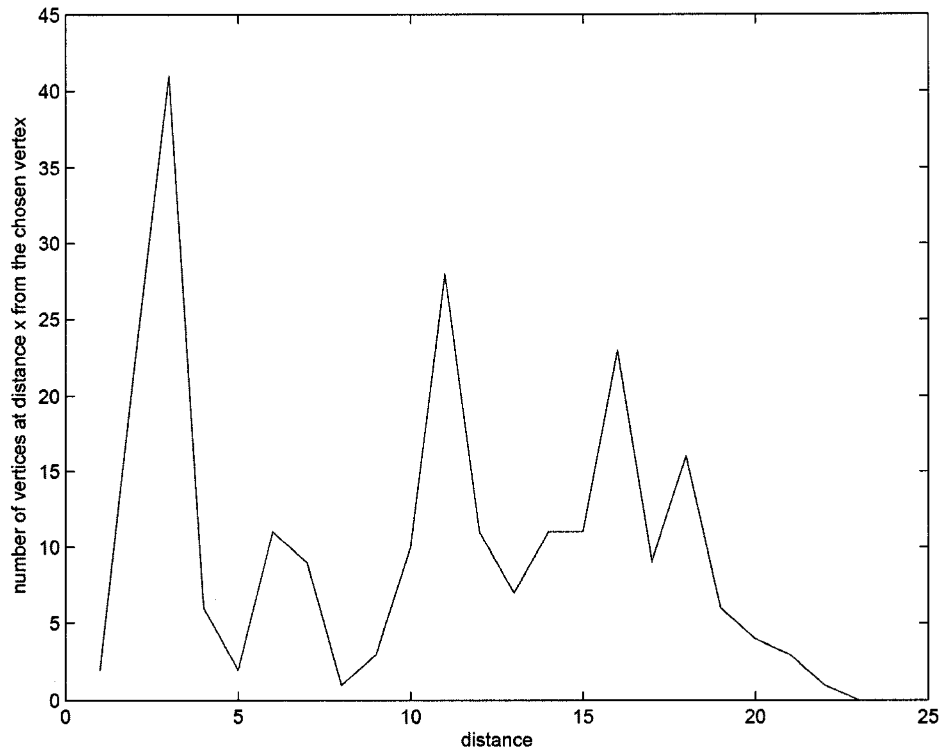

Figure 11

Note that Figure 10 shows a marked decrease in the number of vertices as the distance from 203 increases. This is justified by considering that 203 is in the center of the graph. Vertex 179, which is not in the center of the connected component C, does not show the same decreasing property (Figure 11).

The main goal of this research was to develop a tool that assists in designing future experiments, and possibly obtain information about the structure of the gene expression network by using graph theoretical methods. By using our model and method of analysis, several possible questions can be answered. Besides finding a suitable pair of thresholds and determining the general structure of the graph such as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the number and sizes of the clusters and their density, we can answer certain questions about specific genes. These include the following:

- For any gene (for example one of biological interest), we can determine the following: degree, eccentricity, distances to all other genes in the same cluster, size of the cluster containing it, and whether or not the gene lies in the center or median. Also, we can determine its "behavior" while changing the thresholds; for example, we can find out if a particular gene that is in the center keeps its position in the center as the threshold varies.

- For a specified pair (or a larger set) of genes we can determine whether or not they lie in the same cluster. If they are in the same cluster, we can find the distance between them. Also, we can determine how they "behave" as we change the thresholds; whether they stay in the same cluster, or their relative positions change as we change the thresholds.

Clearly, the numerical results obtained in this thesis depend on the chosen set of genes and cell samples. However, the approach taken in this thesis can be applied to any set of data of gene expression levels that might be of interest to researchers in medical sciences. The notions of a gene expression graph and perfect correlation graph, and the theory developed, are independent of the genes and cell samples chosen. Therefore, the lemmas, theorems, algorithms, and code that we have used in this thesis can be applied to any (sufficiently small) database of gene expression levels.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 11

Further Research

This thesis started a new approach for obtaining information about functional relationships between genes by analyzing the data of gene expression levels from a graph-theoretical point of view. This approach is based on applying mathematical theory to biological data, and it takes advantage of other areas such as computer science and statistics. Therefore, there are several possible ways for future work on this topic. Here, we will present some suggestions and ideas for continuing this research.

