GENEPART ALGORITHM, CLUSTERING AND FEATURE SELECTION FOR DNA MICRO-ARRAY DATA

by

Weihua Zhang

A thesis submitted in partial fulfillment
Of the requirements for the degree
of
Master of Science
in
Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

November 2004

©COPYRIGHT

by

Weihua Zhang

2004

All Rights Reserved

APPROVAL

of a thesis submitted by

Weihua Zhang

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Approved for the thesis committee chair

Brendan. Mumey, Ph.D.

Approved for the Department of Computer Science

Michael. Oudshoorn, Ph.D.

Approved for the College of Graduate Studies

Bruce R. McLeod, Ph.D.

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Weihua Zhang

11/29/2004

TABLE OF CONTENTS

1. INTRODUCTION 1

2. BACKGROUND THEORY 3

DNA Micro- Array Data 3

Branch and Bound 9

3. GENEPART ALGORTHM BACKGROUND 13

Model Background 13

GENEPART Algorithm 17

Selected Gene 24

4. ALGORITHM ANALYSIS 28

Complexity Analysis 28

5. EXPERIMENTAL RESULTS 33

Test Data Sets 33

Description of The Experiment Source Data File 35

BRCA Data Set Test 38

Adenoma Data Set Test 41

6. CONCLUSIONS 43

REFERENCES CITED 45

APPENDICES 48

APPENDIX A 49

APPENDIX B 79

APPENDIX C 81

LIST OF TABLES

TABLE Page

1. 2-PARTITION BREAST CANCER DATA SET RESULT 39

2. 3-PARTITION BREAST CANCER DATA SET RESULT 40

3. ADENOMA RESULT 42

LIST OF FIGURES

Figure Page

1. GEGE STRUCTURE[13] 5
2. GENE STRUCTURE IN EUKARYOTES 6
3. AN EXAMPLE OF MICRO-ARRAY IMAGE[13] 7
4. CDNA MICROARRAY[13] 8
5. ILLUSTRATION OF THE SEARCH SPACE OF B&B 12
6. THE GENE EXPRESSION PROFILE HISTOGRAMS 16
7. COLOR CHANGE EXAMPLE 17
8. GENE RANKED BY MIXTURE-OVERLAP PROBABILITY 19
9. COLOR CHANGE EXAMPLE 26
10. OVERVIEW OF CDNA MICROARRAY AND TISSUE MICROARRAY PROCEFURE[9] 34
11. EXPRESSION INTERSITY IN NORMAL COMPARED TO TUMOR SAMPLES 41
12. EXPRESSION DATA MATRIX FILE EXAMPLE 80
13. GENE INFO FILE EXAMPLE 80
14. SUBSET OF GENES FROM THE TOP 176 ONES WITH FEWEST COLOR CHANGES, 3-PARTITION, BRCA DATA SET 82
15. SUBSET OF GENES FORM THE TOP 200 ONES WITH FEWEST COLOR CHANGES, 2-PARTITION, ADENOMA DATA SET 83
16. SUBSET OF THE LISTED DISCRIMINATION GENES FOR BRCA DATA SET 84

ABSTRACT

This paper provides the theoretical analysis of a new clustering and feature selection algorithm for the DNA micro-array data. This algorithm utilizes a branch and bound algorithm as the basic tool to quickly generate the optimal tissue sample partitions and select the gene subset which contributes the most to certain sample partition, it also combines the statistical probability method to identify important genes that have meaningful biological relationships to the classification or clustering problem. The proposed method combines feature selection and clustering processes and can be applied to the diagnostic system. Simulation results and analysis shown in the paper support the effectiveness of the combined algorithm.

CHAPTER 1

INTRODUCTION

DNA micro-array analysis is well known as a problem with high-dimensional space and small sample set. It exemplifies a situation that will be increasingly common in the analysis of micro-array data using machine learning techniques such as classification or clustering, feature selection methods are essential particularly if the goal of the study is to identify genes whose expression patterns have meaningful biological relationships to the classification or clustering problem. This problem is very valuable in clinical as well as theoretical study, the main problem is to cluster samples into homogeneous groups that may correspond to particular macroscopic phenotypes, such as clinical syndromes or cancer types. In practice, a working mechanistic hypothesis that is testable and largely captures the biological truth seldom involves more than a few dozens of genes, and knowing the identity of these relevant genes is just as important as finding the grouping of samples they induce. Thus, finding a way that involves an interplay between clustering and feature selection is encouraging.

The goal of feature selection is to select relevant features and eliminate irrelevant ones. This can be achieved by either explicitly looking for a good subset of features, or by assigning all features appropriate weights. Explicit feature selection is generally most

natural when the result is intended to be understood by humans or fed into different induction algorithms. Feature weighting, on the other hand, is more directly motivated by pure modeling or performance concerns. The weighting process is usually an integral part of the induction algorithm and the weights often come out as a byproduct of the learned hypothesis.

The paper continues in Chapter 2, introduce some of the background theory behind branch and bound algorithm and DNA micro-array data. The background analysis of the GENEPART algorithm introduced by Professor Brendan Mumey [14] will be discussed in Chapter 3. Chapter 4 provides the analysis of the algorithm ,testing data set description and the experimental results. The paper concludes with observations and suggestions for future work in Chapter 5.

CHAPTER 2

BACKGROUND THEORY

DNA Micro-Array Data

Genetics as a set of principles and analytical procedures did not begin until 1866, when an Augustinian monk named Gregor Mendel performed a set of experiments that pointed to the existence of biological elements called genes - the basic units responsible for possession and passing on of a single characteristic. Until 1944, it was generally assumed that chromosomal proteins carry genetic information, and that DNA plays a secondary role. This view was shattered by Avery and McCarty who demonstrated that the molecule deoxy-ribonucleic acid (DNA) is the major carrier of genetic material in living organisms, i.e., responsible for inheritance. In 1953 James Watson and Francis Crick deduced the three dimensional double helix structure of DNA and immediately inferred its method of replication (see [2], pages 859-866). In February 2001, due to a joint venture of the Human Genome Project and a commercial company Celera (www.celera.com), the first draft of the human genome was published.

A gene is a region of DNA that controls a discrete hereditary characteristic, usually corresponding to a single mRNA carrying the information for constructing a protein (see [4], pages 98-99). It contains one or more regulatory sequences that either increase or decrease the rate of its transcription (see Figure 1). In 1977 molecular biologists

discovered that most Eukaryotic genes have their coding sequences, called exons, interrupted by non-coding sequences called introns (see Figure 2). In humans genes constitute approximately 2-3% of the DNA, leaving 97-98% of non-genic junk DNA. The role of the latter is as yet unknown, however experiments involving removal of these parts proved to be lethal. Several theories have been suggested, such as physically fixing the DNA in its compressed position, preserving old genetic data, etc.

It is widely believed that thousands of genes and their products (i.e., RNA and proteins) in a given living organism function in a complicated and orchestrated way that creates the mystery of life. However, traditional methods in molecular biology generally work on a "one gene in one experiment" basis, which means that the throughput is very limited and the "whole picture" of gene function is hard to obtain. In the past several years, a new technology, called DNA microarrays, has attracted tremendous interests among biologists. This technology promises to monitor the whole genome on a single chip so that researchers can have a better picture of the interactions among thousands of genes simultaneously.

Terminologies that have been used in the literature to describe this technology include, but not limited to: biochip, DNA chip, DNA microarray, and gene array. An array is an orderly arrangement of samples. Those samples can be either DNA or DNA products. Each spot in the array contains many copies of the sample. The array provides a medium for matching known and unknown DNA samples based on base-pairing

(hybridization) rules and automating the process of identifying the unknowns. The sample spot sizes in microarray are typically less than 200 microns in diameter and these arrays usually contain thousands of spots. As a result microarrays require specialized robotics and imaging equipment. An experiment with a single DNA chip can provide researchers information on thousands of genes simultaneously - a dramatic increase in throughput.

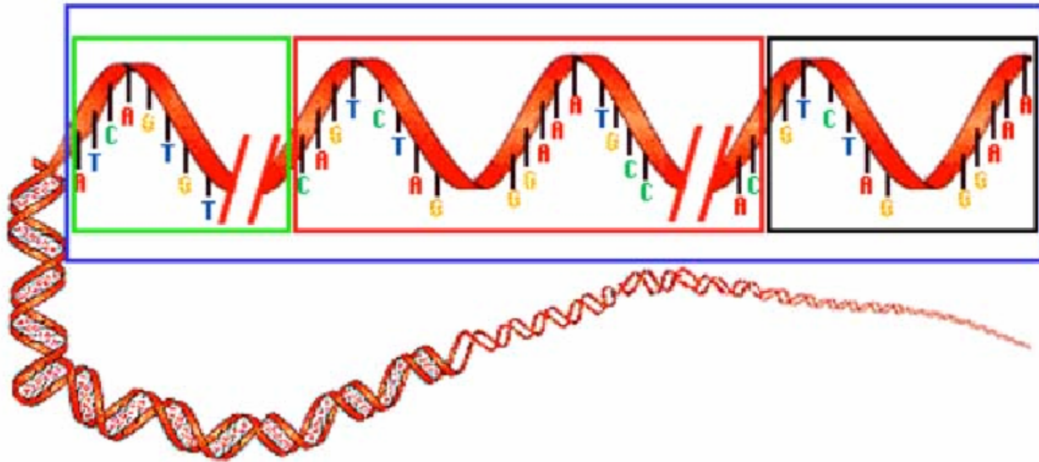

Figure 1 Gene structure[13]

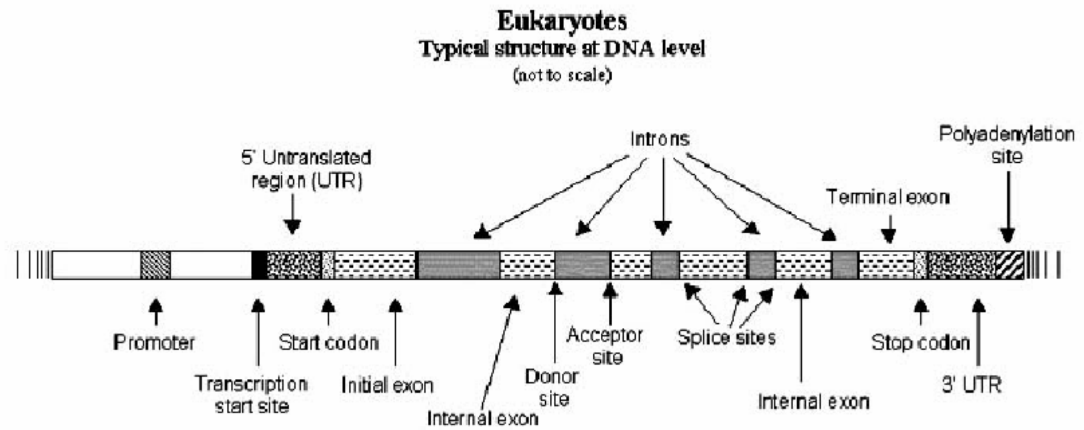

Figure 2 Gene structure in Eukaryotes

This technology enables a researcher to analyze the expression of thousands of genes in a single experiment and provides quantitative measurements of the differential expression of these genes. In this approach, each spot in the chip contains, a cDNA clone, which represents a gene. The chip as a whole represents thousands of genes. The target is the mRNA extracted from a specific cell. Since almost all the mRNA in the cell is translated into a protein, the total mRNA in a cell represents the genes expressed in that cell. Therefore hybridization of mRNA is an indication of a gene being expressed in the target cell. Since cDNA clones are very long (can be thousands of nucleotides), a successful hybridization with a clone is an almost certain match for the gene. However, due to the different structure of each clone and the fact that unknown amount of cDNA is printed at each probe, we cannot associate directly the hybridization level with transcription level and so cDNA chips experiments are limited to comparisons of a

reference extract and a target extract. Comparative genomic hybridization is designed to help clinicians determine the relative amount of a given genetic sequence in a particular patient. This type of chip is designed to look at the level of aberration. This is usually done by using a healthy tissue sample as a reference and comparing it with a sample from the diseased tumor. To perform a cDNA array experiment, we label green the reference extract, representing the normal level of expression in our model system, and label red the target culture of cells which were transformed to some condition of interest. Usually we hybridize the mixture of reference and target extracts and read a green signal in case the condition reduced the expression level and a red signal in case our condition increased the expression level.

