2013

A Spectral Alternative to K-means Clustering for Graph Data

Becca Simon
College of Saint Benedict/Saint John's University

A Spectral Alternative to K-means Clustering for Graph Data

AN HONORS THESIS

College of St. Benedict/St. John's University

In Partial Fulfillment

of the Requirements for All College Honors

and Distinction

in the Department of Computer Science

by

Becca Simon

April 2013

PROJECT TITLE: A Spectral Alternative to K-means Clustering for Graph Data

Approved by:

Michael Heroux
Scientist-in-Residence, Computer Science Department

Robert Hesse
Associate Professor of Mathematics

Michael Gass
Associate Professor of Mathematics

James Schnepf
Chair, Department of Computer Science

Anthony Cunningham
Director, Honors Thesis Program

Contents

1 Introduction **4**

2 Background **5**
2.1 Linear Algebra Concepts 5
2.2 K-means Clustering 6
2.3 Preprocessing 6
2.3.1 Singular Value Decomposition 6
2.3.2 Cosine Similarities 7
2.3.3 Thresholding 7
2.4 Representing Graphs 8
2.4.1 Adjacency Matrix 8
2.4.2 Degree Matrix 8
2.4.3 Laplacian Matrix 8
2.4.4 Fiedler Vector 9
2.5 Accuracy Measures 9

3 Purpose **10**

4 Methods **11**
4.1 recursivePartition 11
4.2 recursiveRatio 12
4.3 recursiveRatioSort 13
4.4 fiedlerRatioSort 16
4.4.1 laplacian 16
4.4.2 fiedler 16
4.4.3 ratioCut 17
4.5 updateVector 17
4.6 boolean 17

5 Results **18**

6 Conclusions **27**
6.1 Future Work 28

Appendices **31**

Appendix A Cut Criteria **31**

Appendix B Programs **33**
B.1 recursivePartition.m 33
B.2 preprocess2.m 34
B.3 recursiveRatio.m 35
B.4 recursiveRatioSort.m 36
B.5 laplacian.m 38
B.6 fiedler.m 39
B.7 cut.m 40
B.8 ratioCut.m 41
B.9 minMaxCut.m 42
B.10 nCut.m 43
B.11 minCut.m 44
B.12 updateVector.m 45
B.13 boolean.m 46

Abstract

Producing meaningful clusterings for graph data requires the user to provide some insight to the program which he or she may not have regarding the data. The standard clustering algorithm, K-means, requires the user to specify k, the number of clusters to be produced by the algorithm. This paper discusses the recursivePartition algorithm, a recursive alternative to K-means clustering. The input to recursivePartition asks the user to specify n, the maximum size of any cluster. Using maximum cluster size and spectral methods based on the Laplacian matrix of the graph, recursivePartition has demonstrated an ability to produce highly accurate clusters over a range of inputs, even producing an exact match of the true clusterings present in the data in multiple tests. recursivePartition is capable of producing highly accurate clusters with a robustness to user input which the K-means clustering algorithm cannot match.

1 Introduction

Graphs are a convenient and efficient means of representing social networks and term-document data. Meaningful analysis of this data often includes a need for clustering the data into distinct subsets of related nodes. While the standard algorithm for graph clustering, K-means, is known to produce meaningful clusters of data given an appropriate k value specifying the number of clusters to produce, it is an iterative algorithm which can be quite slow for large datasets. Given the nature of the algorithm and necessary calculations, K-means is not conducive to speedup through implementations involving parallel computing. Based on results by other researchers (see [12], [8], [10], [2], [14]), this experiment sought to use aspects of spectral theory as the basis for a recursive clustering algorithm which could be implemented in parallel in the future for speedup over K-means. Due to time constraints and changing interests, the focus of this project shifted toward producing an algorithm which was not as tightly restricted by the input value provided by the user as K-means is. This algorithm, recursivePartition, was designed to take the nature of the data into account in the number of clusters it produces. It still requires the user to provide an input parameter, n, but n denotes the maximum allowable size of any cluster rather than the total number of clusters, specified as k in K-means. This experiment seeks to show that the recursivePartition algorithm is somewhat robust to changing

inputs while still being capable of producing quality clusterings. The results of recursivePartition produced in this experiment are compared to the results of a K-means clustering for ensuring that the high level of accuracy provided by K-means is not lost. All computations in this experiment were performed using Matlab because it provides a simple platform for producing and manipulating matrices.

2 Background

This section gives a brief overview of the linear algebra concepts being used in this work. It will also provide an explanation of k-means clustering, the standard clustering technique being used for baseline comparison. An explanation of data preprocessing techniques used and accuracy measures for evaluating clusters is also included.

2.1 Linear Algebra Concepts

The clustering algorithm being explored in this paper is based on the linear algebra concepts of eigenvalues and eigenvectors.

Definition An **eigenvector** of an n x n matrix A is a nonzero vector x such that $A\mathbf{x} = \lambda\mathbf{x}$ for some scalar λ. A scalar λ is called an **eigenvalue** of A if there is a non-trivial solution of $A\mathbf{x} = \lambda\mathbf{x}$; such an $\mathbf{x}$ is called an *eigenvector corresponding to* λ[9].

Manually, the eigenvalues of a matrix A are found by finding all scalars λ such that the matrix equation $(A - \lambda I)\mathbf{x} = 0$ has a nontrivial solution. I is the n x n identity matrix. The corresponding eigevector,v_i, of an eigenvalue λ_i is then found by using row operations to reduce $A - \lambda_i I$ in order to solve the eigenvalue equation by inspection[9]. Matlab provides a built-in function for finding the k largest magnitude eigenvalues of a matrix A and their corresponding eigenvectors. This function, $[v, d] = eigs(A, k)$, will be used for our purposes. The output v is an n x k matrix containing the eigenvectors of A as columns. d is a k x k matrix containing the k largest magnitude eigenvalues of A along the major diagonal.

2.2 K-means Clustering

K-means is one of the most commonly used clustering algorithms and is based on a simple idea. A set of representative centroids is chosen, and each point in the dataset is assigned to its closest centroid. The number of centroids chosen is based upon a user-specified parameter k, the desired number of clusters to be found. Each centroid is then updated to be the mean of all the points assigned to it, and the process repeats itself. This continues until the centroids make it through an iteration unchanged[13].

This experiment used an implementation of the K-means algorithm written in Matlab as a baseline comparison for cluster quality. It was provided by Dr. Daniel M. Dunlavy of Sandia National Laboratories [7].

2.3 Preprocessing

The following sequence of preprocessing techniques was applied to the data prior to clustering.

2.3.1 Singular Value Decomposition

Singular Value Decomposition (SVD) is a matrix factorization from applied linear algebra. Given an m x n matrix A, the singular value decomposition of A takes the form

$$A = U\Sigma V^T \tag{1}$$

where Σ is an m x n matrix with the singular values of A in non-increasing order along the major diagonal, U is an m x m matrix of the left singular vectors of A ordered to correspond with Σ, and V is an n x n matrix of the right singular vectors of A ordered to correspond with Σ.

The singular values of a matrix A, denoted by $\sigma_1, ..., \sigma_n$, are the square roots of the non-zero eigenvalues of A^TA. The right singular vectors of A, $V = [v_1, v_2, ..., v_n]$, are the eigenvectors of A^TA, and the left singular vectors of A, $U = [u_1, u_2, ..., u_m]$, are the eigenvectors of AA^T (see section 2.1)[9].

Matlab provides an implementation of SVD for sparse matrices which returns the k largest magnitude singular values and their corresponding right and left singular vectors for a matrix A. This function, $[U, S, V] = svds(A, k)$, was used to perform all singular value decompositions for this experiment.