1. Extending the parameters. This would include adding the calculation of other biologically meaningful parameters for the gene expression graph. Chapters 6 and 7 of this thesis contain results and algorithms for finding cut-vertices, blocks, and maximal and maximum cliques of a graph. To extend the analysis to these additional parameters on the graph, one could implement the algorithms presented in these two chapters. For example, cut-vertices for various thresholds can be found. A study of behaviour of cut-vertices and blocks and their biological significance might be of interest.

   Also, parameters other than the ones studied in this thesis could be explored.

2. Adding more attributes to describe the relationship between the genes. Given the size of the problem, it was important for us to start with the simplest possible scenario to be sure that the problem is computationally feasible. In subsequent work, the weights on the edges of the graph can be extended to n-tuples from

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the current single value. In this thesis we only considered correlation coefficient, which corresponds to the idea of fitting a linear function to the data. One could try to fit higher order polynomials to the data. In addition, the n-tuple representing the edge weight could include the goodness of fit (error measure), experimental error, combinations of the types of distribution of expression levels of the two genes (unimodal, bimodal), etc.

3. Extending the graph. In this thesis we have performed the analysis for the 256 selected genes of highest variance of the gene expression level over the set of samples. One option is to perform a similar analysis for a larger set of genes available from StemBase. This may possibly lead to analysis of the gene expression levels for all of the 45,000 genes. Clearly, in choosing this option one should study and examine available software and hardware for larger computations.

4. Applying the same method to other data. The approach taken in this thesis is based on the data from gene expression levels. One possible extension is to use a similar approach in the analysis of other data. For example, a similar graph-theoretical analysis may be possible for a database of proteins instead of genes, and post-transcriptional factors instead of transcription factors. This may assist in proteomics instead of genomics [17].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[2] U. Alon et al., Broad Patterns if Gene Expressin Revealed by Clustering Analysis of Tumor and Normal Colon Tissues Probed by Oligonucleotide Array, *Proc. Natl. Acad. Sci. USA*, Vol. 96(12), 6745-6750, June 1999.

[3] A. Ameur et al., Global Gene Expression Analysis by Combinatorial Optimization, *Silico Biology*, 4 (20), 2004.

[4] M.B. Avison, *Meauring Gene Expression*, Taylor and Francis, 2007.

[5] S. Baase, *Computer Algorithms, Introduction to Design and Analysis*, Addison-Wesley, 1993.

[6] R. Breitling et al., Graph-based Iterative Group Analysis Enhances Microarray Interpretation, *BMC Informatics*, 5(100), July 2004.

[7] J. Bondy and U.S.R. Murty, *Graph Theory with Applications*, Macmillan Press, 1976.

[8] F. Buckley and F. Harary, *Distance in Graphs*, Addison-Wesley Publishing Company, 1990.

[9] F. Buckley and M. Lewinter, *A Friendly Introduction to Graph Theory*, Prentice-Hall, 2003.

[10] G. Casella and R.L. Berger, *Statistical Inference*, Brooks/Cole, 1990.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[11] G. Chartrand and O. Oellermann, *Applied and Algorithmic Graph Theory*, Mcgraw-Hill, 1993.

[12] Y. Cheng and G. Churuch, Biclustering of Expression Data, *Proc. Eighth International Conference on Intelligent Systems for Molecular Biology*, 8, 93-103, July 1998.

[13] T. Chu, Limitations of Statistical Learning from Gene Expression Data, *Proc. Interface 2004: Computational Biology and Bioinformatics*, May 2004.

[14] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall, 1974.

[15] R. Diestel, *Graph Theory*, Springer, 2005.

[16] Eisen et al., Cluster Analysis and Display of Genome-wide Expression Patterns, *Proc. Natl. Acad. Sci. USA*, 95(25), 14863-14868, December 1998.

[17] C.S. Findlay, Oral communication.