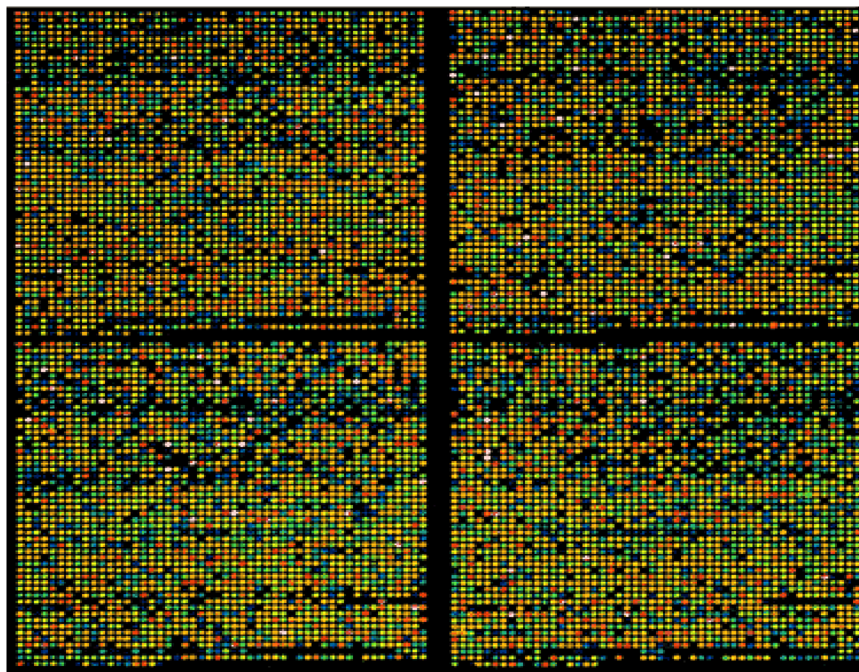

Figure 3 An example of micro-array image[13]

The intensity and color of each spot encode information on a specific gene from the tested sample.

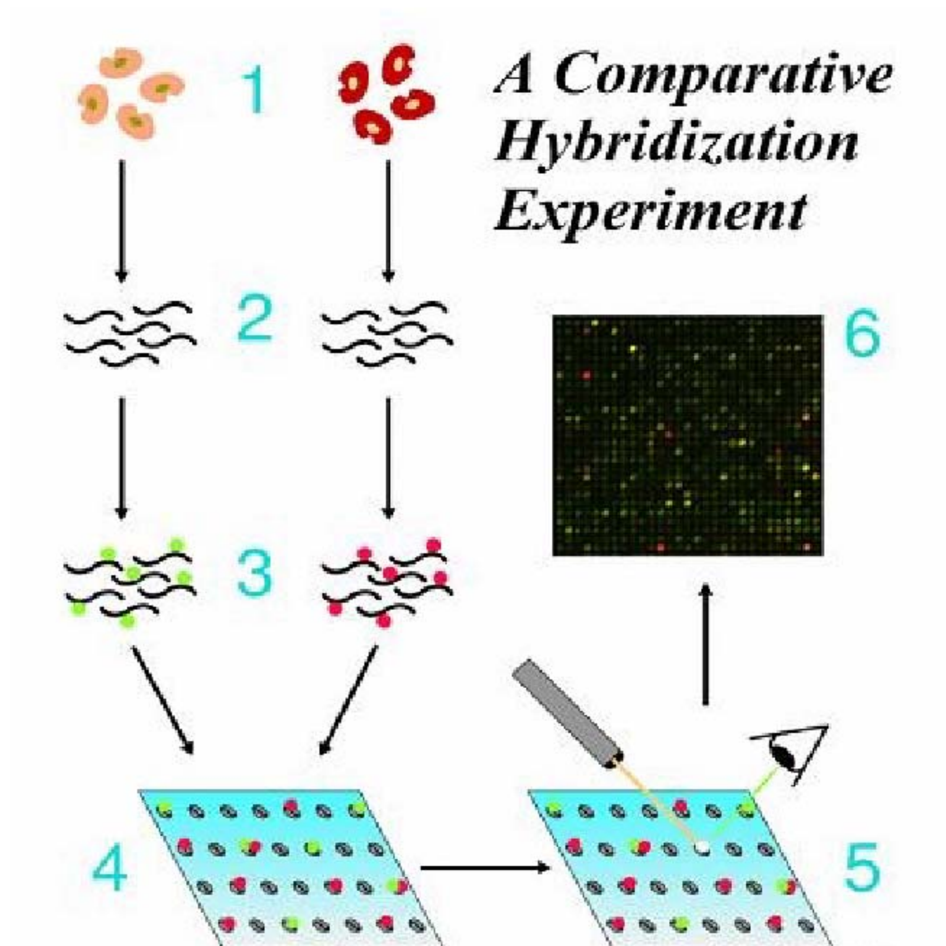

Figure 4 cDNA Microarray[13]

1) Two cells to be compared. On the left is the reference cell and on the right the target cell. 2) The mRNA is extracted from both cells. 3) Reference mRNA is labeled green, and the target mRNA is labeled red. 4) The mRNA is introduced to the Micro-array. 5) According to the color of each gene clone the relative expression level is educed. 6) cDNA chip after scanning.

Branch and Bound

Branch-and-bound is an approach developed for solving discrete and combinatorial optimization problems. The discrete optimization problems are problems in which the decision variables assume discrete values from a specified set; when this set is set of integers, we have an integer programming problem. The combinatorial optimization problems, on the other hand, are problems of choosing the best combination out of all possible combinations. Most combinatorial problems can be formulated as integer programs (see [11] an excellent review of these problems).

The major difficulty with these problems, is that we do not have any optimality conditions to check if a given (feasible) solution is optimal or not. For example, in linear programming we do have an optimality condition: Given a candidate solution, I'll check if there exists an "improving feasible direction" to move, if there isn't, then your solution is optimal. If I can find a direction to move that results in a better solution, then your solution is not optimal. There are no such global optimality conditions in discrete or combinatorial optimization problems. In order to guarantee a given feasible solution's optimality is "to compare" it with every other feasible solution. To do this explicitly, amounts to total enumeration of all possible alternatives which is computationally prohibitive due to the NP-Completeness of integer programming problems. Therefore, this comparison must be done implicitly, resulting in partial enumeration of all possible alternatives.

The essence of the branch-and-bound approach is the following observation: in the total enumeration tree, at any node, if one can show that the optimal solution cannot occur in any of its descendents, then there is no need to consider those descendent nodes. Hence, we can ″prune″ the tree at that node. If enough branches of the tree can be pruned in this way, it may be reduced to a computationally manageable size. Note that, we are *not* ignoring those solutions in the *leaves* of the branches that have been pruned, we have left them out of consideration *after* we have made sure that the optimal solution cannot be at any one of these nodes. Thus, the branch-and-bound approach is not a heuristic, or approximating, procedure, but it is an exact, optimizing procedure that finds an optimal solution.

How can we make sure that the optimal solution cannot be at one of the descendents of a particular node on the tree? An ingenious answer to this question was given, independently by K. G. Murty, C. Karel**,** and J. D. C. Little in 1962 [12] in an unpublished paper in the context of a combinatorial problem, and by A. H. Land and A. G. Doig in 1960 [1]. It is always possible to find a feasible solution to a combinatorial or discrete optimization problem. If available, one can use some heuristics to obtain, usually, a "reasonably good" solution. Usually we call this solution *the incumbent*. Then at any node of the tree, if we can compute a "bound" on the best possible solution that can expected from any descendent of that node, we can compare the "bound" with the objective value of the incumbent. If what we have on hand, the incumbent, is better than

what we can ever expect from any solution resulting from that node, then it is safe to stop branching from that node. In other words, we can discard that part of the tree from further consideration.

At any point during the solution finding process, the status of the solution with respect to the search of the solution space are described by a pool of yet unexplored subset of this and the best solution found so far. Initially only one subset exists, namely the complete solution space, and the best solution found so far is ∞. The unexplored subspaces are represented as nodes in a dynamically generated search tree, which initially only contain the root, and each iteration of a classical B&B algorithm processes one such node. The iteration has three main components: selection of the node to process, bound calculation, and branching. In Figure 5, the initial situation and the first step of the process is illustrated.

The sequence of process may vary according to the strategy chosen for selecting the next node to process. If the selection of next sub-problem is based on the bound value of the sub-problems, then the first operation of an iteration after choosing the node is branching, i.e. subdivision of the solution space of the node into two or more subspaces to be investigated in a subsequent iteration. For each of these, it is checked whether the subspace consists of a single solution, in which case it is compared to the current best solution keeping the best of these. Otherwise the bounding function for the subspace is calculated and compared to the current best solution. If it can be established that the

subspace cannot contain the optimal solution, the whole subspace is discarded, else it is stored in the pool of live nodes together with it's bound. This is so called the eager strategy for node evaluation, since bounds are calculated as soon as nodes are available. An alternative way is to start by calculating the bound of the selected node and then branch on the node if necessary. The nodes created are then stored together with the bound of the processed node. This strategy is called lazy and is often used when the next node to be processed is chosen to be a live node of maximal depth in the search tree.

The search terminates when there are no unexplored parts of the solution space left, and the optimal solution is then the one recorded as "current best".

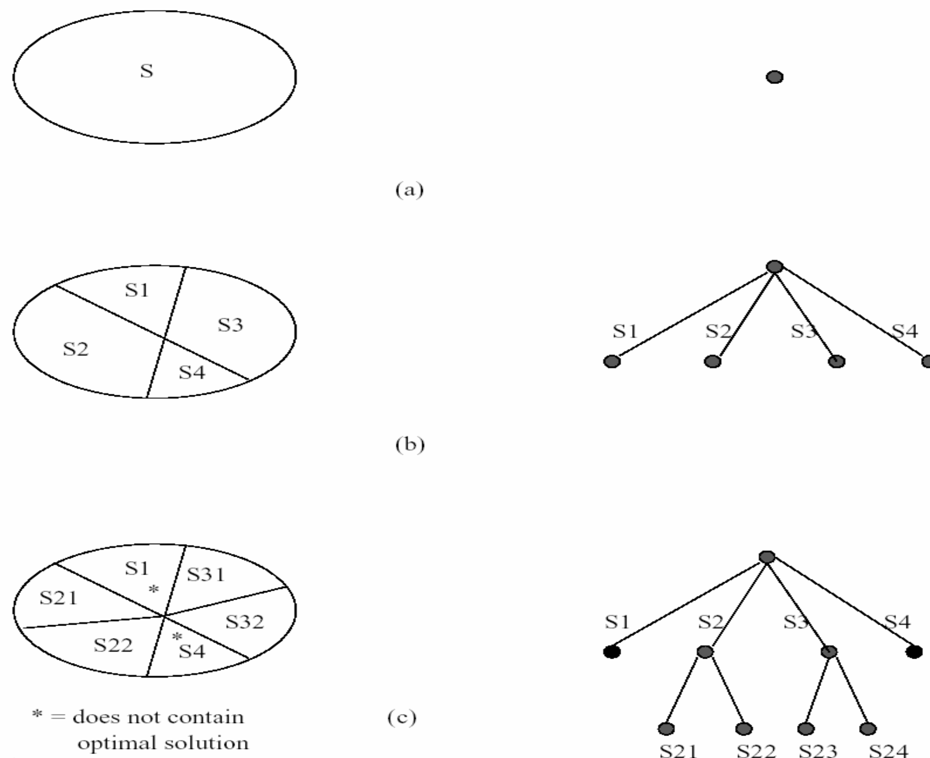

Figure 5 Illustration of the search space of B&B

CHAPTER 3

GENEPART ALGORITHM BACKGROUND

Model Background

The outcome of DNA Micro-arrays is a matrix associating for each gene (row) and condition/profile (column) the expression level. Expression levels can be absolute or relative. We wish to identify biological meaningful phenomena from the expression matrix, which is often very large (thousands of genes and hundreds of conditions). The most popular and natural first step in this analysis is clustering of the genes or experiments. Clustering techniques are used to identify subsets of genes that behave similarly under the set of tested conditions. By clustering the data, the biologists can view the data in a concise way and try to interpret it more easily. Using additional source of information (known genes annotations or conditions details), one can try and associate each cluster with some biological semantics. Other computational challenges are:

- Classification: Given a partition of the conditions into several types, classify an unknown tissue.
- Feature selection: Given a partition of the conditions into several types, find a subset of the genes for each type that distinguishes it from the rest.
- Normalization: How does one best normalizes thousands of signals from same/different conditions/experiments?

- Experiment design: Choose which (pairs of) conditions will be most informative. The cells must differ in the condition under research but be alike as much as possible in all other aspects (phenotypes) in order to avoid distractions.
- Detect regulatory signals in promoter regions of co-expressed genes.

One recent paper by Eric P.Xing and Richard M.Karp [19] introduces an interesting algorithm which iterates between two computational processes, feature filtering and clustering. Given a reference partition that approximates the correct clustering of the samples, the feature filtering procedure ranks the features according to their intrinsic discriminability, relevance to the reference partition, and irredundancy to other relevant features, and uses this ranking to select the features to be used in the following round of clustering. The clustering algorithm, which is based on concept of a normalized cut, clusters the samples into a new reference partition on the basis of the selected features. Other randomized approaches to unsupervised clustering have also been proposed, in particular the CLICK algorithm based on a graph-clustering algorithm.[5,6,17].