A new n x k matrix VS was then constructed by multiplying $V * S$, where V, S are the right singular vectors of A and singular values of A, respectively(see 2.3.2).

2.3.2 Cosine Similarities

Given the high dimensionality of the original data and its sparse nature (matrix is populated primariliy with zeros), cosine similarity was used to transform the original term-document matrix into a similarity matrix. Cosine similarity is a $[0, 1]$ similarity measure based on the cosine of the angle between two vectors of the same dimensionality, x and y. $cos(x, y) = 1$ iff $x = y$ and $cos(x, y) = 0$ iff x and y share exactly zero terms. The cosine similarity between x and y is computed as

$$cos(x, y) = \frac{x \cdot y}{\|x\|\|y\|} \tag{2}$$

$$= \frac{x}{\|x\|} \cdot \frac{y}{\|y\|} \tag{3}$$

$$= x' \cdot y' \tag{4}$$

where $x' = \frac{x}{\|x\|}$[13]. Thus, by first normalizing each row vector to unit length, an m x m cosine similarity matrix $VSsims$ can be computed by straightforward matrix multiplication, $VSsims = VS * VS^T$[8].

The cosine similarity matrix is computed from the n x k matrix VS (see section 2.3.1), so the final dimensions of the symmetric cosine similarity matrix $VSsims$ are n x n, where n is the number of columns in the original m x n term-document matrix A.

2.3.3 Thresholding

The preprocessing performed up to this point has transformed an m x n sparse term-document matrix into an n x n full matrix $VSsims$ generated by the most strongly correlated dimensions as determined by cosine similarity and singular value decomposition. As a result of these preprocessing steps, most (over 95%) of the values in the full matrix $VSsims$ are small in magnitude ($\leq .2$) compared to the rest of the matrix; thus, these values contribute relatively little information for clustering and are likely noise. To eliminate those data points contributing the least information and allow for more efficient storage and clustering, $VSsims$ is thresholded into a sparse

matrix $VSsims\_t$ containing only those values where $VSsims > .2$. The threshold was selected based on a spy plot of the data. It appears to retain nearly all of the data in clusters while eliminating much of the surrounding noise present in the data.

2.4 Representing Graphs

The following sections detail the matrices used to represent the graphs processed in this experiment. The term-document data is represented as a graph $G = (V, E)$, where $V = v_1, v_2, \ldots, v_n$ is the vertex set. Each vertex v_i represents a document in the dataset. The edge set E indicates a connection between vertices. In this case, an edge would represent a shared term.

2.4.1 Adjacency Matrix

An adjacency matrix A of graph G, also called a similarity matrix of graph G, is an n x n symmetric matrix, where n is the number of vertices v_i in G. The values of A are determined by the non-negative weights of the edges in the graph. Due to the preprocessing described above (see section 2.3), the weights of G correspond to the cosine similarities of the data after SVD. A positive entry $A_{i,j}$ indicates shared terms between vertices v_i and v_j. $A_{i,j} = 0$ indicates no shared terms between vertices (or a low similarity which was thresholded to zero during preprocessing)[10].

2.4.2 Degree Matrix

The degree matrix D of a graph G is the diagonal matrix of the degrees of each vertex v_i in G. The degree is calculated as the sum of the weights of all the edges incident to v_i, ie $d(v_i) = \sum_j w(i, j)$, where $w(i, j)$ is the weight of the edge between v_i and v_j[3]. All nondiagonal entries of D are zero. Given the adjacency matrix A, $d(v_i) = \sum_j a_{i,j}$[11].

2.4.3 Laplacian Matrix

A matrix representation of a graph, referred to as the Laplacian of the graph, is the basis for spectral clustering of graph data[10]. Given the adjacency matrix A (see section 2.4.1) and the degree matrix D (see section 2.4.2) for an unweighted graph G, the Laplacian of G is defined as $L(G) = D(G) - A(G)$[11]. Due to the preprocessing techniques applied to the data, the graphs for this

experiment are weighted and may contain loops. The generalized Laplacian for this data is defined as

$$L(u,v) = \begin{cases} d_v - w(v,v) & \text{if } u = v \\ -w(u,v) & \text{if } u,\ v \text{ adjacent} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

[3]. This is the format of the Laplacian which will be used for all experiments in this project.

2.4.4 Fiedler Vector

Using the eigenvectors (see section 2.1) of the Laplacian of the graph G (see section 2.4.3) provides a means for determining an appropriate bisection of the vertices of G. This bisection is based on an ordering of the eigenvector referred to as the Fiedler vector. The Fiedler vector is the eigenvector corresponding to the smallest magnitude, non-null eigenvalue of the graph Laplacian, also known as the Fiedler value. The Fiedler value is also referred to as the "algebraic connectivity of a graph" because it indicates how strongly connected the graph is. The further the Fiedler value is from zero, the more strongly connected the graph[12]. It has been shown that a small magnitude Fiedler value will lead to a good partition of the graph based on the ratio cut criterion[1]. Ratio cut and a number of other cut criteria based on the sorted Fiedler vector will be compared in this paper.

2.5 Accuracy Measures

Given any clustering of graph data, accuracy measures must be used in order to determine how good the clustering is. All of the data used for this paper has labels indicating what the ideal clustering of the data should be. These labels were used as the "true clusters" for comparison when determining the accuracy of clusters produced in this experiment. Two evaluations of cluster accuracy were computed: Rand Index and Jaccard Index. The same two accuracy calculations were computed for each dataset clustered by K-means and used as a baseline comparison. Each accuracy measure is a value in the range [0,1], with 1 indicating that the two clusterings are identical up to a permutation, that is, for any subset of points forming a cluster in one clustering, the same set of points forms a cluster in the other cluster with no additions or deletions.

This experiment used Matlab implementations of Rand Index and Jaccard Index provided by Dr. Daniel M. Dunlavy of Sandia National Laboratories [6], [5].

3 Purpose

The purpose of this experiment was to produce an algorithm (which I have named recursivePartition) for recursively determining clusters in graph data. The means of producing these clusters was based on spectral theory and the results of other researchers (see [12], [8], [10], [2], [14]). My initial purpose was to produce a recursive clustering algorithm for graph data which maintains the accuracy levels of K-means clustering and could be implemented in parallel for speedup over K-means when clustering large graphs. Due to changing circumstances and time constraints, I decided to leave the parallel implementation to a future experiment and focus on the algorithm for the serial implementation. I wrote a serial prototype in Matlab which uses a series of functions to perform a recursive clustering of graph data until all clusters meet a user-given maximum cluster size. This implementation performs a bisection at each level of recursion, continuing the clustering on both subgraphs produced by the bisection until the clusters are appropriately small. The user has an option among Ratio cut, Normalized cut, Min cut, and Min-Max cut as the mathematical criteria for determining where the bisection occurs. Explanations of all these cut criteria may be found in Appendix A and in more detail in [10].

One of the primary difficulties in using clustering algorithms is determining that the optimal number of clusters for the data because it is usually unknown. For K-means, this requires running the algorithm on the same data repeatedly with varying k values and choosing a seemingly suitable clustering from among the results. This can be costly because the K-means algorithm is $O(n * k * I * d)$, where n is the number of nodes in the data, k is the number of clusters to be produced, I is the number of iterations required for convergence, and d is the number of attributes present in the data [13]. The recursivePartition algorithm is designed to be somewhat robust to the issue of determining an appropriate number of clusters. By having the user specify a maximum cluster size rather than a required number of clusters, the algorithm allows for some flexibility based on the bisections. This experiment will seek to demonstrate that flexibility.