[18] A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

[19] E.G. Goodaire and M.M. Paramenter, *Discrete Mathematics with Graph Theory*, Prentice-Hall, 2002.

[20] R.P. Grimaldi, *Discrete and Combinatorial Mathematics: An Applied Introductio*, Addison-Wesley Longman, 1999.

[21] J. Gross and J. Yellen, *Graph Theory and Its Applications*, CRC Press, 1999.

[22] W. Hofmann, *Gene Expression Profiling by Micorarrays*, Cambridge University Press, 2006.

[23] R.V. Hogg and E.A. Tanis, *Probability and Statistical Inference*, Macmillan, 1983.

[24] E. Horowitz and S. Sahni, *Fundamentals of Data Structure*, Computer Science Press, 1982.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[25] D. Jiang, J. Pei and A. Zhang, Towards Interactive Exploration of Gene Expression Patterns, *ACM SIGKDD Explorations, Special Issue on Microarray Data Analysis*, Vol 5(2), 79-90, 2003.

[26] D. Jiang, J. Pei and A. Zhang, A Density-based hierarchical Clustering Method for Time Series Gene Expression Data, *Proc. Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington DC, USA*, 2003.

[27] B. Kaba et al., Clustering Gene Expression Data using Graph Separators, *Publications of LIMOS*, LIMOS/RR-07-02, May 2007.

[28] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers*, John Wiley, 1999.

[29] H. Pearson, *Genetics: what is a gene?*, Nature 441 (7092): 398-401, 2006.

[30] J. Pei et al., Mining Corss-Graph Quasi-Cliques in Gene Expression and Protein Interaction Data, *Proc. 21 International Conference on Data Engineering*, Japan, April 2005.

[31] C. Perez-Iratxeta, G. Palidwor, C.J. Porter, N.A. Sanche, M.R. Huska, B.P. Suomela, E.M. Muro, P.M. Krzyzanowski, E. Hughes, P.A. Campbell, M.A. Rudnicki and M.A. Andrade, Study of stem cell function using microarray experiments, *FEBS Letters* 579, 1795-1801, 2005.

[32] M. Schena, *Microarray Analysis*, John Wiley and Sons, 2003.

[33] S. Seno et al., A Method for Clustering Gene Expression Data Based on Graph Structure, *Genome Informatics*, 15(2), 151-160, 2004.

[34] P. Tamayo et al., Interpreting Patterns of Gene Expression with Self-Organizing Maps: Methods and Applications to Hematopoietic Differentiation, *Proc. Natl. Acad. Sci. USA*, 96(6), 2907-2912, March 1999.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[35] A. Tanay, R. Sharan and R. Shamir, Discovering Statistically Significant Biclusters in Gene Expression Data, *Bioinformatics*, 18(1), 136-144, 2002.

[36] S. Tavazoie et al., Systematic Determination of Genetic Network Architecture, *Nature Genet*, 281-285, 1999.

[37] A. Tsalenko et al., Analysis of SNP-Expression Association Matrices, em Journal of Bioinformatics and Computational Biology, 4(2), 259-274, 2006.

[38] G. Valiente, *Algorithms on Trees and Graphs*, Springer, 1998.

[39] R.E. Walpole, R.H. Myers, S.L. Myers and K. Ye, *Probability and Statistics for Engineers and Scientists* Prentice Hall,2007.

[40] J.D. Watson, T.A. Baker, S.P. Bell, A. Gann, M. Levine, R. Losick, *Molecular Biology of the Gene*, Benjamin Cummings, 2004.

[41] Y. Xu, D. Xu and V. Olman, Clustering Gene Expression Data Using a Graph-Theoretic Approach: An Application of Minimum Spanning Trees, *Bioinformatics*, 18(4), 536-545, April 2002.

[42] C. Yeang, Discovering Biological Functional Modules from Gene Expression Data, http://www.ai.mit.edu/research/abstracts/abstract2000/pdf/2-yeang.

[43] http://www.ottawagenomecenter.ca/projects/stembase/

[44] http://www.ottawagenomecenter.ca/projects/SCNcourse, online microarray analysis courses created by M. Andrade's bioinformatics lab at the Ottawa Health Research Institute.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.