Before introducing the structure of the GENEPART algorithm, one important theory needs to mentioned, that is: Discretization and Discriminability assessment of features [20]. The measurements we obtained from micro-arrays are continuous values. In many situations in functional annotations (e.g., constructing regulatory networks) or data analysis (e.g., the information-theoretic-based filter technique), however, it is convenient

to assume the entries are discrete values. One way to achieve this is to deduce the functional states of the genes based on their observed measurements. A widely adopted empirical assumption about the activity of genes and hence their expression, is that they generally assume distinct functional states such as ‘on’ or ‘off’. (We assume binary states for simplicity but generalization to more states is straightforward). The combination of such binary patterns from multiple genes determines the sample phenotype. And a feature with discriminative power should have bimodal distribution, a simple model would be a mixture of two univariate Gaussians. Consider a particular gene i (feature F_i), suppose that the expression levels of F_i in those samples where F_i is in the ’on’ state can be modeled by a probability distribution, such as Gaussian distribution $N(x \mid m_1, s_1)$ where m_1 and s_1 are the mean and standard deviation. Similarly another Gaussian distribution $N(x \mid m_2, s_2)$ can be assumed to model the expression levels of F_i in those samples where F_i is in the ‘off’ state. Given the above assumptions, the marginal probability of any given expression level x_i of gene i can be modeled by a weighted sum of the two Gaussian probability functions corresponding to the two functional states of this gene (where the weight $\pi_{1/2}$ correspond to the prior probabilities of gene being in the on/off states):

$$P(x_i) = \pi_1 N(x_i \mid m_1, s_1) + \pi_2 N(x_i \mid m_2, s_2)$$

This univariate mixture model with two components includes the degenerate case of a single component when either of the weights is zero. The histogram in Figure 6a gives

the empirical marginal of a gene in leukemia samples data set, which clearly demonstrates the case of the two-component mixture distribution of the expression levels of this gene, whereas Figure 6b is an example of a nearly uni-component distribution (which indicates that this gene remains in the same functional state in all the samples, therefore have no discriminative power in clustering process)

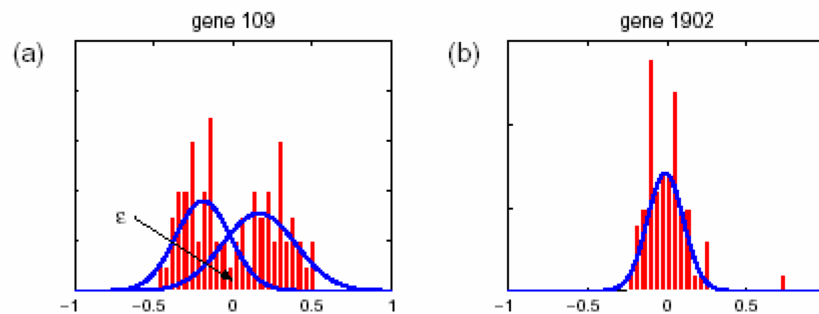

Figure 6 The gene expression profile histograms

These are estimated density functions of the expression profiles of two representative genes. The x-axes represent the normalized expression level. [20]

For feature selection, if the underlying binary state of the gene does not vary between the two classes, then the gene is not discriminative for the classification problem and should be discarded. This suggests a heuristic procedure in which we measure the separability of the mixture components as an evidence of the discrimination of the feature.

GENEPART Algorithm

The GENEPART algorithm is based on the recent research by Brendan Mumey, 2004 [14], a partition and coloring method is introduced to provide the criteria of ranking the genes according to their expression behavior. Given a DNA micro array data set, suppose that we have partitioned the tissue samples, and a unique color is assigned to each sample partition. For a particular gene, we can color the value of this gene according to the partitions colors of the corresponding colored samples. Then we sort the colored value of this gene, finally, the specific gene's amount of color changes can be counted. This procedure can be illustrated by a simple example shown in Figure 7.

	Tissue sample				
	0	**1**	**2**	**3**	**4**
Gene values	**~~2.3~~**	**0.2**	**~~10.6~~**	**8.2**	**4.5**
Partition	**~~1~~**	**0**	**~~1~~**	**0**	**0**
Values sorted	**0.2**	**~~2.3~~**	**4.5**	**8.2**	**~~10.6~~**

Number of color changes = 3

Figure 7 color change example

According to the above assumption and analysis, it's safe to say that the gene with less color changes is likely playing an more important role or is informative of contributing for the given sample partitioning. If we find a set of genes that all have low color changes, we can say there is evidence that the current sample partition is

meaningful for the tissue clustering and this set of genes are relevant in the biological processes that discriminate the tissue sample classes.

Given N microarray experiments for which gene is measured in each experiment, the complete likelihood of all observations $X_i = \{x_{1i}, ..., x_{Ni}\}$ and their corresponding state indicator $Z_i = \{z_{1i}, ..., z_{Ni}\}$ is:

$$P_c(X_i, Z_i \mid \theta_i) = \prod_{n=1}^{N} \prod_{k=0}^{1} (\pi_{i.k} [\frac{1}{\sqrt{2\pi} s_{i,k}} \exp\{-\frac{(x_{ni} - m_{i,k})^2}{2(s_{i,k})^2}\}])^{z_{ni}^k}$$

Random variable $z_{ni} \in \{0,1\}$ indicates the underlying state of gene i in sample (we omit sample index in the subscript in the later presentation for simplicity) and is usually latent. The solid curve in Figure 6a depict the density functions of the two Gaussian components fitted on the observed expression levels of the gene. The curve in Figure6b is the density of the single-component Gaussian distribution fitted on another gene. Note that each feature is fitted independently based on its measurements in all N microarray experiments.

Suppose we define a decision $d(F_i)$ on feature F_i to be 0 if the posterior probability of $\{z_i = 0\}$ is greater than 0.5 under the mixture model, and let $d(F_i)$ equal 1 otherwise. We can define a mixture-overlap probability:

$$\varepsilon = P(z_i = 0) P(d(F_i) = 1 \mid z_i = 0) + P(z_i = 1) P(d(F_i) = 0 \mid z_i = 1)$$

If the mixture model were a true representation of the probability of gene expression, then ε would represent the Bayesian error of classification under this model (which

equals to the area indicated by the arrow in Figure 6a). We can use this probability as a heuristic surrogate for the discriminating potential of the gene. Figure 8 shows the mixture overlap probability ε for the genes in the leukemia dataset in ascending order. It can be seen that only a small percentage of the genes have an overlap probability significantly smaller than 0.5, where 0.5 would constitute a random guessing under a Gaussian model if the underlying mixture components were constructed as class labels. Accordingly, it's straightforward to see that those genes with smaller overlap probability will generate less color changes given certain sample partition, thus, these genes' behavior vary between different expression state for different sample partition, which make them 'interesting' to our analysis, and what's more, if we find a set of genes that all have low color changes over the whole gene set, then it is evident that the current partitioning associate with the gene set is interesting to investigate for clustering biologically or clinically and the genes selected according to the color change are relevant in the biological processes discriminate between these tissue samples classes.

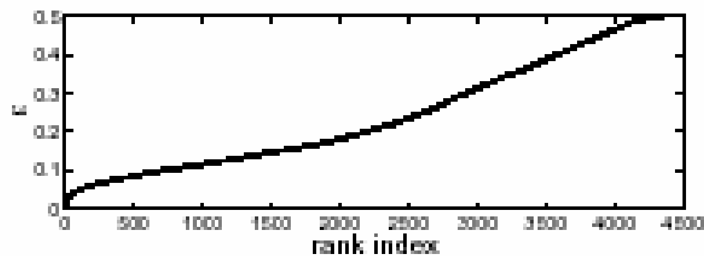

Figure 8 Gene ranked by mixture-overlap probability ε.

Only 2-state genes (those whose distributions of expressions in all samples have two mixture components corresponding to the 'on' and 'off' states) are displayed.[19]

Given the above idea, one problem remains: how to find the optimal partition/clustering of the sample sets in such a big solution space. If the total number of samples is N and the number of partition is 2, the total permutation of the samples partition would be $2^N - 1$, which makes it impossible to solve explicitly exploring every possible partition choices. Several approaches have been taken to selection features for microarray sample clustering. One approach is to group the feature into coherent sets and then project the samples onto a lower-dimensional space spanned by the average expression patterns of the coherent feature sets ([8]). This approach only deals with the feature redundancy problem, but fails to detect non-discriminating or irrelevant features. Principal component analysis (PCA) may remove non-discriminating and irrelevant features by restricting attention to so-called eigenfeatures corresponding to the large eigenvalues, but each basis element of the new feature subspace is a linear combination of all the original features, making it difficult to identify the important features ([3]). The CLIFF algorithm introduced by Xing and Karp 2002([19]) uses the approximate normalized cut algorithm [16] to iterate between selecting a subset of genes and clustering the tissue samples, the feature selection part begins with a reference partition and produces the selected gene subset using the filtering rank and the clustering part based on normalized cut partitions the samples set according to the gene subset, the program iterates between these two process until the result converges. The idea of GENEPART algorithm is similar but uses a more combinatorial approach: use the color

change as the gene selection ranking criteria and use branch and bound as the clustering algorithm to avoid testing every possible partition among the whole solution space.

Specifically, the algorithm starts with unsupervised clustering, starting with the possible partial solutions, every one will be associated with a rank calculated from the possible color changes with all the unassigned samples and the accumulate color changes of the ‘interesting’ gene subset, this gene subset is constituted by K genes with lowest color changed given current partition pattern. When the algorithm reach a ‘complete’ solution, every sample has been assigned to a partition, it’s rank will be the current lower bound, the criteria to eliminate all the partial solutions whose rank is already larger than it. And this lower bound will be updated every time when a complete solution is tested. The scenario is that although we do not know that exact target partition a priori, with respect to which we would like to optimize the feature subset, at each internal testing solution, we can expect to obtain an approximate partition that is close to the target one, and thus allows the selection of an approximately good feature subset, which will hopefully draw the partition even closer to the target one when the solution move toward completeness.

In GENEPART, each gene’s values are colored according to the array sample partitions they come from, then the values when sorted should have a minimal number of color changes; we call this the *color change* of the gene. We also consider a slightly

different scoring scheme called *black and white* change. This statistic looks at a partitioning as if one partition was colored black and remainder were colored white then counts the number of changes that occur; we chose the black partition so as to minimize the number of changes. The idea of also counting B&W changes is to find genes that are good at differentiating one class from all of the remaining classes.

We can define this formally as follows: Let m be the number of genes, n be the number of array samples and let the matrix $E = (e_{ij})$, where e_{ij} is the expression value of gene i in sample j. Suppose that $P = \{P_1, ..., P_k\}$ is a k-partitioning of the arrays samples and that $P(i)$ denotes the partition that sample i belongs to. Let π_g sort the expression values of gene g, i.e. $e_{g,\pi g(1)} <= e_{g,\pi g(2)} <= ... <= e_{g.\pi g(n)}$. Let $I()$ be a Boolean indicator function, i.e. $I[true] = 1$, $I[false] = 0$. We define:

$$colorchange(g) = \sum_{i=1}^{n-1} I[p(\pi_g(i)) \neq p(\pi_g(i+1))]$$

And:

$$B\&Wchange(g) = \min_{c=1}^{k} \sum_{i=1}^{n-1} I[|\{\pi_g(i), \pi_g(i+1)\} \cap P_c| = 1]$$

The goal of this algorithm is to find the specific partition which will minimize the number of color changes and the B&W changes given the sample cluster partition info. The formal algorithm is provided below:

- Input: $m*n$ expression data matrix, the number of partitions k, minimum class size s, the size of the selected gene set p.

- Construct search tree T, internal nodes for partial result with possible partition assignment, leaf nodes for complete result.

- For each current testing node, find the best set of genes by calculating the color changes for all genes and pick the top p, and the expected score (total color changes of the best gene set divided by the node depth in T).

- Each time a leaf node is reached, keep the best full solution expected score as the bound.

- Prune the nodes that have large score than the bound.

- Continue the search process until no search node can be found.

- Output: the current best leaf node, that is, the optimal complete solution, and the selected gene set of this node.

Selected Gene

After the GENEPART algorithm produced the result, we need to find out how to evaluate the cluster partition and the selected gene set. Normally, the partition can be verified with the experimental information which is usually given by previous research. And one way to address the gene set evaluation is to compute the number of color changes of selected gene set and the probability that you could find them in random data, this probability bound is often called 'p-value' (e.g, BLAST DNA sequence research, etc).