4 Methods

This section will provide an overview of the algorithm developed for this experiment. It assumes the data has not been preprocessed. All programs are included in Appendix B. Each subsection below represents a distinct Matlab program. All Matlab functions written for the recursivePartition algorithm are available at **https://github.com/blsimon/recursivePartition.git**.

4.1 recursivePartition

As previously mentioned, there are a number of different cut criteria that may be used to determine the point of bisection at each level of partitioning. Therefore, my implementation features a generic wrapper method recursivePartition which takes three parameters:

A the unpreprocessed data to be clustered

n an integer representing the maximum size any cluster may be

criteria a String specifying either 'normalized', 'ratio', min', or 'minmax' as the cut criteria to be used

I will trace an implementation of ratio cut, but all the implementations follow the same basic pattern, differing only in the specific cut criteria used and the names of the methods. The return values of recursivePartition are

partitions a vector containing the index of the beginning of each distinct cluster in the data

index a matrix with the same number of rows as there are data points and a column for each level of recursion; each column of *index* contains the node ordering at that level of recursion; the first column is in numerical order because it represents the starting point of the data, and the last column of *index* contains the final node ordering produced by the program

Combined, *partitions* and *index* provide an ordering of the nodes with the index of each new cluster, providing all information necessary for determining to which cluster each node belongs.

recursivePartition first calls preprocess2, which takes the data matrix A as a parameter and returns the preprocessed data matrix A as described

in section 2.3. Assuming we are tracing the implementation of ratio cut, recursivePartition calls the recursiveRatio method, which takes the first two parameters of recursivePartition, A and n, and returns *partitions* and *index*, the return values of recursivePartition.

4.2 recursiveRatio

recursiveRatio then initializes a series of variables for use in the coming recursion.

sizeA an integer representing the number of nodes in the data currently being partitioned at that level of recursion.

partitions return value initialized as a 2 x 1 vector of zeros to be filled later with the index of the first node in each cluster

i an integer initialized to one which will track the location in the *partitions* vector where the next value should be placed

temp an integer initialized to zero which will later be used for temporarily storing the beginning of a partition if the lower portion of the data partitioned into a subset less than or equal to the max cluster size n while the upper portion of the data is too large and must undergo additional levels of recursive partitioning

index return value initialized as an m x 1 matrix of zeros, where m is the total number of nodes to be clustered; this will later be filled with the current node ordering at each level of recursion (as described in section 4.1); the first column of *index* gets initialized to $1, 2, 3, ..., m$ because the current node ordering is numeric since the data has not yet undergone any clustering;

constant an integer initialized to zero; *constant* will later be used in updating the universal node ordering for *index*, where *constant* will represent the offset from zero at which the data currently being partitioned begins

j an integer initialized to one which is used to maintain the current column of *index* to be filled with the node ordering produced by that level of recursion

recursiveRatio then calls recursiveRatioSort, which, despite the names of some of the other functions, is the true recursive function of the implementation and will be explained in section 4.3. After recursiveRatioSort returns, recursiveRatio cleans up and finalizes the values of *partitions* and *index* before returning them. *index* is finalized by using the built-in Matlab find function to locate any remaining zeros in *index* and replace them with the value in the previous column. A zero value in *index* would occur whenever a cluster was found with fewer levels of recursion than the maximum number of levels of recursion required by any cluster. Replacing any zero values guarantees that the last column of *index* is the true node ordering for the clusters produced by the algorithm. *partitions* is finalized by checking if the last element of *partitions* exceeds the number of elements in the original data. If so, that element is removed from *partitions*. This removal is necessary when the lower subset of the data has finished clustering before the upper subset and the value saved for later filling into the *partitions* vector. Removing the non-existent element has no effect on the rest of the clustering.

4.3 recursiveRatioSort

recursiveRatioSort handles the recursive calls for the algorithm using all the functions described below as helpers. The inputs are as follows:

A the n x n adjacency matrix of the graph to be partitioned

n an integer representing the maximum cluster size as specified by the user

partitions the current state of the *partitions* vector described in section 4.1

i an integer tracking the current number of partitions; it is used as an index in the *partitions* vector and must be kept current at each level of recursion for use in the boolean function (see section 4.6)

temp an integer for temporarily tracking the index of the lower partition if the lower partition is finished partitioning while the upper partition must undergo additional levels of recursion; it must also be maintained for use in the boolean function

index the current state of the *index* matrix described in section 4.1

constant an integer used as an offset for determining where in the overall graph the current subgraph occurs; *constant* is determined by the size of the partition presently being processed and will be used for updating the *index* matrix in the function updateVector (see section 4.5)

j an integer track which column of *index* is currently being updated for this subgraph; j is also used as an input to the updateVector function for updating *index*

The following outputs are returned by the function recursiveRatioSort:

*sizepartition*1 **and** *sizepartition*2 integers representing the sizes of the upper and lower subgraphs found by the function, respectively; they are needed for inputs to the boolean function (see section 4.6) and for determining whether additional recursion is necessary for either or both subgraphs

partitions an updated version of the input described above

i an updated version of the input described above

index an updated version of the input described above

constant an updated version of the input described above

recursiveRatioSort begins by incrementing j, the integer tracking which column of the *index* matrix is being affected by this current level of recursion. It then calls fiedlerRatioSort (see section 4.4) and uses the returned permutation vector p as an input to updateVector (see section 4.5). recursiveRatioSort then sets *sizepartition*1 and *sizepartition*2 as the sizes of $g1$ and $g2$, respectively. $g1$ and $g2$ are the adjacency matrices of the two subgraphs found by the call to fiedlerRatioSort.

The function then enters a four-part if-statement based upon the sizes of *sizepartition*1 and *sizepartition*2 relative to the user-provided maximum cluster size, n.

If both *sizepartition*1 and *sizepartition*2 are currently larger than n, the recursion will continue for both the upper and lower subgraphs. The variables *boolean*1 and *boolean*2 are set to 0, indicating that neither subgraph is finished being partitioned. The boolean function (see section 4.6) is called to ensure the *partitions* vector is updated appropriately before the next recursive call to recursiveRatioSort. For the case of both subgraphs needing to

undergo continued partitioning, boolean will only check if the *partitions* vector is currently all zeros, replacing the first index with 1 if so. This indicates that the first partition will begin at index 1 of the original graph. recursiveRatioSort then makes a recursive call to itself, using the upper subgraph $g1$ for the input A. This change will begin the recursive partitioning process on the upper subgraph, stopping only when all partitions of this subgraph meet the maximum cluster size criteria. Once the call to recursiveRatioSort on the upper subgraph returns, constant is updated to be 1 less than the location of the first 0 in the j^{th} column of *index*, ie the column of *index* corresponding to the current level of recursion. Another recursive call is then made to recursiveRatioSort, but this time the subgraph $g2$ from fiedlerRatioSort is used as the input graph to the function. Once this recursion returns, the current call to recursiveRatioSort returns.

If only $sizepartition1$ is less than n, ie only the upper subgraph meets the user-given maximum cluster size, recursion will continue on the upper subgraph and the lower partition will be saved to be recorded later. $boolean1$ is set to 0, indicating the upper subgraph is not finished partitioning. $boolean2$ is set to 1 to indicate that the lower subgraph is finished. The boolean function is called to store the necessary information about the lower subgraph in the *temp* variable until the upper subgraph is finished. A recursive call to recursiveRatioSort is then made on $g1$, the upper subgraph. After the recursion returns, a call to the boolean function will use the *temp* variable to save the lower subgraph information to the *partitions* vector appropriately. The current call to recursiveRatioSort then returns.

If only $sizepartition2$ is less than n, ie only the lower subgraph meets the user-given maximum cluster size, the upper partition will be recorded in *partitions* and recursion will continue on the lower subgraph. $boolean1$ is set to 1, indicating that the upper subgraph is finished being partitioned. $boolean2$ is set to 0 to indicate that the lower subgraph must undergo additional partitioning. *constat*, the variable used for updating the universal node ordering in the *index* matrix, is updated by adding $sizepartition1$ to its current value because the node ordering for the upper subgraph will not be affected by the additional recursion on the lower subgraph. The boolean function is called to record the beginning index of the lower subgraph as the start of the next partition in the *partitions* vector. A recursive call to recursiveRatioSort is then made on $g2$, the lower subgraph. After this recursion returns, the current call to recursiveRatioSort returns.

If the else portion of the if-statement executes, ie both the upper and lower subgraphs are smaller than the maximum cluster size, no recursive call to recursiveRatioSort will be made. The variables *boolean*1 and *boolean*2 are both set to 1, indicating that both portions of the graph are finished partitioning. A call to the boolean function updates the *partitions* vector using the sizes of both the upper and lower subgraphs and the current call to recursiveRatioSort returns.

4.4 fiedlerRatioSort

fiedlerRatioSort is a function to partition the graph into two distinct subgraphs based on the sorted Fiedler vector of the graph. This function takes the n x n adjacency matrix I of the graph to be partitioned. It generates the Laplacian of I (see section 4.4.1). It then generates the Fiedler vector of that Laplacian (see section 4.4.2), which gets sorted by the built-in Matlab sort function. The nodes of I are rearranged to match the order of the sorted Fiedler vector. Finally, I is partitioned into subgraphs A and B using the ratioCut function (see section 4.4.3). fiedlerRatioSort returns an n x 1 vector p containing the permutation of nodes used to sort I based on its Fiedler vector, and the adjacency matrices A and B of the two distinct subgraphs found.

4.4.1 laplacian

laplacian is a helper method to compute the Laplacian of the graph data, as described in section 2.4.3. laplacian takes the n x n adjacency matrix A of the data to be clustered and returns the n x n matrix L representing the Laplacian of A. L is computed by forming the diagonal n x n matrix D of the sums of each column of A and subtracting $L = D - A$.

4.4.2 fiedler

fiedler is a helper method to generate the Fiedler value and corresponding Fiedler vector (ie the smallest magnitude nonzero eigenvalue and its corresponding eigenvector) as described in section 2.4.4. The input is the n x n Laplacian matrix L produced by the laplacian function. The output arguments v and d are the Fiedler vector of L and corresponding Fiedler vector of L, respectively.

4.4.3 ratioCut

ratioCut is a function for determining where to bisect the Fiedler vector. In running the algorithm with a different initial cut criteria input, ratioCut would be replaced by the appropriate corresponding function, such as minMaxCut, nCut, or minCut. The input to each of these cut functions is the n x n matrix $sortedI$, which is the graph I permuted based on the sorted Fiedler vector, that is $sortedI = I(p, p)$, where I is the current subset of the graph data at this level of recursion. ratioCut returns two matrices, A and B, which are the adjacency matrices of the distinct subgraphs of I partitioned at the optimum cutpoint according to the Ratio cut criteria. It uses a function called cut to evaluate how many edges would be cut for any given subsets A and B. Ratio cut and all other cut criteria implemented are explained mathematically in Appendix A.

4.5 updateVector

updateVector is a helper method used to update the output matrix *index*, the universal ordering matrix consisting of the ordering of the nodes at each level of recursion. The first input to the function is the vector p produced by fiedlerRatioSort (see section 4.4). The node ordering matrix *index* is the second input, and an updated version of this matrix will serve as the output of the function. The third input is an integer called *constant* which is used as an offset for determining where in the overall graph this subset occurs. *constant* is determined by the size of the partition presently being processed. The final input is the integer j, which indicates which column of *index* is to be updated based on the present level of recursion.

4.6 boolean

boolean is a helper method to update the *partitions* vector. The inputs are as follows:

partitions the vector described above which records the first index of each of the partitions

*boolean*1 **and** *boolean*2 booleans indicating if the upper portion and lower portion of the partition, respectively, are less than the user-given maximum cluster size

i an integer tracking the total number of partitions; it is used as an index in the *partitions* vector

*sizepartition*1 **and** *sizepartition*2 the sizes of the upper and lower portion of the partition, respectively

temp an integer for temporarily tracking the index of the lower partition if the lower partition is finished partitioning while the upper partition must undergo additional levels of recursion

n the user-given maximum cluster size

The outputs of this function are the integer i, the vector *partitions*, and the integer *temp*. These outputs are the updated versions of their corresponding inputs.

See section 4.3 for an explanation of how this function updates the *partitions* vector under the various cases relating to the size of the upper and lower subgraphs.

5 Results

While this experiment is motivated by the fact that the ideal clusters or number of clusters are generally unknown for a dataset, all of the graphs used in this experiment have known clusterings. This is necessary for using accuracy measures to determine how the recursive algorithm is performing compared to the K-means algorithm (see section 2.5 for explanation of accuracy measures used). This experiment uses the standard K-means algorithm (see section 2.2) as a baseline accuracy comparison as well as a version of K-means using stochastic perturbations of the clusterings "to avoid local minima of the multimodal objective function being minimized in the standard k-mean algorithm" [7]. The perturbation method of K-means produces higher accuracy values than the standard approach and is used as a goal for the clustering quality of recursivePartition. This experiment uses one real dataset, a weighted term-document matrix representing a collection of 298 newswire documents from the Associated Press and the New York Times. This collection contains 8118 terms and 30 clusters determined by the main topic of each document. It was provided by Dr. Daniel M. Dunlavy of Sandia National Laboratories. Artificial datasets were generated using a Matlab

function developed by Dr. Daniel M. Dunlavy [4]. This function produces adjacency matrices for an artificial graph containing a user-specified number of clusters, minimum cluster size, and maximum cluster size. It also produces the true clusters corresponding to the graph. Each experiment is performed 10 times, and the average accuracy values are reported below. The initial parameters selected for each algorithm are based on educated guesses with some knowledge of the data and highlighted in yellow in the results tables. The parameters resulting in the highest accuracy for each algorithm are highlighted in orange.

Table 1 summarizes the results of the experiment using the Associated Press and New York Times newswire document collection. Because this data contains 30 clusters, the K-means algorithm was run with $k = 30$. Additional runs using $k = 28$, $k = 29$, $k = 31$, and $k = 32$ were used to demonstrate how K-means performs if too few or too many clusters are sought. The initial n value for recursivePartition was determined by dividing the total number of nodes in the data by the number of known clusters (ie $\frac{298 nodes}{30 clusters} \approx 10$ nodes per cluster). Because n is the *maximum* cluster size to be found by recursivePartition, there is some danger of choosing too small an n value. Experiments using n greater than the initial $n = 10$ approximation demonstrate the robustness of recursivePartition to larger n values. recursivePartition outscores the baseline K-means and K-means with perturbations using $k = 30$ in both Rand Index and Jaccard Index for $n = 10$ to $n = 15$.