Specific advantages and applications of statistically sound relevance scoring are:

- Genes with very low p-values are very rare in random data and their relevance to the studied phenomenon is therefore likely to have biological, mechanistic or protocol reasons. Genes with low p-values for which the latter two options can be ruled out are interesting subjects for further investigation and are expected to provide deeper insight into the studied phenomena.
- p-values for relevance scores allow for comparing a candidate partition of the sample set in the data to a uniformly drawn partition of the same composition, in terms of the abundance of very informative genes. This serves to underline the biological meaning of a partition. In other words, this comparison statistically validates a candidate partition as having properties that would only very rarely occur for random partitions.

- In actual gene expression data it is often the case that expression levels for some genes are not reported for some samples. This is typically due to technical measurement problems. The result is that the mixture of labels that needs to be considered is dependent on the gene in question. When selecting a subset of genes as a classification platform or when looking for insight into the studied biological process we should therefore consider the relevance of each gene in the context of the appropriate mixture. Absolute score values do not provide a uniform figure of merit in this context. We use p-values as a uniform platform for such comparisons, as they do account for the mixture that defines the model.

Suppose the original microarray data matrix is m genes by n samples, p genes are found, each with t color changes or less, and the samples are partitioned into sizes($S_1,\ldots,S_k$), we need to count the number of ways that n objects partitioned into ($S_1,\ldots,S_k$) distinguishable classes with t color changes or less.

We start from simple, suppose k equals 2, that is, only 2 partitions in sample set, and we have S_1 black samples and S_2 white samples ($S_1 + S_2 = n$), suppose the number of color changes c, is even, we have 2 cases to deal with:

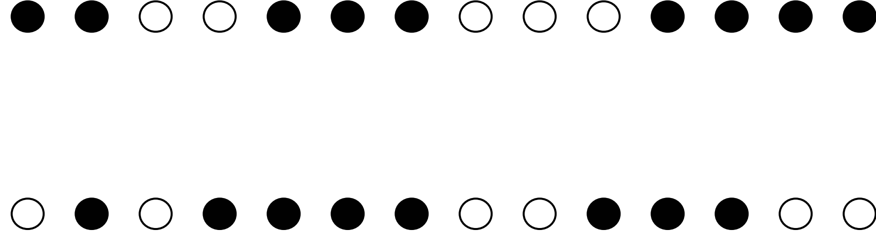

Figure 9 Color change example

In the first case, we insert S_2 white items into the sequence at $c/2$ positions chosen from $S_1 - 1$ internal black gaps, and these S_2 white items were separated into $c/2-1$ groups. There are $\binom{S_1-1}{c/2}\binom{S_2-1}{c/2-1}$ number of ways to do that. The second case is symmetric: insert S_1 black items into the sequence at $c/2$ positions chosen from $S_2 - 1$ internal white gaps, and these S_1 black items were separated into $c/2-1$ groups. The total number of distinguishable permutations is $\binom{S_1-1}{c/2}\binom{S_2-1}{c/2-1}+\binom{S_2-1}{c/2}\binom{S_1-1}{c/2-1}$.

Similar as above, when $k=2$ and c is odd, we can get the total number of distinguishable permutations is $2\binom{S_1-1}{(c-1)/2}\binom{S_2-1}{(c-1)/2}$.

Now, let $CC(\{S_1,...,S_k\},c)$ be the number of distinguishable permutations with c color changes when $k>2$. We can calculate this number by deduction: let $CC(\{S_1,...,S_{k-1}\},d)$ be the number of permutations with k-1 partitions and d color change. $CC(\{S_1,...,S_k\},c)$ will equals to $CC(\{S_1,...,S_{k-1}\},d)$ multiplied by the number of ways inserting the last partition S_k sample items into this sequence with the number of color changes increased to c. There are two cases of the inserting way that increase the number of color change:

Inserting the new items into the sequence will increase the number of color change by 1, that is, new items' neighbors belong to different partition. Let e be the number of places in the original sequence where case 1 happens when new partition items inserted.

Inserting the new items into the sequence will increase the number of color change by 2, that is, new items' neighbors belong to same partition. Let f be the number of places in the original sequence where case 2 happens when new partition items inserted.

It's straightforward to see that, inserting case 1 will happen whenever the new items are placed between the color change gaps in the original sequence as well as at the beginning and the end of the sequence. So there is $\binom{d+2}{e}$ number of ways to do that. The inserting case 2 will happen when the places where new partition items inserted are not the color change gaps, there are $n-S_k-d-1$ number of such non color change gaps. Finally, the S_k number of last partition sample items will be separated into $e+f$ groups, thus, we can calculate the total number of permutation:

$$CC(\{S_1,...,S_k\},c) = \sum_{\substack{d+e+2f=c \\ d,e,f>=0}} CC(\{S_1,...,S_{k-1}\},d)\binom{d+2}{e}\binom{n-S_k-d-1}{f}\binom{S_k-1}{e+f-1}$$

Now, we can now write down an expression for the chance of finding a $\{S_1,...,S_k\}$ partition with p genes that each has at most t color changes in random data:

$$\Pr = \binom{m}{p}\binom{n}{S_1,...,S_k}\left[\frac{\sum_{c=1}^{t} CC(\{S_1,...,S_k\},c)}{\binom{n}{S_1,...,S_k}}\right]^p$$

CHAPTER 4

ALGORITHM ANALYSIS

Complexity Analysis

Let us first briefly remind a few concepts from complexity theory. The problems polynomially solvable by deterministic algorithms belong to the P class. On the other hand, all the problems solvable by nondeterministic algorithms belong to the NP class. It can easily be shown that $P \subseteq NP$. Also, there is widespread belief that $P \neq NP$. Many problems of interest are optimization problems, in which each feasible solution has an associated value, and we wish to find a feasible solution with the best value. However, the theory of complexity is designed to be applied only to decision problems, i.e., problems which have either yes or no as an answer. Although showing that a problem is NP-complete confines to the realm of decision problems, there is a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. If we can provide evidence that a decision problem is hard, we can also provide evidence that its related optimization problem is hard. Thus, even though it restricts attention to decision problems, the theory of NP-completeness often has implications for optimization problems as well.

Informally, a decision problem X is said to be NP-complete if $X \in NP$ and for all other problems $X^{'} \in NP$, there exists a polynomial transformation from $X^{'}$ to X (we write $X^{'} \leq_{p} X$). There are two important properties of the NP-complete class. If any NP-complete problem could be solved in polynomial time, then all problems in NP could also be solved. If any problem in NP is intractable (we refer to the problem as intractable if no polynomial time algorithm can possibly solve it), then so are all NP-complete problems. Presently, there is a large collection of problems considered to be intractable. We usually use the reduction method solving the NP-complete proof problem. Consider a decision problem, say A, which we would like to solve in polynomial time. Suppose we have another different decision problem, say B, that we already know how to solve in polynomial time. Finally, suppose a procedure exists that can transform any instance of A into some instance of B with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for instance of A is 'yes' if and only if the answer for instance of B is also 'yes'.

Use all these information together, by reducing solving problem A to solving problem B, we use the 'easiness' of B to prove the 'easiness' of A as long as the step involved take polynomial time. Likewise, suppose that we already know that there is no polynomial time algorithm for A, and a simple contradiction proof can show that there is

no polynomial time algorithm can exist for B. For NP-complete proof, this is the major method used to find new NPC problems.

The GENEPART algorithm is an optimization problem, we can certainly describe the related decision problem as follows in favor of the later NP-complete proof:

GENEPART

Instance: a n-by-m matrix, A, positive integers, k, s

Question: does there exist a partition p, and gene subset G with size s, which has at most k color changes

Note that the gene subset G constitute an s-by-m submatrix with order preserving row which means that this gene has smallest number of color changes given the current partition permutation. For example, such order preserving gene has the single color change property under 2 partition condition, that is, in a binary array, all the 0's stay together at the one end of the array and all the 1's stay together at the other end, no mixed up.

In the following we prove that GENEPART problem is intractable, that is, it belongs to the NP-complete class. Because of its similarity to the balanced complete bipartite subgragh problem, we can construct our proof using deduction from it which is already well known as NP-complete problem in Gery and Johnson's book[15]..

Balanced Complete Bipartite Subgraph

Instance: Bipartite Graph $G = (V, U, E)$, positive integer $k < |V| + |U|$

Question : can we find two disjoint subsets $X \subseteq V, Y \subseteq U$ such that $|X|=|Y|=k$, and for all $x \in X$, $y \in Y$,edge $(x, y) \in E$.

This problem was proved to be NP-complete using transformation from CLIQUE. Now, we prove the following theorem:

Theorem 2: GENEPART is NP-complete.

Proof:

It's straightforward to see that GENEPART is in NP, we can just use one full partition solution as the certificate, our proof focus on the NP-hard part.

Given a bipartite graph $G = (V, U, E)$, define the matrix A as follows: if $e_x(v_i, u_j) \in E$, then $A_{x,i} = -1, A_{x,j} = 1$, x is the label of the edge, all the other entries of A are set to 0.

We show now that G contains a balanced complete bipartite graph of size k if and only if the matrix A contains an order preserving submatrix B of size k^2-by-m. The theorem follows.

The first direction of the proof follows by construction, if matrix A has a order preserving submatrix B of size k^2-by-m , this matrix corresponds to a complete bipartite subgraph in G.

The second direction is to show that the instance of balanced complete bipartite subgraph can transform to the instance of GENEPART, if G contains a balanced complete subgraph of size k, then at the same indices we have an order preserving

submatrix B of size k^2-by-m, each edge crossing the two subset X,Y constructs a row of B that has the smallest number of color changes, say c. To see this, note that the entries belongs to two different partition are separated and placed at the two different ends after this row get sorted, and the entries reflecting those edges which connect the vertices in the same subset are all 0, since there are at least two partitions so that this row contradict the order preserving property, the number of color changes of the row corresponding to this kind of edges is at least c+1. This concludes the proof.

CHAPTER 5

EXPERIMENTAL RESULTS

Test Data Sets

We need some test instances to investigate the properties of our proposed algorithm and testing the performance. Normally, two possible approaches can be used to come up with test instances: First, we can start to test on data from real-world cases. However, even if an algorithm performs well on some instances, it does not guarantee that it will perform well on other instances, and as we mentioned before, microarrays are not typically task-specific and most of the features are not necessarily related to the phenotype of interest, and even the same set of samples may also display gender, age, or other disease variability, which may also serve as partitioning criteria. Thus a second approach is to use the test instances which have been studied well and the correct samples partition and discriminating gene set are already provide by previous research. We can evaluate our algorithm's performance according to its convergence rate to the known result.

We use two data sets to perform the algorithm test:

- BRCA dataset, 2001 [9]: this dataset is based on the research of NHGRI (National Human Genome Research Institution). It contains 3226 genes, 15 tissue samples, including BRCA1 Mutation Positive and BRCA2 Mutation

Positive breast cancers. This data set is will studied, the tissue sample partition and the list of discriminating gene set are provided.

- Notterman Adenoma Data set [18]: this data set is based on the research of gene expression project of Princeton University. It contains 7086 genes for 8 tissue sample data, 4 tumor tissues, 4 normal ones.

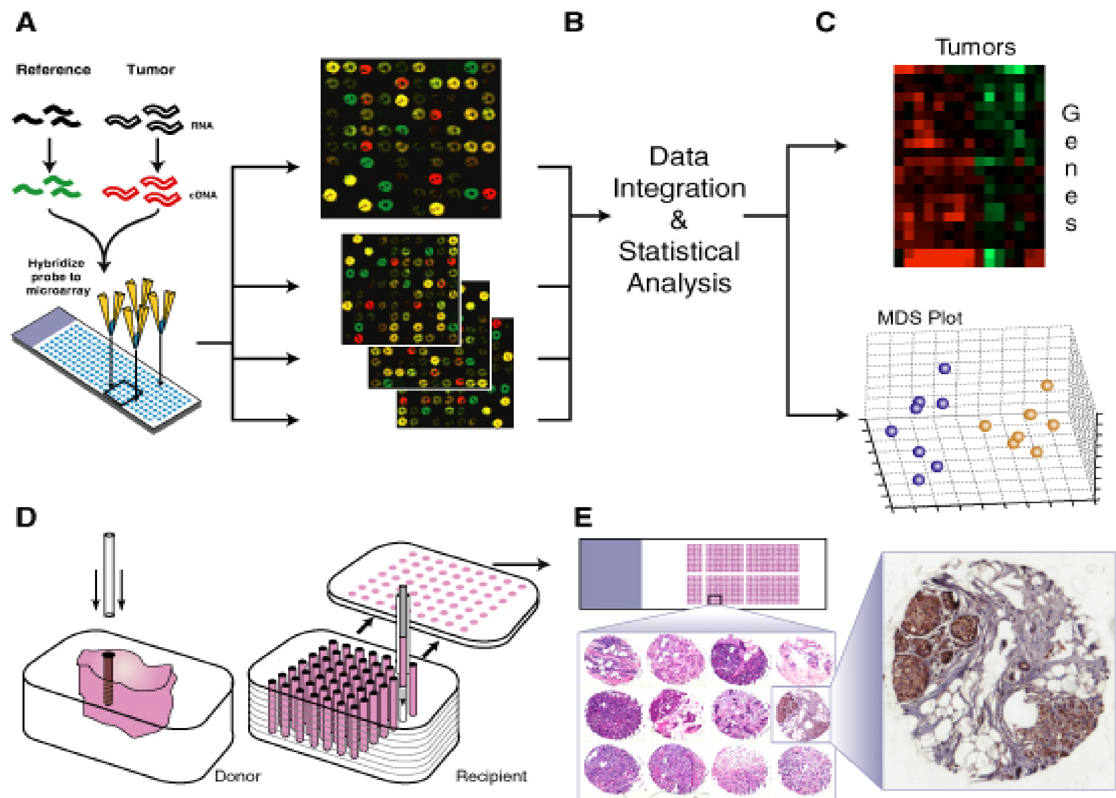

Figure 10 Overview of cDNA Microarray and Tissue Microarray Procedure.[9]

Description of The Experiment Source Data File

We use three types of file as the input data source of our algorithm: data file, gene info file, and control sample set file (optional). See Appendix B for file example.