Table 1: Results of K-means and recursivePartition on AP and NYT data
Data contains 30 real clusters
Yellow highlighting denotes the initial parameter value for each algorithm
Orange highlighting indicates the highest accuracies

Algorithm	Parameter	RI	JI	Num Clusters
K-means	$k = 28$	.9809	.5881	28
K-means	$k = 29$	.9800	.5747	29
K-means	$k = 30$	.9831	.6140	30
K-means	$k = 31$	.9836	.6205	31
K-means	$k = 32$	.9856	.6495	32
K-means perturb	$k = 28$	.9824	.6097	28
K-means perturb	$k = 29$	.9818	.6015	29
K-means perturb	$k = 30$	.9847	.6442	30
K-means perturb	$k = 31$	.9852	.6481	31
K-means perturb	$k = 32$	.9879	.6906	32
recursivePartition	$n = 9$	.9849	.5318	52
recursivePartition	$n = 10$	.9886	.6536	42
recursivePartition	$n = 11$	.9892	.6757	39
recursivePartition	$n = 12$	.9888	.6760	36
recursivePartition	$n = 13$	.9878	.6583	34
recursivePartition	$n = 14$	.9877	.6605	33
recursivePartition	$n = 15$	.9870	.6524	31

Artificial data was generated using the gen_cluster_graph function [4]. The data contains 453 nodes in 30 clusters of minimum size 10 and maximum size 20. Table 2 summarizes the results. $k = 30$ was chosen as the starting parameter for K-means because the data is known to contain 30 clusters. The starting parameter for recursivePartition was chosen as $n = 16$ by dividing the number of nodes by the number of known clusters to produce average cluster size. The recursivePartition algorithm for $n = 16$ to $n = 21$ produced higher RI and JI values than K-means and K-means with perturbations for the values of $k = 28$ to $k = 32$, including $k = 30$, the true number of clusters present in this artificial data. Additionally, recursivePartition with $n = 20$ produced 30 clusters which identically matched the true clusters in the data up to a permutation, as demonstrated by its $RI = 1.0000$ and $JI = 1.0000$ accuracies.

Table 2: Results of K-means and recursivePartition on Artificial Data
Data contains 30 real clusters of minimum 10 and maximum 20 nodes

Algorithm	Parameter	RI	JI	Num Clusters
K-means	$k = 28$	.9853	.6859	28
K-means	$k = 29$	.9871	.7152	29
K-means	$k = 30$	.9830	.6556	30
K-means	$k = 31$	.9882	.7346	31
K-means	$k = 32$	.9872	.7230	32
K-means perturb	$k = 28$	.9877	.7237	28
K-means perturb	$k = 29$	.9886	.7407	29
K-means perturb	$k = 30$	.9866	.7041	30
K-means perturb	$k = 31$	.9911	.7836	31
K-means perturb	$k = 32$	.9915	.7915	32
recursivePartition	$n = 15$	.9914	.7359	46
recursivePartition	$n = 16$	.9942	.8229	40
recursivePartition	$n = 17$	.9952	.8523	38
recursivePartition	$n = 18$	.9964	.8880	36
recursivePartition	$n = 19$	.9981	.9414	33
recursivePartition	$n = 20$	1.0000	.1.0000	30
recursivePartition	$n = 21$	.9999	.9968	30

Artificial data was generated using the gen_cluster_graph function [4]. The data contains 568 nodes in 30 clusters of minimum size 18 and maximum size 20. Table 3 summarizes the results. $k = 30$ was chosen as the starting parameter for K-means because the data is known to contain 30 clusters. The starting parameter for recursivePartition was chosen as $n = 19$ by dividing the number of nodes by the number of known clusters to produce average cluster size. Again, recursivePartition using a range of n values outperforms K-means and K-means with perturbations for the known k value.

Table 3: Results of K-means and recursivePartition on Artificial Data Data contains 30 real clusters of minimum 18 and maximum 20 nodes

Algorithm	Parameter	RI	JI	Num Clusters
K-means	$k = 28$	.9787	.5945	28
K-means	$k = 29$	.9793	.6034	29
K-means	$k = 30$	.9808	.6192	30
K-means	$k = 31$	.9818	.6321	31
K-means	$k = 32$	.9842	.6617	32
K-means perturb	$k = 28$	.9822	.6361	28
K-means perturb	$k = 29$	.9831	.6488	29
K-means perturb	$k = 30$	.9851	.6733	30
K-means perturb	$k = 31$	.9845	.6654	31
K-means perturb	$k = 32$	.9867	.6980	32
recursivePartition	$n = 18$	.9916	.7365	52
recursivePartition	$n = 19$	.9936	.8006	40
recursivePartition	$n = 20$	.9968	.9030	30
recursivePartition	$n = 21$	.9964	.8924	35
recursivePartition	$n = 22$	.9967	.8984	30
recursivePartition	$n = 23$	.9969	.9059	30
recursivePartition	$n = 24$	.9981	.9423	30

Artificial data containing 600 nodes in 30 clusters of size 20 was generated using the gen_cluster_graph function [4]. Table 4 summarizes the results. $k = 30$ was chosen as the starting parameter for K-means because the data is known to contain 30 clusters. The starting parameter for recursivePartition was chosen as $n = 20$ because the data was generated to have all clusters of exactly size 20. With the exception of the JI value for recursivePartition with $n = 19$, the recursivePartition algorithm for the range of values of $n = 19$ to $n = 25$ produced higher accuracies than K-means and K-means with perturbations.

Table 4: Results of K-means and recursivePartition on Artificial Data
Data contains 30 real clusters of 20 nodes

Algorithm	Parameter	RI	JI	Num Clusters
K-means	$k = 28$	.9773	.5812	28
K-means	$k = 29$	.9789	.5971	29
K-means	$k = 30$	.9810	.6214	30
K-means	$k = 31$	.9814	.6293	31
K-means	$k = 32$	.9824	.6386	32
K-means perturb	$k = 28$	.9804	.6135	28
K-means perturb	$k = 29$	.9828	.6429	29
K-means perturb	$k = 30$	.9823	.6364	30
K-means perturb	$k = 31$	.9846	.6671	31
K-means perturb	$k = 32$	.9870	.7015	32
recursivePartition	$n = 19$	.9888	.6577	50
recursivePartition	$n = 20$	.9965	.8928	31
recursivePartition	$n = 21$	.9965	.8923	30
recursivePartition	$n = 22$	.9932	.7990	37
recursivePartition	$n = 23$	.9939	.8213	43
recursivePartition	$n = 24$	.9967	.8991	30
recursivePartition	$n = 25$	.9966	.8984	32

Artificial data containing 460 nodes in 30 clusters of minimum size 10 and maximum size 20 was generated with noise added using the gen_cluster_graph function [4]. Again, $k = 30$ was used as the starting K-means parameter because the data contained 30 known clusters. Dividing the number of nodes by the number of clusters produced a starting recursivePartition parameter of $n = 16$. Table 5 shows recursivePartition for $n = 15$ to $n = 21$ outperformed K-means and K-means with perturbations for $k = 30$ with respect to RI value. recursivePartition outperformed all runs of K-means and K-means with perturbations, $k = 28$ to $k = 32$ for both RI and JI accuracy for the range of $n = 17$ to $n = 21$. In fact, recursivePartition using $n = 20$ and $n = 21$ both produced 30 clusters which exactly matched the true clustering of the data up to a permutation, as indicated by their RI and JI scores of 1.0000.

Table 5: Results of K-means and recursivePartition on noisy Artificial Data Data contains 30 real clusters of minimum 10 and maximum 20 nodes

Algorithm	Parameter	RI	JI	Num Clusters
K-means	$k = 28$	.9867	.7128	28
K-means	$k = 29$	.9870	.7141	29
K-means	$k = 30$	.9881	.7304	30
K-means	$k = 31$	.9895	.7515	31
K-means	$k = 32$	.9868	.7186	32
K-means perturb	$k = 28$	.9888	.7482	28
K-means perturb	$k = 29$	.9895	.7566	29
K-means perturb	$k = 30$	.9898	.7601	30
K-means perturb	$k = 31$	.9910	.7812	31
K-means perturb	$k = 32$	.9909	.7787	32
recursivePartition	$n = 15$	.9919	.7500	45
recursivePartition	$n = 16$	.9928	.7779	43
recursivePartition	$n = 17$	.9943	.8256	40
recursivePartition	$n = 18$	.9948	.8419	39
recursivePartition	$n = 19$	.9994	.9814	31
recursivePartition	$n = 20$	1.0000	1.0000	30
recursivePartition	$n = 21$	1.0000	1.0000	30

Artificial data containing 568 nodes in 30 clusters of minimum size 18 and maximum size 20 was generated with noise added using the gen_cluster_graph function [4]. Again, $k = 30$ was used as the starting K-means parameter because the data contained 30 known clusters. Dividing the number of nodes by the number of clusters produced a starting recursivePartition parameter of $n = 19$. Table 6 shows all runs of recursivePartition $n = 18$ to $n = 24$ outperformed K-means and K-means with perturbations for $k = 30$, and recursivePartition $n = 20$ to $n = 24$ produced identical matches up to a permutation of the true 30 clusters present in the data.

Table 6: Results of K-means and recursivePartition on noisy Artificial Data Data contains 30 real clusters of minimum 18 and maximum 20 nodes

Algorithm	Parameter	RI	JI	Num Clusters
K-means	$k = 28$	.9831	.6512	28
K-means	$k = 29$	.9836	.6573	29
K-means	$k = 30$	.9868	.7031	30
K-means	$k = 31$	.9862	.6961	31
K-means	$k = 32$	.9878	.7183	32
K-means perturb	$k = 28$	.9842	.6677	28
K-means perturb	$k = 29$	.9879	.7267	29
K-means perturb	$k = 30$	.9851	.6796	30
K-means perturb	$k = 31$	.9893	.7469	31
K-means perturb	$k = 32$	.9892	.7424	32
recursivePartition	$n = 18$	.9927	.7695	49
recursivePartition	$n = 19$	.9964	.8871	39
recursivePartition	$n = 20$	1.0000	1.0000	30
recursivePartition	$n = 21$	1.0000	1.0000	30
recursivePartition	$n = 22$	1.0000	1.0000	30
recursivePartition	$n = 23$	1.0000	1.0000	30
recursivePartition	$n = 24$	1.0000	1.0000	30

Artificial data containing 600 nodes in 30 clusters of size 20 was generated with noise added using the gen_cluster_graph function [4]. Again, $k = 30$ was used as the starting K-means parameter because the data contained 30 known clusters. A starting recursivePartition parameter of $n = 20$ was selected because all clusters were generated as uniform size 20. Table 7 shows all runs of recursivePartition $n = 20$ to $n = 25$ outperformed K-means and K-means with perturbations for $k = 30$, producing identical matches up to a permutation of the true 30 clusters present in the data.

Table 7: Results of K-means and recursivePartition on noisy Artificial Data
Data contains 30 real clusters of 20 nodes

Algorithm	Parameter	RI	JI	Num Clusters
K-means	$k = 28$	.9824	.6428	28
K-means	$k = 29$	.9857	.6880	29
K-means	$k = 30$	.9843	.6655	30
K-means	$k = 31$	.9868	.7065	31
K-means	$k = 32$	.9852	.6766	32
K-means perturb	$k = 28$	.9852	.6827	28
K-means perturb	$k = 29$	.9836	.6556	29
K-means perturb	$k = 30$	.9847	.6718	30
K-means perturb	$k = 31$	.9868	.7063	31
K-means perturb	$k = 32$	.9863	.6976	32
recursivePartition	$n = 19$	.9891	.6574	60
recursivePartition	$n = 20$	1.0000	1.0000	30
recursivePartition	$n = 21$	1.0000	1.0000	30
recursivePartition	$n = 22$	1.0000	1.0000	30
recursivePartition	$n = 23$	1.0000	1.0000	30
recursivePartition	$n = 24$	1.0000	1.0000	30
recursivePartition	$n = 25$	1.0000	1.0000	30

6 Conclusions

As demonstrated in section 5, recursivePartition is capable of producing clusters with comparable or higher levels of accuracy than the K-means algorithm with the known number of true clusters present in the data. The flexibility of recursivePartition in producing accurate results over a range of parameter values helps to combat one of the major drawbacks of the K-means algorithm, namely the difficulty in choosing the correct k value for the dataset. Although where to begin selecting an n value for recursivePartition is as unknown as which k value to choose for K-means and our method of dividing the total number of nodes by the number of clusters present in the data will not work because the number of clusters in a meaningful dataset is likely unknown, recursivePartition is more forgiving with its parameter value. Because recursivePartition has the user specify a maximum cluster

size rather than the number of clusters to return, it allows the algorithm to have some influence on what the appropriate number of clusters would be. For instance, in table 7 recursivePartition returned 30 clusters for the range of $n = 20$ to $n = 25$. Flexibility like that could allow a user to run fewer repetitions of recursivePartition than K-means to find the appropriate number of clusters for the data. With this flexibility does come a danger of setting too low a maximum cluster size and thus producing more clusters than desirable. This is also demonstrated in table 7 where the parameter $n = 19$ causes recursivePartition to produce 60 clusters rather than the desired 30 clusters produced by $n = 20$ to $n = 25$. While this danger of producing too many/too small clusters is something to take care to avoid, it is also a threat with K-means if an inappropriate k value is chosen. The flexibility in recursivePartition's ability to produce highly accurate clusters over a range of parameter values can help to reduce the overall number of runs necessary to produce an appropriate clustering.

6.1 Future Work

Future work based on this experiment shows promise for producing interesting results. In line with the original goals for this project, the recursivePartition algorithm could be implemented in parallel. Once the algorithm has paritioned the initial graph into distinct subgraphs (as many as the user wants threads), each of these subgraphs partitioned independently on its own thread using the recursivePartition algorithm. The speedup of that parallel version of the algorithm could then be studied. Another continuation of this project would be a deeper study into how it performs on different types of data, including those with drastically differing sized clusters or clusters with widely-ranging densities. Further work should also include methods for determining an appropriate guess for the input value n given that the actual number of clusters present in the data is unknown. Specifically, further research into setting an appropriate n value should explore how high the n value can be pushed before the accuracy suffers. The maximum cluster size cannot be so large that it does not force enough levels of recursion to produce reasonably sized clusters. It would seem that n must not exceed the sum of the sizes of the two largest clusters in the data, but n needs to be studied further to determine a stricter upper limit for good clusterings.

References

[1] Noga Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[2] Benjamin Auffarth. Spectral graph clustering. Technical report, January 2007.

[3] F.R.K. Chung. *Spectral graph theory*. Regional conference series in mathematics. Published for the Conference Board of the mathematical sciences by the American Mathematical Society, 1997.

[4] D. Dunlavy. gen_cluster_graph matlab function, 2009. Sandia National Laboratories.

[5] D. Dunlavy. jaccard_index matlab function, 2009. Sandia National Laboratories.

[6] D. Dunlavy. rand_index matlab function, 2009. Sandia National Laboratories.

[7] D. Dunlavy. kmeans_dmd matlab function, 2010. Sandia National Laboratories.

[8] D. Dunlavy. Sectral clustering of data vectors, February 2011. Sandia National Laboratories.

[9] D.C. Lay. *Linear Algebra and Its Applications*. Pearson Education, 2002.

[10] Ulrike Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[11] B. Mohar. *Some applications of Laplace eigenvalues of graphs*. Preprint series // Institute of Mathematics, Physics and Mechanics, Department of Mathematics, University of Ljubljana. Univ. of Ljubljana, Inst. of Mathematics, Physics and Mechanics, Dep. of Mathematics, 1997.

[12] Daniel A. Spielman and Shang Teng. Spectral partitioning works: Planar graphs and finite element meshes. Technical report, Berkeley, CA, USA, 1996.