Contained in the data file are the gene expression ratios from several microarray experiments. The format of the file is:

- Tab-delimited text file
- The first row provides the number of genes included in the dataset.
- The second row provides the number of tissue samples included in the dataset.
- The following rows contain gene expression ratio for each gene in each experiment.

Gene expression ratios included in the data file were derived from the fluorescent intensity (proportional to the gene expression level) from a tumor sample divided by the fluorescent intensity from a common reference sample (MCF-10A cell line). The common reference sample is used for all microarray experiments. Therefore, the ratio may take value from 0 to infinity. (There is no negative value in the data table). Ratios, included in the downloadable data file, for each experiment was normalized (or calibrated) such that the majority of the gene expression ratios from a pre-selected internal control gene set was around 1.0.

These genes were selected based on following criterions:

- Average fluorescent intensity (level of expression) of more than 2,500 (gray level) across all samples,
- Average spot area of more than 40 pixels across all samples
- No more than one sample in which the spot area is zero pixel. There are total of genes satisfy these requirements and thus included in the data file.

The gene info file provides the necessary information for the result display and thus enables us to investigate the detail of the selected gene and calculation the convergence rate for the selected gene set. Contained in the gene info file are the description for the all the information related with the genes. The format of the file is:

- The first row provides the number of genes included in the dataset.
- The second row provides the number of tissue samples included in the dataset.
- The following rows provide the gene information for each gene, The first column is the microtiter plate ID where each clone physically locates. The second column is the IMAGE Clone ID, which can be used to perform database lookup. The third column is the Clone Title.

The control sample set file is optional for our algorithm, it provide the grouped information for the samples which are known to be in the same partition (e,g,. normal/health tissue samples), and the remaining samples can not be partitioned into this

control group by our algorithm. This file provided a supervised method to improve the accuracy and the performance of our algorithm. The format of the file is:

- The first row provides the number of control sample partitions.
- Then there follows a row provides the size of the first control partition, and following rows provide the ID of each sample in this control partition. Again there will be a row showing the size of next control partition, followed by ID rows until all the control partitions with their content listed.
- After all the control sample partition information provided, a new row will provide the number of partitions that should be separated. Same as above, a size row followed by sample ID rows until done.

BRCA Data Set Test

The RNA used in this data set was derived from samples of primary tumors from seven carriers of the *BRCA1* mutation, seven carriers of the *BRCA2* mutation, and was compared with a microarray of 6512 complementary DNA clones of 5361 genes. Statistical analyses were used to identify a set of genes that could distinguish the *BRCA1* genotype from the *BRCA2* genotype.

Inheritance of a mutant *BRCA1* or *BRCA2* gene (numbers 113705 and 600185, respectively, in Online Mendelian Inheritance in Man, a catalogue of inherited diseases) confers a lifetime risk of breast cancer of 50 to 85 percent and a lifetime risk of ovarian cancer of 15 to 45 percent. These germ-line mutations account for a substantial proportion of inherited breast and ovarian cancers, but it is likely that additional susceptibility genes will be discovered. Certain pathological features can help to distinguish breast tumors with *BRCA1* mutations from those with *BRCA2* mutations. Tumors with *BRCA1* mutations are high-grade cancers with a high mitotic index, “pushing” tumor margins (i.e., non-infiltrating, smooth edges), and a lymphocytic infiltrate, whereas tumors with *BRCA2* mutations are heterogeneous, are often relatively high grade, and display substantially less tubule formation. The proportion of the perimeter with continuous pushing margins can distinguish both types of tumors from sporadic cases of breast cancer. Here in our experiment, we only pick up the BRCA1 and

BRCA2 patients' tissue sample for the input data set to our algorithm. We did the 2-parition and 3-partition experiments, and found some interesting results on them.

Table 1 2-partition Breast Cancer data set result

Maximum number of search nodes	Selected gene pool size	Number of sample partition	Minimum partition size	Examined Search nodes	Total color changes
5000000	176	2	7	7151	936
Partition 1			sample 0: BRCA1-s1996 sample 5: BRCA1-s1510 sample 6: BRCA1-s1905 sample 11: BRCA2-s1816 sample 12: BRCA2-s1616 sample 13: BRCA2-s1063 sample 14: BRCA2-s1936		
Partition 2			sample 1: BRCA1-s1822 sample 2: BRCA1-s1714 sample 3: BRCA1-s1224 sample 4: BRCA1-s1252 sample 7: BRCA2-s1900 sample 8: BRCA2-s1787 sample 9: BRCA2-s1721 sample 10: BRCA2-s1486		

From the above results, we can see that the partition mixed the BRCA1 and BRCA2 tissue samples together. Thus the algorithm was unable to classify the two types of breast cancer accordingly using the 2-partition configuration. However, when we execute the algorithm under 3-partition case (Table 3, Figure 11), the BRCA1 and BRAC2 tissue samples were perfectly separated, and BRCA2 samples were interestingly clustered into 2 sub-groups which are the possible indication of the different develop stagy or different

sample sub-category. What's more, none of the top 40 genes produced by the GENEPART under 2-partition configuration match the discrimating gene list provided by GHGRI [Appendix C], whereas, 60% of the top 40 genes produced by the GENEPART using 3-partition configuration [Appendix C] can be found in the list.

Table 2 3-partition Breast Cancer data set result

Maximum number of search nodes	Selected gene pool size	Number of sample partition	Minimum partition size	Examined Search nodes	Total color changes
5000000	176	3	4	196356	1056

Partition 1	sample 0: BRCA1-s1996 sample 1: BRCA1-s1822 sample 2: BRCA1-s1714 sample 3: BRCA1-s1224 sample 4: BRCA1-s1252 sample 5: BRCA1-s1510 sample 6: BRCA1-s1905
Partition 2	sample 7: BRCA2-s1900 sample 8: BRCA2-s1787 sample 9: BRCA2-s1721 sample 10: BRCA2-s1486
Partition 3	sample 11: BRCA2-s1816 sample 12: BRCA2-s1616 sample 13: BRCA2-s1063 sample 14: BRCA2-s1936

Adenoma Data Set Test

In this data set, colon adenocarcinoma specimens (snap-frozen in liquid nitrogen within 20 min of removal) were collected from patients. From some of these patients, paired normal colon tissue also was obtained.

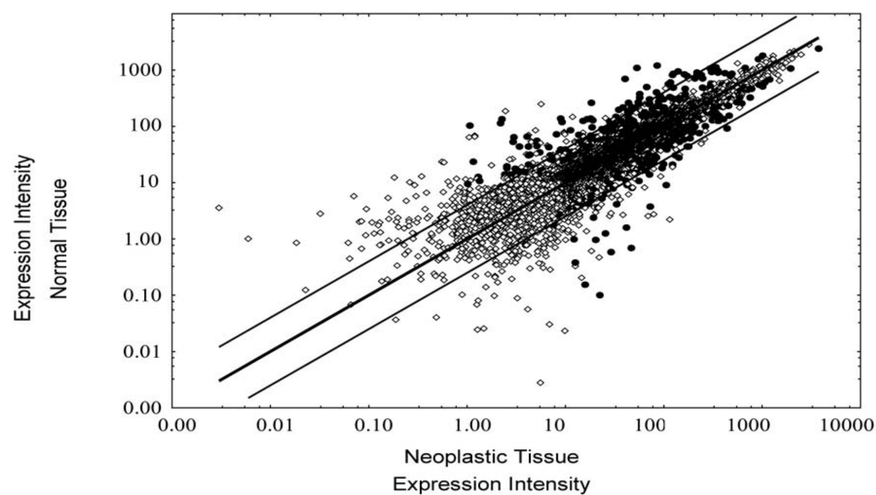

Figure 11 Expression intensity in normal compared to tumor samples

Colon adenocarcinoma samples and paired normal samples were hybridized to GeneChips, (Affymetrix) and the expression levels were analysisd with GeneChip 3.0 analysis software (Affymetrix). The upper and lower boundaries represent a 4-fold difference in the average of each gene's expression between adenoma and normal tissue. Closed circles denote genes for which average expression in adenoma was significantly higher or lower than it was in the matched normal sample ($p<0.001$). Approximately 2

% of the 4000 genes analysisd for this figure displayed a statistically significant, 4-fold difference in expression intensity between tumor and matched normal samples.

Table 3 Adenoma result

Maximum number of search nodes	Selected gene pool size	Number of sample partition	Minimum partition size	Examined Search nodes	Total color changes
50000	200	2	4	181	400
Partition 1			sample 0: Adenoma 1 sample 1 Adenoma 2 sample 2: Adenoma 3 sample 3: Adenoma 4		
Partition 2			sample 7: Normal(Ad) 1 sample 8: Normal(Ad) 2 sample 9: Normal(Ad) 3 sample 10: Normal(Ad)4		

We can see from Table 3 that GENEPART can perfectly separate the tumor tissues and the health ones, and can very quickly find the optimal partition/gene set pair, only examed 181 search nodes, revealing broad coherent patterns that suggest a high degree of organization underlying gene expression in these tissues. Of the top10 genes selected by GENEPART [Appendix C], 5 are Human ribosomal proteins whose intersity are known relatively low in the normal colon tissues and high in the colon tumor tissues [18].

CHAPTER 6

CONCLUSIONS

We have studied a new combinatorial approach to the problem of feature selection and clustering of microarray data. Based on the color change method and the branch and bound algorithm, we are able to compute the optimal sample partition and interesting gene set. Provided with high-dimensional feature space and sparse sample number, our algorithm can find the partitioning with considerable accuracy: perfectly cluster the BRCA data set into groups separating BRCA1 and BRCA2 in 3-partition experiment and classify the Adenoma data set into correct tumor and normal sample groups. Also, it is important to note that, and the sparsity of the data, the high dimensionality of the feature space, and the fact that many feature are irrelevant or redundant cause the following problems:

- A clustering algorithm is not guaranteed to capture a ‘meaningful’ partition corresponding to some phenotypes of actual empirical interest, such having or not having a particular type of tumor, because the same set of samples may also display gender, age, or other disease variability, which may also serve as partitioning criteria.
- Microarrays are not typically task-specific and most of the features are not necessarily related to the phenotype of interest. Thus, even when the

phenotype of interest, such as tumor type, induces a strong discriminating pattern in the feature space, the distance calculation between samples is still subject to interference from the large number of irrelevant feature.

- The goal of clustering is often not merely to find out the underlying grouping of samples, but also to form some generalizable cluster representations and samples recognition rules so that future novel samples can be correctly labeled.

In summary, our results suggest that the algorithm, with its use of color change method, is capable of capturing the partition that characterizes the samples but is masked in the original high-dimensional feature space. Not only can hidden biologically meaningful partitions of the sample set be identified in this way, but also the selected features are of significant interest because they represent a set of causal factors that elicit such partitions. This information can be useful to the modeling process for the further research of the connecting between biological/clinical phenotypes and lab qualitative and statistics.

REFERENCES CITED

[1] A. H. Land and A. G. Doig, *An Automatic Method for Solving Discrete Programming Problems,* Econometrica, Vol.28, 1960, pp. 497-520

[2] A. L. Lehninger. *Biochemistry*. Worth Publishers, Inc., 1975.

[3] Alter, O., P. Brown, and D. Botstein (2000). Singular value decomposition for genome-wide expression data processing and modeling. Proc Natl Acad Sci. USA 97, 10101–10106.

[4] B. Alberts, D. Bray, J. Lewis, M. Ra., K. Roberts, and J. D.Watson. Molecular Biology Of The Cell. Garland Publishing, Inc., 1994.

[5] Ben-Dor, A.., B. Chor, R. Karp, Z. Yakhini. 2002. Discovering local structure in gene expression data: the order-preserving submatrix problem. Proceedings of the sixth annual international conference on Computational biology.