[13] P.N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson International Edition. Pearson Addison Wesley, 2006.

[14] Deepak Verma and Marina Meila. A comparison of spectral clustering algorithms. *University of Washington Tech Rep UWCSE030501*, 1(03-05-01):1–18, 2003.

Appendix A Cut Criteria

All implemented cut criteria are based on the work of [10].
Notation:

$$s(A,B) = \sum_{i \in A} \sum_{j \in B} w_{ij} = cut(A,B) \tag{6}$$

is the similarity between the two non-empty mutually disjoint sets A and B.

$$d_A = \sum_{i \in A} d_i \tag{7}$$

is the degree of node i

$$\mid A \mid = \mid B \mid \tag{8}$$

is number of vertices in A,B.

$$vol(A) = \sum_{i \in A} d_i \tag{9}$$

is the volume of the subgraph A.

Various cut criteria have been developed in order to maximize within cluster similarities while minimizing the number of edges cut between clusters. The following are commonly used cut criteria that have been developed.
Ratio cut:

$$J_{RCut}(A,B) = \frac{s(A,B)}{\mid A \mid} + \frac{s(A,B)}{\mid B \mid} = \sum_{i=1}^{k} \frac{cut(A_i, \overline{A_i})}{\mid A_i \mid} \tag{10}$$

Note that Ratio cut does not take into account the within cluster similarities.

Normalized cut:

$$J_{NCut}(A,B) = \frac{s(A,B)}{d_A} + \frac{s(A,B)}{d_B} \tag{11}$$

$$= \frac{s(A,B)}{s(A,A) + s(A,B)} + \frac{s(A,B)}{s(B,B) + s(A,B)} \tag{12}$$

$$= \sum_{i=1}^{k} \frac{cut(A_i, \overline{A_i})}{vol(A_i)} \tag{13}$$

$$= cut(A,B)\left(\frac{1}{vol(A)} + \frac{1}{vol(B)}\right) \tag{14}$$

MinMax cut:

$$J_{MMC}(A,B) = \frac{s(A,B)}{s(A,A)} + \frac{s(A,B)}{s(B,B)} \tag{15}$$

Min Cut:

$$J_{MinCut}(A,B) = min(cut(A_1,...,A_k)) = min(\sum_{i=1}^{k} cut(A_i, \overline{A_i})) \tag{16}$$

where

$$cut(A,\overline{A}) = \sum_{i \in A_k j \in \overline{A_k}} w_{ij} \tag{17}$$

Appendix B Programs

B.1 recursivePartition.m

```
% Overarching method face to allow for various algorithms to be
% used in partitioning. The desired algorithm is specified via
% the 3rd paramter, a String denoting the name of the algorithm
% to use. This method then calls the appropriate partitioning
% method where all necessary variables are assigned.

function [partitions, index] = recursivePartition(A,n,criteria)

[ignore, A] = preprocess2(A);
if strcmpi(criteria, 'normalized')
    [partitions, index] = recursiveNormalized(A,n);
elseif strcmpi(criteria, 'ratio')
    [partitions, index] = recursiveRatio(A,n);
elseif strcmpi(criteria, 'min')
    [partitions, index] = recursiveMinCut(A,n);
elseif strcmpi(criteria, 'minmax')
    [partitions, index] = recursiveMinMax(A,n);
elseif strcmpi(criteria, 'bcut')
    [partitions, index] = recursiveBCut(A,n);
else
    error('Unknown partitioning criteria');
end
```

B.2 preprocess2.m

```
% Preprocessing of the adjacency matrix A of a graph to produce
% a symmetric matrix, VSsims, of the cosine similarities of
% V, the right singular vector of A, and S, the singular values
% of A. VSsims is then thresholded into a sparse symmetric
% matrix, VSsims_t,  by eliminating those values of VSsims which
% are effectively 0. This implementation uses a threshold of 0.2

function [VS, VSsims_t] = preprocess2(A)

t = 0.2;
k = 30;

[num_terms,num_documents] = size(A);

[U,S,V] = svds(A,k);
VS = V*S;

for j = 1:num_documents
    VS(j,:) = VS(j,:) / norm(VS(j,:),2);
end
VSsims = VS*VS';
VSsims_thresh = VSsims > .2;
VSsims_new = VSsims.*VSsims_thresh;
VSsims_t = sparse(VSsims_new);
```

B.3 recursiveRatio.m

```
% Overhead method with parameters limited to the essential
% information required from the user in recursivePartition.m.
% This function appropriately assigns all other necessary
% parameters for method recursiveRatioSort

function [partitions,index] = recursiveRatio(A,n)

sizeA = size(A,1);
partitions = zeros(2,1);
i = 1;
temp = 0;
index = zeros(sizeA,1);
index(1:sizeA) = 1:sizeA;
constant = 0;
j = 1;

[sizepartition1,sizepartition2,partitions,i,index,constant] = ...
    recursiveRatioSort(A,n,partitions,i,temp,index,constant,j);

ind = find(index==0);
amt = numel(ind);
for(k = 0:amt);
    [row,col] = find(index==0,1,'first');
    index(row,col) = index(row,col-1);
end
num = numel(partitions);
if partitions(num) > sizeA;
    partitions = partitions(1:num-1);
end
```

B.4 recursiveRatioSort.m

```
% Function to perform a recursive partitioning of a data matrix,
% A, associated with a graph until partitions are reduced to a
% user-specified maximum size, n. Initialization of all other
% parameters is handled in recursiveRatio.m

function [sizepartition1,sizepartition2,partitions,i,index,...
    constant] = recursiveRatioSort(A,n,partitions,i,temp,...
    index,constant,j)

j = j+1;

[p,g1,g2] = fiedlerRatioSort(A);
[index] = updateVector(p,index,constant,j);
sizepartition1 = size(g1,1);
sizepartition2 = size(g2,1);
if sizepartition1 > n && sizepartition2 > n
    boolean1 = 0;
    boolean2 = 0;

    [i,partitions,temp] = boolean(partitions,boolean1,...
        boolean2,i,sizepartition1,sizepartition2,temp,n);

    [sizepartition1,sizepartition2,partitions,i,index,...
        constant] = recursiveRatioSort(g1,n,partitions,...
        i,temp,index,constant,j);

    [constant] = find(index(:,j+1)==0,1,'first');
    constant = constant-1;

    [sizepartition1,sizepartition2,partitions,i,index,...
        constant] = recursiveRatioSort(g2,n,partitions,...
        i,temp,index,constant,j);

elseif sizepartition1 > n
    boolean1 = 0;
    boolean2 = 1;

    [i,partitions,temp] = boolean(partitions,boolean1,...
        boolean2,i,sizepartition1,sizepartition2,temp,n);

    [sizepartition1,sizepartition2,partitions,i,index,...
        constant] = recursiveRatioSort(g1,n,partitions,...
```

```
        i,temp,index,constant,j);

    [i,partitions,temp] = boolean(partitions,boolean1,...
        boolean2,i,sizepartition1,sizepartition2,temp,n);

elseif sizepartition2 > n;
    boolean1 = 1;
    boolean2 = 0;
    constant = constant + sizepartition1;

    [i,partitions,temp] = boolean(partitions,boolean1,...
        boolean2,i,sizepartition1,sizepartition2,temp,n);

    [sizepartition1,sizepartition2,partitions,i,index,...
        constant] = recursiveRatioSort(g2,n,partitions,...
        i,temp,index,constant,j);
else
    boolean1 = 1;
    boolean2 = 1;

    [i,partitions,temp] = boolean(partitions,boolean1,...
        boolean2,i,sizepartition1,sizepartition2,temp,n);

   % boolean1 = 0;
    %boolean2 = 0;
end
```