[6] Ben-Dor, A. N. Friedman, Z.Yakhini. 2001. Class discovery in gene expression data.Proceedings of the fifth annual international conference on Computational biology.

[7] Hassler Whitney. On the abstract properties of linear dependence. American Journal of Mathematics,57:509-533,1935

[8] Hastie, T., R. Tibshirani, M. Eisen, P. Brown, D. Ross, U. Scherf, J. Weinstein, A.Alizadeh, L. Staudt, and D. Botstein (2000). Gene shaving: a new class of clustering methods for expression arrays. In Tech. report, Stanford University.

[9] Ingrid H., David D,.etc. Gene-Expression Profiles in Hereditary Breast Cancer, The New Engliand Journal of Medicine, Vol, 344, Feb. 22. 2001

[10] Kari, L., A. Loboda, M. Nebozhyn, A.H. Rook, E.C.Vonderheid , C. Nichols, D. Virok, C. Chang, W.-H. Horng, J. Johnston, M. Wysocka, M.K. Showe, and L.C. Showe.*Classification and Prediction of Survival in Patients with the Leukemic Phase of Cutaneous T-Cell Lymphoma.* J. Exp Med 197: 1477-1488.

[11] Katta G. Murty, *Operations Research: Deterministic Optimization Models,* Prentice Hall, 1994, Chapter 9

[12] *K. G. Murty, C. Karel, and J. D. C. Little, Case Institute of Technology: The Traveling Salesman Problem: Solution by a Method of Ranking ssignments,1962*

[13] Leming, Shi, DNA Microarrays, www.gene-chips.com

[14] Mumey,B, 2004. A Combinatorial Approach to Clustering Gene Expression Data. Submitted to RECOMB 2005

[15] M.R. Garey and D.S. Johnson. COMPUTERS AND INTRACTABILITY, *A Guide to the Theory of NP-Completeness*, page 196.Freeman, 1979.

[16] Shi, J. and J. Malik (2000). Normalized cuts and image segmentation.*IEEE Transactions on Pattern Analysis and Machine Intelligence22(8)*, 888–905.

[17] Sharan, R., A. Maron-Katz, R. Shamir. 2003. *CLICK and EXPANDER: A System for Clustering and Visualizing Gene Expression Data.* Bioinformatics Vol. 19 No. 14 pp. 1787-1799.

[18] U. Alon, N. Barkai, D.,… *Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. Proceeding of the national academic of science of USA*

[19] *Xing, E. Karp, R. 2001. CLIFF: clustering of high dimensional microarray data via iterative feature filtering using normalized cuts. Bioinformatics, 17, 306-315.*

[20] Xing, E., M. Jordan, and R. Karp (2001). Feature selection for high-dimensional genomic microarray data. In *the Eighteenth International Conference on Machine Learning, in press*.

APPENDICES

APPENDIX A

GENEPART ALGORITHM SOURCE CODE

genepart.h

```
#include <string>
#include <map>
#include <list>
#include <math.h>

const int maxPartitions = 10;
const int maxSamples = 30;
//const double minFoldChange = 2.5;
//const double foldPenalty = 100.0;
const int maxNameLength = 500;
const char* colorName[] = {"green", "red", "navy", "olive", "teal"};
const int numColors = 5;


// prototypes
double log(double x);
double fabs(double x);
long combN(int k, char size[]);
long comb(int n, int k);
long count2color(int i, int j, int c);
long countNcolor(int k, char size[],    int c);



#define max2(a, b) ((a < b) ? b : a)
#define min2(a, b) ((a < b) ? a : b)
#define max3(a, b, c) max2(a, (max2(b, c)))
#define max4(a, b, c, d) max2(a, (max3(b, c, d)))


class GeneralError {
  private:
    string _errText;
  public:
    GeneralError(const string& errText) {_errText = errText;}
    void print() {
        cerr << _errText << endl;
    }
};
```

```
class GeneMatrix {
  public:
    double* m;
    int numRows, numCols;

    GeneMatrix(int nRows, int nCols) {
        numRows = nRows;
        numCols = nCols;
        m = (double*) malloc(sizeof(double) * numRows * numCols);
    }
    ~GeneMatrix() {
        free(m);
    }
    void set(int i, int j, double b) {
        m[i * numCols + j] = b;
    }
    double get(int i, int j) {
        return m[i * numCols + j];
    }
    void print();
};


class SearchNode;

class GenePartInfo {
  public:
    int maxSearchNodes;
    int genePoolSize;
    int numSampleParts;
    int minPartitionSize;
    int numGenes;
    int numSamples;
    GeneMatrix* gm;

    ofstream resultsFile;

    // gene information
    char** geneNames;
    int** sortPerm;
    int* bwChanges;
    int* biPart;
    int* colorChanges;
```

```
    double** mean;
    int* scoreSort;
    double* score;

    // sample information
    char** sampleNames;

    SearchNode* sNodes;
    SearchNode* bestNode;
    int** compatMatrix;

    GenePartInfo(int argc, char** argv);
    //    void findSizes(SearchNode* n, int* size);
    void findColorChanges(int g, char* part);
    void findMeans(int g, char* part, char* size);
    void readMatrixFile(string& matFileName);
    void readCompatFile(string& compatFileName);
    void readNamesFile(string& namesFileName);
    void initGeneSet();
    void refinePartition();
    void refineGeneSet(SearchNode* n);
    void outputResults(SearchNode* n);
    void run();
};


class SearchNode {
  public:
    SearchNode* parent;
    char partition[maxSamples];
    char lastPartUsed;
    char size[maxPartitions];
    char sampleIndex;
    //    int setNumber;
    int cumulChanges;

    bool initialize(int sample, int set, int maxP, SearchNode* par, GenePartInfo* gpi,
                    int lowestChangesFound, double* expectedChanges);
    int lowerBoundChanges(GenePartInfo* gpi);
};


// iterators:
typedef multimap<double, SearchNode*>::iterator MMDSPI;
```

genepart.cpp

```
#include <fstream>
#include <cstdlib>
#include <stdlib.h>
#include <stdio.h>
#include <vector>

#include "genepart.h"

long numSearchNodes = 0;



int partition(int* sortPerm, double* row, int p, int r) {
  double pivot = row[sortPerm[r]];
  int i = p - 1;
  for (int j = p; j < r; j++) {
    if (row[sortPerm[j]] <= pivot) {
      i++;
      int t = sortPerm[i];
      sortPerm[i] = sortPerm[j];
      sortPerm[j] = t;
    }
```

```
    }
    int t = sortPerm[i+1];
    sortPerm[i+1] = sortPerm[r];
    sortPerm[r] = t;
    return i+1;
}


void quicksort(int* sortPerm, double* row, int p, int r) {
    if (p < r) {
        int q = partition(sortPerm, row, p, r);
        quicksort(sortPerm, row, p, q-1);
        quicksort(sortPerm, row, q+1, r);
    }
}


void quickSelect(int* sortPerm, double* row, int p, int r, int max) {
    if (p < r) {
        int q = partition(sortPerm, row, p, r);
        quicksort(sortPerm, row, p, q-1);
        if (q < max)
            quicksort(sortPerm, row, q+1, r);
    }
}
```

```
bool SearchNode::initialize(int sample, int part, int lastP,
                            SearchNode* par, GenePartInfo* gpi,
                            int lowestChangesFound, double* expectedChanges) {
  numSearchNodes++;

  sampleIndex = sample;
  lastPartUsed = lastP;
  parent = par;

  for (int i = 0; i < gpi->numSamples; i++)
    if (i < sampleIndex)
      partition[i] = parent->partition[i];
    else
      partition[i] = -1;
  partition[sampleIndex] = part;

  // check for grouping pre-specified dependencies
  for (int i = 0; i < sampleIndex; i++) {
    switch (gpi->compatMatrix[sampleIndex][i]) {
    case 1: // same cluster requirement
      if (partition[sampleIndex] != partition[i])
   return false;
      break;
    case 2: // different cluster requirement
      if (partition[sampleIndex] == partition[i])
```

```
  return false;
      break;
    }
}

for (int i = 0; i < gpi->numSampleParts; i++)
    size[i] = 0;
if (par)
    for (int i = 0; i <= par->lastPartUsed; i++)
        size[i] = parent->size[i];
size[part]++;

int sum = 0;
for (int i = 0; i < gpi->numSampleParts; i++)
    if (size[i] < gpi->minPartitionSize)
        sum += gpi->minPartitionSize - size[i];
int remainingSamples = gpi->numSamples - sampleIndex - 1;
if (sum > remainingSamples)
    return false;

gpi->refineGeneSet(this);
int numUnusedParts = gpi->numSampleParts - (lastPartUsed + 1);
*expectedChanges = (double) (cumulChanges + numUnusedParts) / (sample + 1.0);

if (lowestChangesFound != -1 &&
        (cumulChanges + (gpi->numSampleParts - (sampleIndex + 1)))
```

```
            > lowestChangesFound)
        return false;

    return true;
}



void GenePartInfo::refinePartition() {
    multimap<double, SearchNode*> frontier;
    int node = 0, lowestChangesFound = -1;;
    double e;

    sNodes[node].initialize(0, 0, 0, NULL, this, lowestChangesFound, &e);
    frontier.insert(make_pair(e, &sNodes[node]));
    while (frontier.size() > 0) {
        MMDSPI f = frontier.begin();
        SearchNode* curNode = f->second;
        frontier.erase(f);
        int maxP = curNode->lastPartUsed;
        if (curNode->sampleIndex < numSamples - 1) {
            // at internal search tree node:
            for (int i = 0; i < min2(maxP + 2, numSampleParts); i++) {
      if (node >= maxSearchNodes - 1) {
          resultsFile << "Out of searchnodes, optimality not guaranteed.<br>" << endl;
          return;
      }
```

```
        if (sNodes[++node].initialize(curNode->sampleIndex + 1, i,
                                      max2(i,maxP), curNode, this,
                                      lowestChangesFound, &e))
           frontier.insert(make_pair(e, &sNodes[node]));
        else
           node--;
            }
         } else { // curNode is a leaf:
            if ((lowestChangesFound == -1) ||
           (curNode->cumulChanges < lowestChangesFound)) {
        lowestChangesFound = curNode->cumulChanges;
        bestNode = curNode;
            }
         }
      }
   }

void GenePartInfo::findColorChanges(int g, char* part) {
   // counts the number of color changes for gene g with current partition
   int *sortPermG = sortPerm[g];
   int count = 0;
   int partCount[numSampleParts];
   for (int i = 0; i < numSampleParts; i++)
      partCount[i] = 0;
```

```
    int i = 0;
    while (i < numSamples && part[sortPermG[i]] == -1)
      i++;
    while (i < numSamples) {
      char prevPart = part[sortPermG[i]];
      i++;
      while (i < numSamples && part[sortPermG[i]] == -1)
        i++;
      if (i < numSamples && part[sortPermG[i]] != prevPart) {
        count++;
        partCount[prevPart]++;
        partCount[part[sortPermG[i]]]++;
      }
    }
    colorChanges[g] = count;

    bwChanges[g] = partCount[0];
    biPart[g] = 0;
    for (int i = 1; i < numSampleParts; i++)
      if (partCount[i] < bwChanges[g]) {
        bwChanges[g] = partCount[i];
        biPart[g] = i;
      }
  }
```

```
void GenePartInfo::findMeans(int g, char* part, char* size) {
   // computes means of each group
   int *sortPermG = sortPerm[g];
   for (int i = 0; i < numSampleParts; i++)
      mean[g][i] = 0.0;

   for (int i = 0; i < numSamples; i++)
      if (part[i] != -1)
         mean[g][part[i]] += gm->get(g, i);

   for (int i = 0; i < numSampleParts; i++)
      if (size[i] > 0)
         mean[g][i] /= size[i];
}


void GenePartInfo::refineGeneSet(SearchNode* n) {
   for (int g = 0; g < numGenes; g++) {
      findColorChanges(g, n->partition);
      //       findMeans(g, n->partition, n->size);
      score[g] = bwChanges[g] + colorChanges[g];
   }
   quickSelect(scoreSort, score, 0, numGenes - 1, genePoolSize);

   n->cumulChanges = 0;
   for (int i = 0; i < genePoolSize; i++)
```

```
        n->cumulChanges +=    bwChanges[scoreSort[i]]
            + colorChanges[scoreSort[i]];
}



void GenePartInfo::initGeneSet() {
    double* row;
    row = (double*) malloc(sizeof(double) * numSamples);
    sortPerm = (int**) malloc(sizeof(int*) * numGenes);
    for (int i = 0; i < numGenes; i++) {
        sortPerm[i] = (int*) malloc(sizeof(int) * numSamples);
        for (int j = 0; j < numSamples; j++) {
            row[j] = gm->get(i, j);
            sortPerm[i][j] = j;
        }
        // sort sample values
        quicksort(sortPerm[i], row, 0, numSamples-1);
    }

    bwChanges = (int*) malloc(sizeof(int) * numGenes);
    biPart = (int*) malloc(sizeof(int) * numGenes);
    colorChanges = (int*) malloc(sizeof(int) * numGenes);
    score = (double*) malloc(sizeof(double) * numGenes);
    scoreSort = (int*) malloc(sizeof(int) * numGenes);
    for (int i = 0; i < numGenes; i++)
        scoreSort[i] = i;
```

```
}


void GenePartInfo::readMatrixFile(string& matFileName) {
   ifstream matFile(matFileName.c_str());
   if (! matFile)
      throw GeneralError("Unable to open " + matFileName);

   matFile >> numGenes;
   matFile >> numSamples;

   gm = new GeneMatrix(numGenes, numSamples);
   double value;
   for (int i = 0; i < numGenes; i++)
      for (int j = 0; j < numSamples; j++) {
           matFile >> value;
           gm->set(i, j, value);
      }
   matFile.close();
}


void GenePartInfo::readNamesFile(string& namesFileName) {
   ifstream namesFile(namesFileName.c_str());
   if (! namesFile)
      throw GeneralError("Unable to open " + namesFileName);
```

```
int nG, nS;
namesFile >> nG;
namesFile >> nS;
if (nG != numGenes)
    throw GeneralError("Number of genes does not agree in names file");
if (nS != numSamples)
    throw GeneralError("Number of samples does not agree in names file");

geneNames = (char**) malloc(sizeof(char*) * numGenes);
sampleNames = (char**) malloc(sizeof(char*) * numSamples);
char x[10];
namesFile.getline(x, 10);
for (int i = 0; i < numGenes; i++) {
    geneNames[i] = (char*) malloc(sizeof(char) * maxNameLength);
    namesFile.getline(geneNames[i], maxNameLength);
    //      cout << geneNames[i] << endl;
}
for (int i = 0; i < numSamples; i++) {
    sampleNames[i] = (char*) malloc(sizeof(char) * maxNameLength);
    namesFile.getline(sampleNames[i], maxNameLength);
}
namesFile.close();
resultsFile << "last gene name: " << geneNames[numGenes - 1] << "<br>" << endl;
resultsFile << "last sample name: " << sampleNames[numSamples - 1] << "<br>" <<
endl;
```

```
}


void GenePartInfo::readCompatFile(string& compatFileName) {
   ifstream compatFile(compatFileName.c_str());
   if (! compatFile)
      throw GeneralError("Unable to open " + compatFileName);

   int numSameGp, numDiffGp, gpSize, gpSize2;
   int sample;
   int group[numSamples];
   int group2[numSamples];

   compatFile >> numSameGp;
   for (int i = 0; i < numSameGp; i++) {
      compatFile >> gpSize;
      for (int j = 0; j < gpSize; j++) {
         compatFile >> sample;
         group[j] = sample;
      }
      for (int j = 0; j < gpSize; j++)
         for (int k = 0; k < gpSize; k++)
     compatMatrix[group[j]][group[k]] =
        compatMatrix[group[k]][group[j]] = 1;
   }
   compatFile >> numDiffGp;
```

```
    for (int i = 0; i < numDiffGp; i++) {
        compatFile >> gpSize;
        for (int j = 0; j < gpSize; j++) {
            compatFile >> sample;
            group[j] = sample;
        }
        compatFile >> gpSize2;
        for (int j = 0; j < gpSize2; j++) {
            compatFile >> sample;
            group2[j] = sample;
        }
        for (int j = 0; j < gpSize; j++)
            for (int k = 0; k < gpSize2; k++)
      compatMatrix[group[j]][group2[k]] =
          compatMatrix[group2[k]][group[j]] = 2;
    }
    compatFile.close();
    /*
    for (int i = 0; i < numSamples; i++) {
        for (int j = 0; j < numSamples; j++) {
            cout <<    " " << compatMatrix[i][j];
        }
        cout << endl;
    }
    */
}
```

```
GenePartInfo::GenePartInfo(int argc, char** argv) {
   maxSearchNodes = atoi(argv[1]);
   genePoolSize = atoi(argv[2]);
   numSampleParts = atoi(argv[3]);
   minPartitionSize = atoi(argv[4]);
   char* filePrefix = argv[5];
   string fn;

   fn = string(filePrefix) + "-mat.txt";
   readMatrixFile(fn);
   fn = string(filePrefix) + "-names.txt";
   readNamesFile(fn);
   fn = string(filePrefix) + "-results.htm";

   resultsFile.open(fn.c_str());

   resultsFile << "<b>GENEPART version 0.8 RESULTS</b><br>" << endl;
   resultsFile << "Maximum number of search nodes: " << maxSearchNodes << "<br>"
<< endl;
   resultsFile << "Selected gene pool size: " << genePoolSize << "<br>" << endl;
   resultsFile << "Number of sample partitions: " << numSampleParts << "<br>" <<
endl;
   resultsFile << "Minimum partition size: " << minPartitionSize << "<br>" << endl;
   resultsFile << "File prefix: " << filePrefix << "<br>" << endl;
```

```
if (numSamples / numSampleParts < minPartitionSize)
   throw GeneralError("Minimum partition size too large.");

compatMatrix = (int**) malloc(sizeof(int*) * numSamples);
for (int i = 0; i < numSamples; i++) {
   compatMatrix[i] = (int*) malloc(sizeof(int) * numSamples);
   for (int j = 0; j < numSamples; j++)
      compatMatrix[i][j] = 0; // compatible by default
}

mean = (double**) malloc(sizeof(double*) * numGenes);
for (int i = 0; i < numGenes; i++)
   mean[i] = (double*) malloc(sizeof(double) * numSampleParts);

if (argc == 7) {
   char* compatFileName = argv[6];
   resultsFile << "Compatability file: " << compatFileName << "<br>" << endl;
   fn = string(compatFileName);
   readCompatFile(fn);
}

sNodes = (SearchNode*) malloc(sizeof(SearchNode) * maxSearchNodes);
if (! sNodes)
   throw GeneralError("Unable to allocate search node array.");
bestNode = NULL;
```

```
}


void GenePartInfo::outputResults(SearchNode* n) {
   double biProb[numSampleParts][numSamples];
   double allProb[numSamples];
   long totalPerms;
   char* sz = n->size;
   for (int i = 0; i < numSampleParts; i++) {
      totalPerms = comb(numSamples, sz[i]);
      for (int c = 1; c < 2 * min2(sz[i], numSamples - sz[i]); c++) {
         biProb[i][c] = (double) count2color(sz[i], numSamples-sz[i], c) / totalPerms;
      }
   }
   totalPerms = combN(numSampleParts, sz);
   for (int c = 1; c < numSamples - 1; c++) {
      allProb[c] = (double) countNcolor(numSampleParts, sz, c) / totalPerms;
   }

   double foldChange[genePoolSize];
   for (int i = 0; i < genePoolSize; i++) {
      int g = scoreSort[i];
      findMeans(g, n->partition, n->size);
      int min = 0, max = 0;
      for (int j = 1; j < numSampleParts; j++)
         if (n->size[j] > 0) {
```

```
  if (mean[g][j] < mean[g][min])
     min = j;
  if (mean[g][j] > mean[g][max])
     max = j;
      }
   if (min != max &&
  n->size[min] >= minPartitionSize &&
  n->size[max] >= minPartitionSize &&
  mean[g][min] > 0.0)
      foldChange[i] = mean[g][max] / mean[g][min];
   else
      foldChange[i] = -1.0; // represents undefined
}

resultsFile << "Sample partitions:" << "<br>" << endl;
for (int i = 0; i < numSampleParts; i++) {
   resultsFile << "Partition " << i << ":" << "<br>" << endl;
   for (int j = 0; j < numSamples; j++)
      if (n->partition[j] == i) {
  resultsFile << " sample " << j << ": ";
  resultsFile << "<font color=" << colorName[i % numColors] << ">"
             << sampleNames[j] << "</font><br>" << endl;
      }
}

resultsFile.setf(ios::fixed);
```

```
    resultsFile.precision(2);
    resultsFile << "Genes selected:<br>" << endl;
    resultsFile << "<table border=1>" << endl;
    resultsFile << "<tr>"
                << "<th>Gene</th>"
                << "<th>Coloring</th>"
                << "<th>B&W changes</th>"
                << "<th>-log prob</th>"
                << "<th>Color changes</th>"
                << "<th>-log prob</th>";
    for (int c = 0; c < numSampleParts; c++)
        resultsFile << "<th><font color=" << colorName[c % numColors] <<
">Mean</font></th>";
    resultsFile << "<th>Fold change</th>";
    resultsFile << "</tr>" << endl;

    for (int i = 0; i < genePoolSize; i++) {
        int g = scoreSort[i];
        resultsFile << "<tr>";
        resultsFile << "<td>" << geneNames[g] << "</td>";

        resultsFile << "<td>";
        for (int j = 0; j < numSamples; j++) {
            resultsFile << "<font color="
                 << colorName[n->partition[sortPerm[g][j]] % numColors] << ">";
            resultsFile << " " << gm->get(g, sortPerm[g][j]);
```

```
        resultsFile <<"</font>";
        resultsFile << "<sup>" << sortPerm[g][j] << "</sup>";


      }
      resultsFile << "</td>";
      resultsFile << "<td>" << bwChanges[g] << "</td>";
      resultsFile << "<td>" << -log(biProb[biPart[g]][bwChanges[g]]) << "</td>";
      resultsFile << "<td>" << colorChanges[g] << "</td>";
      resultsFile << "<td>" << -log(allProb[colorChanges[g]]) << "</td>";
      for (int c = 0; c < numSampleParts; c++)
        resultsFile << "<td><font color=" << colorName[c % numColors] << ">"
              << mean[g][c] << "</font></td>";
      if (foldChange[i] >= 0.0)
        resultsFile << "<td>" << foldChange[i] << "</td>";
      else
        resultsFile << "<td>undef</td>";

      resultsFile << "</tr>" << endl;
    }
    resultsFile << "</table>" << endl;
    resultsFile.close();
}


void GenePartInfo::run() {
    initGeneSet();
```

```
   refinePartition();
   if (! bestNode)
      throw GeneralError("Unable to find a best node.");
   refineGeneSet(bestNode);
   resultsFile << "Examined " << numSearchNodes << " search nodes." << "<br>" <<
endl;
   resultsFile << "Total color changes: " << bestNode->cumulChanges << "<br>" <<
endl;
   outputResults(bestNode);
}


int main (int argc, char** argv) {
   try {
      if (argc < 6)
         throw GeneralError("Usage: genepart <max search nodes> <gene pool size>
<sample partitions> <min partition size> <file prefix> <optional: compat file");

      GenePartInfo* gpInfo = new GenePartInfo(argc, argv);
      gpInfo->run();

   } catch (GeneralError& e) {
      e.print();
      return 1;
   }
   return 0;
```

}

probcalc.cpp

```
// calculates a bound on the probability of observing a certain number of color changes in
a block matrix

#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#define min2(a, b) ((a < b) ? a : b)

long gcd(long a, long b) {
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

long combN(int k, char size[]) {
    long num = 1;
    long den = 1;
    int c = 1;
    long g;
    for (int i = 0; i < k; i++) {
        for (int j = 1; j <= size[i]; j++) {
            num *= c++;
            den *= j;
            int g = gcd(num, den);
            num /= g;
```

```
            den /= g;
        }
    }
    return num / den;
}

long comb(int n, int k) {
    char t[2];
    t[0] = n-k;
    t[1] = k;
    return combN(2, t);
}

long count2color(int i, int j, int c) {
    if (c < 0)
        return 0;
    if (c == 0) {
        if (i > 0 && j > 0)
            return 0;
        else if (i > 0 || j > 0)
            return 1;
        else
            return 0;
    }
    if (c == 1) {
        if (i <= 0 || j <= 0)
```

```
            return 0;
        else
            return 2;
    }

    switch (c % 2) {
    case 0:
        return
            comb(i-1, c/2) * comb(j-1, c/2-1) +
            comb(j-1, c/2) * comb(i-1, c/2-1);
    case 1:
        return
            2 * comb(i-1,(c-1)/2) * comb(j-1, (c-1)/2);
    }
}

long countNcolor(int k, char size[],    int c) {
    if (k < 2)
        return -1;
    if (k == 2)
        return count2color(size[0], size[1], c);

    int n = 0;
    for (int s = 0; s < k-1; s++)
        n += size[s];
```

```
    long sum = 0;
    for (int d = 1; d <= c-1; d++) {
        long cc = countNcolor(k-1, size, d);
        for (int e = 0; e <= c-1; e++) {
            if ((c - d - e) >= 0 &&
           (c - d - e) % 2 == 0) {
      int f = (c - d - e) / 2;
      sum += cc *
           comb(d + 2, e) *
           comb(n - size[k-1] - d - 1, f) *
           comb(size[k-1] - 1, e + f + 1);
            }
        }
    }
    return sum;
}

double blockProb(int numGenes, int numSelectedGenes, int numParts,
                char* partSize, int maxColorChange) {
    long num[10000], den[10000];
    int n = 0;
    int d = 0;
    for (int i = 0; i < numSelectedGenes; i++) {
        num[n++] = numGenes-i;
        den[d++] = i+1;
    }
```

```
int r = 1;
for (int i = 0; i < numParts; i++) {
    for (int j = 1; j <= partSize[i]; j++) {
        num[n++] = r++;
        den[d++] = j;
    }
}
long ccsum = 0;
for (int cc = numParts-1; cc <= maxColorChange; cc++)
    ccsum += countNcolor(numParts, partSize, cc);

long totalPerm = combN(numParts, partSize);

for (int i = 0; i < numSelectedGenes; i++) {
    num[n++] = ccsum;
    den[d++] = totalPerm;
}

for (int i = 0; i < n; i++)
    for (int j = 0; j < d; j++) {
        int g = gcd(num[i], den[j]);
        num[i] /= g;
        den[j] /= g;
    }
int i = 0;
int j = 0;
```

```
    double x = 0.0;
    while ((i < n) || (j < d)) {
        if (i < n)
            x += log((double) num[i++]);
        if (j < d)
            x -= log((double) den[j++]);
    }
    return x;
}
```