B.5 laplacian.m

```
% Function to produce the corresponding Laplacian matrix, L,
% from the adjacency matrix, A, of an undirected graph.
% L is the laplacian matrix given by L = D - A where D is
% the diagonal matrix of the degree of each node ie the
% sum of each row or column of A

function L = laplacian(A)

D = diag(sum(A));
L = D - A;
```

B.6 fiedler.m

```
% Function to generate the Fiedler value and corresponding
% Fiedler vector (second smallest eigenvalue and eigenvector)
% for the given Laplacian matrix L. d is the Fiedler value, v
% is the sorted Fiedler vector

function [v d] = fiedler(L)

[V D] = eigs(L,2,'sa');
if D(2,2) < .5
    d = D(2,2);
    v = V(:,2);
else
    [V D] = eigs(L,size(L,1)-1,'sa');
    [row,col] = find(floor(D)>0,1,'first');
    d = D(row,row);
    v = V(:,row);
end
```

B.7 cut.m

```
% Function to calculate cut(A,B), ie the set of edges between two
% disjoint sets A,B both in the graph I, where A+B=I, the
% complete graph.
% @precondition: subset A is the first portion of the graph,
%       the graph, sortedI, and subset B is the distinct second
%       portion of sortedI.
%       A and B are square symmetric adjacency matrices.

function edgeTotal = cut(sortedI,sizeA,sizeI)


discard = sortedI(sizeA+1:sizeI, 1:sizeA);
edgeTotal = sum(sum(discard));
```

B.8 ratioCut.m

```
% Function implementing the Ratio Cut algorithm which tends
% toward partitions of an equal or nearly equal size.
% RatioCut = Cut(A,B){1/(Vol(A)*Vol(B))}
% where RatioCut is minimized. Input is sortedI = I(p,p), the
% graph I sorted based on the sorted Fiedler vector

function [A,B] = ratioCut(sortedI)

numRows = size(sortedI,1);
cutVector = zeros(1,1);

breakpoint = floor(sqrt(numRows));

for j = breakpoint:numRows-breakpoint;
    cutAB = cut(sortedI,j,numRows);
    cutVector(j-breakpoint+1) = (cutAB/j) + (cutAB/(numRows-j));
end

[ignore cutPoint] = min(cutVector);
cutPoint = cutPoint + breakpoint -1;
A = sortedI(1:cutPoint,1:cutPoint);
B = sortedI(cutPoint+1:numRows, cutPoint+1:numRows);
```

B.9 minMaxCut.m

```
% Function to calculate the MinMax Cut of a graph I
% MinMaxCut(A,B) = ((Cut(A,B)/Cut(A,A)) + (Cut(A,B)/Cut(B,B)))
% where MinMaxCut(A,B) is minimized.
% Input is sortedI = I(p,p), the graph I sorted based on the
% sorted Fiedler vector

% Note: Volume portion masked whenever Cut(sortedI,A,B) = 0, but
% it appears the first instance (if any) of 0 may provide the
% optimal cut because that is where the volumes are most
% comparable (among all subsets where Cut(sortedI,A,B) = 0).
% This function operates under the assumption that the first
% instance of 0 is the optimal cut among all 0s.

function [A,B] = minMaxCut(sortedI)

numRows = size(sortedI,1);
cutVector = zeros(1,1);

breakpoint = floor(sqrt(numRows));
for i = 1:breakpoint
    cutVector(i) = 1000000;
end

for j = breakpoint:numRows-breakpoint % ensure nonempty partition
    A = sortedI(1:j,1:j);
    B = sortedI(j+1:numRows,j+1:numRows);
    cutAB = cut(sortedI,j,numRows);
    cutVector(j) = ((cutAB/volume(A)) + (cutAB/volume(B)));
end

[ignore cutPoint] = min(cutVector);
A = sortedI(1:cutPoint,1:cutPoint);
B = sortedI(cutPoint+1:numRows,cutPoint+1:numRows);
```

B.10 nCut.m

```
% Function to calculate the Normalized Cut of a graph I.
% From the Verma and Meila paper:
% NCut(A,B) = Cut(A,B){(1/Vol(A)) + (1/Vol(B))}
% where NCut(A,B) is minimized.
% Input is sortedI = I(p,p), the graph I sorted based on the
% sorted Fiedler vector

% Note: Volume portion masked whenever Cut(sortedI,A,B) = 0, but
% it appears the first instance (if any) of 0 may provide the
% optimal cut because that is where the volumes are most
% comparable (among all subsets where Cut(sortedI,A,B) = 0).
% This function operates under the assumption that the first
% instance of 0 is the optimal cut among all 0s.

function [A,B] = nCut(sortedI)

numRows = size(sortedI,1);
cutVector = zeros(1,1);

breakpoint = floor(sqrt(numRows));
for i = 1:breakpoint
    cutVector(i) = 1000000;
end

for j = breakpoint:numRows-breakpoint % ensure nonempty partition
    A = sortedI(1:j,1:j);
    B = sortedI(j+1:numRows,j+1:numRows);
    cutVector(j) = cut(sortedI,j,numRows) *...
        ((1/volume(A)) + (1/volume(B)));
end

[ignore cutPoint] = min(cutVector);
A = sortedI(1:cutPoint,1:cutPoint);
B = sortedI(cutPoint+1:numRows,cutPoint+1:numRows);
```

B.11 minCut.m

```
% Method to calculate the Min Cut of a graph I:
% MinCut(A,B) = Cut(A,B)
% where MinCut is minimized.
% Input is sortedI = I(p,p), the graph I sorted based on the
% sorted Fiedler vector

function [A,B] = minCut(sortedI)

numRows = size(sortedI,1);
cutVector = zeros(1,1);

breakpoint = floor(sqrt(numRows));
for i = 1:breakpoint
    cutVector(i) = 1000000;
end

for j = breakpoint:numRows-breakpoint % ensure nonempty partition
    cutAB = cut(sortedI,j,numRows);
    cutVector(j) = cutAB;
end

[ignore cutPoint] = min(cutVector);
A = sortedI(1:cutPoint,1:cutPoint);
B = sortedI(cutPoint+1:numRows,cutPoint+1:numRows);
```

B.12 updateVector.m

```
% Function to update the universal ordering vector of the
% nodes at each level of recursion.
% constant is the constant to add to the elements of
% vector p in order to update the vector index properly.
% constant is related to the size of the partition

function [index] = updateVector(p,index,constant,j)
sizep = size(p,1);
constp = p + constant;
for k = 1:sizep;
    index(constant+k,j) = index(constp(k),j-1);
end
ind = find(index(1:constant,:)==0);
amt = numel(ind);
for k = 0:amt;
    [row,col] = find(index(1:constant,:)==0,1,'first');
    index(row,col) = index(row,col-1);
end
```

B.13 boolean.m

```
% Function to update the vector indexing the start of partitions

function [i,partitions,temp] = boolean(partitions,boolean1,...
    boolean2,i,sizepartition1,sizepartition2,temp,n)

if i == 1;
    partitions(1) = 1;
end
if boolean1 == 1 && boolean2 == 1;
    i = i+1;
    partitions(i) = partitions(i-1) + sizepartition1;
    i = i+1;
    partitions(i) = partitions(i-1) + sizepartition2;
elseif boolean1 == 1; % && boolean2 == 0;
    i = i+1;
    partitions(i) = partitions(i-1) + sizepartition1;
elseif boolean2 == 1; % && boolean1 == 0;
    if sizepartition1 > n
        temp = sizepartition2;
    else
        if temp > 0
            i = i+1;
            partitions(i) = partitions(i-1) + temp;
            temp = 0;
        end
    end
end
```