APPENDIX B

EXAMPLE OF INPUT FILES

```
3226
15
0.15  0.22  0.3   0.26  1.22  0.44  0.38  0.35  1.1   1.07  1.46  0.73  0.63  0.77  0.66
1.54  1.27  0.76  0.85  1.27  0.64  0.61  0.9   0.64  0.78  0.55  0.71  0.3   0.62  1
1.72  1.57  2.13  1.09  1.98  0.74  2.43  1.71  1.16  1.33  1.46  1.71  1.26  1.41  3
0.71  1.24  1.69  2.23  1.16  0.82  2.08  1.44  2.03  3.6   1.2   3.24  2.41  1.56  2.56
0.94  1.53  1.87  1.19  1.16  1.54  1.01  1.05  0.91  0.85  1.22  3.25  2.2   1.09  1.29
0.8   0.95  1.53  1.37  1.02  1.22  1.09  0.78  0.96  0.65  1.02  0.66  1.4   1.32  1.13
0.78  0.81  1.42  0.97  1.04  1.12  0.56  1.04  0.97  0.88  1.11  1.57  0.47  0.97  1.36
0.27  0.35  0.87  0.88  0.51  0.45  0.39  0.35  0.42  0.28  0.34  0.21  0.34  0.49  0.32
1.84  1.56  0.85  0.94  0.98  1.1   1.41  0.68  0.91  0.44  0.49  1.5   0.86  1.16  1.57
0.82  1.2   0.74  1.08  0.73  1.27  1.13  0.64  0.41  0.47  0.68  0.33  0.33  0.74  0.17
1.78  0.71  0.77  0.69  0.54  0.7   0.53  0.55  0.96  0.95  1.07  1.47  1.76  2.12  0.8
0.59  1.08  1.16  0.7   0.76  0.52  1.12  0.62  0.41  0.43  0.57  0.68  0.29  0.35  0.69
2.17  1.22  1.64  1.79  0.92  1.65  3.35  1.32  1.44  1.29  1.25  1.27  1.15  2.24  1.42
0.62  0.73  1.05  0.71  0.62  0.54  0.69  0.6   0.38  0.43  0.65  1.37  0.92  0.67  1.05
1.05  0.92  1.58  0.97  0.8   0.73  2.6   0.97  1.1   1.55  0.8   2.21  1.75  2.03  1.84
1.69  1.48  1.93  1.88  1.4   1.58  1.82  1.89  1.7   1.15  2.22  1.65  1.31  1.31  3.62
1.12  1.06  1.57  1.32  1.52  1.31  0.83  1.13  1.51  1.15  1.38  2.22  2.09  3.01  1.95
0.87  1.08  0.81  1.54  1.45  0.97  1.09  1.63  1.48  1.45  1.57  1.79  1.81  1.31  1.68
0.7   0.72  1.17  0.78  0.9   0.68  0.56  0.85  0.59  0.63  1.52  1.72  0.8   1.54  1.69
1.31  3.23  1.56  2.06  2.28  1.91  1.15  1.33  1.47  1.41  1.96  1.23  1.21  0.78  2.27
```

Figure 12 Expression data matrix file example

```
3226
15
HK1A1 21652 catenin (cadherin-associated protein), alpha 1 (102kD)
HK1A2 22012 ADP-ribosylation factor 3
HK1A4 22293 uroporphyrinogen III synthase (congenital erythropoietic porphyria)
HK1A5 22493 ribosomal protein L26
HK1A6 23019 guanine nucleotide binding protein (G protein), alpha stimulating activity polypep
HK1A7 23132 pre-mRNA splicing factor SF3a (120 kDa subunit), similar to S. cerevisiae PRP21
HK1A8 24145 adenylyl cyclase-associated protein
HK1A9 25584 ubiquinol-cytochrome c reductase core protein II
HK1A10      25725 farnesyl-diphosphate farnesyltransferase 1
HK1A11      26184 phosphofructokinase, platelet
HK1B1 26922 deoxyhypusine synthase
HK1B2 27549 heterogeneous nuclear ribonucleoprotein A1
HK1B3 27624 coatomer protein complex, subunit alpha
HK1B4 27848 acetyl-Coenzyme A acyltransferase 1 (peroxisomal 3-oxoacyl-Coenzyme A thiolase)
HK1B5 28309 phosphatidylinositol 4-kinase, catalytic, alpha polypeptide
HK1B6 28410 Human AMP deaminase isoform L (AMPD2) mRNA,  exons 6-18, partial cds
HK1B7 28985 N-acetylgalactosaminidase, alpha-
HK1B8 29054 ARP1 (actin-related protein 1, yeast) homolog A (centractin alpha)
HK1B9 30272 solute carrier family 9 (sodium/hydrogen exchanger), isoform 1 (antiporter, Na+/H
HK1B10      30664 aspartylglucosaminidase
```

Figure 13 Gene info file example

APPENDIX C

SELECTED GENE SUBSET RESULT

Gene	Coloring	B&W changes	-log prob	Color changes	-log prob	Mean	Mean	Mean	Fold change
HV22H6 949932 Major histocompatibility complex, class II, Y box-binding protein I	0.24^{12} 0.50^{14} 0.54^{13} 0.55^{11} 0.80^{8} 0.90^{7} 0.95^{10} 1.06^{9} 1.25^{0} 1.45^{4} 1.49^{5} 1.68^{2} 1.85^{3} 1.85^{1} 2.04^{6}	1	8.08	2	10.13	1.66	0.93	0.46	3.63
HV19C12 784830 D123 gene product	0.49^{12} 0.51^{13} 0.56^{11} 0.62^{14} 0.73^{8} 0.79^{10} 0.86^{7} 0.92^{9} 1.07^{0} 1.13^{4} 1.54^{2} 1.82^{6} 2.17^{5} 2.46^{1} 2.46^{3}	1	8.08	2	10.13	1.81	0.82	0.55	3.32
HV32H3 210646 protein kinase, AMP-activated, gamma 2 non-catalytic subunit	0.01^{9} 0.06^{10} 0.12^{8} 0.14^{7} 0.15^{5} 0.15^{2} 0.16^{6} 0.17^{0} 0.17^{12} 0.21^{14} 0.23^{11} 0.27^{13} 0.28^{4} 0.28^{3} 0.42^{1}	1	6.53	3	8.05	0.23	0.08	0.22	2.79
HV7E4 248531 GUANINE-MONOPHOSPHATE SYNTHETASE	0.54^{12} 0.70^{11} 0.98^{14} 1.16^{10} 1.31^{9} 1.49^{8} 1.57^{7} 1.69^{13} 1.84^{2} 1.94^{3} 1.95^{5} 2.07^{6} 2.67^{4} 2.70^{1} 4.26^{0}	1	8.08	3	8.05	2.49	1.38	0.98	2.55
HV8H1 179804 PWP2 (periodic tryptophan protein, yeast) homolog	0.52^{10} 0.65^{7} 0.92^{2} 0.93^{6} 0.98^{1} 1.04^{3} 1.04^{4} 1.26^{0} 1.51^{5} 1.69^{8} 2.65^{9} 2.95^{14} 3.25^{13} 3.53^{12} 3.53^{11}	1	6.53	3	8.05	1.10	1.38	3.31	3.02
UG7B11 52650 v-ets avian erythroblastosis virus E26 oncogene	[illegible] 0.75^{8} 0.82^{2} 1.09^{3} 1.25^{4} 1.31^{0} 1.45^{13} 1.52^{12} 1.62^{14} 1.63^{11} 1.67^{1} 2.09^{6} 2.96^{5}	1	6.53	3	8.05	1.60	0.57	1.55	2.80

Figure 14 Subset of genes from the top 176 ones with fewest color changes, 3-partition, BRCA data set

Gene	Coloring	B&W changes	-log prob	Color changes	-log prob	Mean	Mean	Fold change
M60854 """Human ribosomal protein S16 ""mRNA,"	905.17^{4} 1164.99^{6} 1352.68^{7} 1758.20^{5} 2958.25^{0} 3161.24^{2} 3467.59^{1} 3479.43^{3}	1	3.56	1	3.56	3266.63	1295.26	2.52
HG821-HT821 Ribosomal Protein S13	236.49^{4} 256.78^{7} 308.94^{6} 397.68^{5} 406.09^{0} 428.82^{3} 450.25^{2} 673.02^{1}	1	3.56	1	3.56	489.54	299.97	1.63
X51466 Human mRNA for elongation factor 2	233.58^{4} 261.10^{6} 303.29^{7} 317.09^{5} 325.90^{0} 374.42^{3} 473.90^{2} 742.98^{1}	1	3.56	1	3.56	479.30	278.76	1.72
X57110 Human mRNA for c-cbl proto-oncogene.	36.87^{5} 45.06^{7} 46.34^{6} 51.47^{4} 60.77^{3} 61.28^{1} 62.52^{2} 66.67^{0}	1	3.56	1	3.56	62.81	44.94	1.40
X52979 """SmB protein gene	17.00^{4} 21.07^{5} 26.91^{6} 29.07^{7} 36.83^{0} 36.97^{3} 57.79^{2} 68.87^{1}	1	3.56	1	3.56	50.12	23.51	2.13
D87735 """Human mRNA for ribosomal protein ""L14,""	194.24^{4} 296.98^{6} 348.83^{7} 379.24^{5} 455.51^{0} 571.62^{3} 629.94^{2} 923.03^{1}	1	3.56	1	3.56	645.02	304.82	2.12
L02785 """Homo sapiens colon mucosa-associated (DRA) ""mRNA,""	-5.42^{1} -2.10^{2} 3.40^{3} 8.39^{0} 368.18^{5} 569.27^{7} 604.42^{6} 645.86^{4}	1	3.56	1	3.56	1.07	546.93	512.35

Figure 15 Subset of genes from the top 200 ones with fewest color changes, 2-partition, Adenoma data set

Image Clone ID	UniGene Cluster ID	UniGene Title
26617	Hs.10247	activated leucocyte cell adhesion molecule
770080	Hs.102497	paxillin
21652	Hs.178452	catenin (cadherin-associated protein), alpha 1 (102kD)
591281	Hs.80680	lung resistance-related protein
290724	Hs.90598	MHC class I polypeptide-related sequence A
30502	Hs.250500	delta-like 1 (mouse) homolog
897781	Hs.242463	keratin 8
139354	Hs.15093	ESTs
39993	Hs.75428	superoxide dismutase 1, soluble (amyotrophic lateral sclerosis
416833	Hs.7557	FK506-binding protein 5
295831	Hs.24332	ESTs, Highly similar to CGI-26 protein [H.sapiens]
810057	Hs.1139	cold shock domain protein A
417226	Hs.79070	v-myc avian myelocytomatosis viral oncogene homolog
789182	Hs.78996	proliferating cell nuclear antigen
950682	Hs.99910	phosphofructokinase, platelet
26184	Hs.99910	phosphofructokinase, platelet
26184	Hs.99910	phosphofructokinase, platelet
344109	Hs.78996	proliferating cell nuclear antigen
783698	Hs.81412	KIAA0188 protein
134748	Hs.77631	glycine cleavage system protein H (aminomethyl carrier)
36775	Hs.75860	hydroxyacyl-Coenzyme A dehydrogenase/3-ketoacyl-Coenzym

Figure 16 Subset of the listed discriminating genes for BRCA data set