Aalborg University
Department of Computer Science

Graph Clustering with an Emphasis on Algorithms Employing the Commuting Times Distance.

Master i IT – Softwarekonstruktion, Masters Projekt 2013

Rodney Michael Miles 20120423 _____

Andrew Bernard Lannie 20120425 _____

Mentor: Manfred Jaeger

06-06-2013

Number of pages: 95
Number of words: 34.718

1. Abstract

It has for some time been found useful to represent networks of various kinds by graphs. The intention is to simplify the data in order to focus on relations between objects in the network represented by edges joining the corresponding vertices. Perhaps this geometric visualization of the reduced data will reveal patterns which might not be seen in other representations.

One of the techniques increasingly used is Graph Clustering in which an attempt is made to find groups of vertices with especially tight bonds. Graph Clustering methods have been used successfully in a number of areas but perhaps especially in the fields of social networks (e.g. networks of scientific collaboration) and biology (e.g. protein-protein interaction networks)

In various ways, the Laplacian matrix of a graph has been found to have a strong relationship with the clusters in that graph. We implement a distance function, the Commuting Times distance, which is simply derived from the Moore-Penrose pseudoinverse of the Laplacian matrix and create an environment in which it may be applied together with distance based clustering methods to cluster graphs. The clustering methods employed are the K-Medoids algorithm and Hierarchical Clustering techniques.

We test these algorithms together with the Commuting Times distance on a number of computer-generated graphs, on two datasets where the cluster structure is well known and on one dataset where it is unknown.

We compare our results with the results of the classic Girvan-Newman algorithm on the same datasets.

Contents

2. Introduction

Here we attempt a brief introduction to the project structured as a brief guide to the report.

We begin in Section 3 with a brief introduction to fairly wide-ranging background material. There are sections on basic definitions in graph theory, on metrics and distance functions, on basic concepts in linear algebra, on distance measures on graphs and on clustering methods using distances.

In Section 4, we attempt to place our efforts in the wider context of graph clustering, describing concepts and methods allied in some way to the Commuting Distance metric and the K-Medoids and Hierarchical Clustering methods which have been the focus of our programming and testing. We also discuss briefly some of the unresolved ambiguities which still characterize this relatively young field of study and which present the most obvious challenges in its future development.

Section 5 focuses on the software constructed to implement and test the Commuting Times distance together with the K-Medoids algorithm and with Hierarchical Clustering. We have attempted to describe both the structure of our own Java code and how this has been merged together with readily available software frameworks whose focus is Graph Representation (JUNG), Data Mining (WEKA) and Linear Algebra (COLT and mtj).

In Section 6, we present the datasets on which we have conducted tests, both computer-generated and “real-life” datasets and the results of the actual testing on these datasets are described in Section 7.

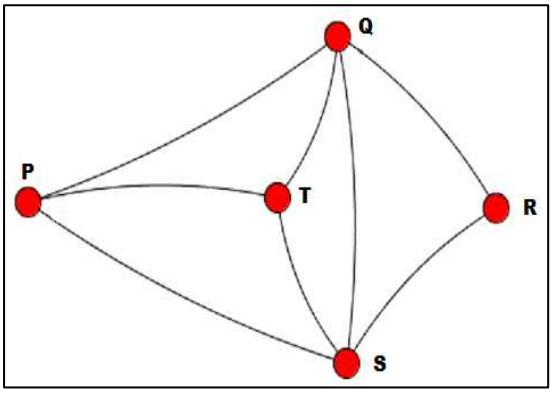
In the final section, Section 8, we summarize our main conclusions.

3. Basic Concepts: Graphs and Clustering

3.1 Graph Definitions

ACKNOWLEDGEMENT: Many of the definitions in this section are either taken from or lean heavily upon (Wilson, 1972)

In layman's terms, a graph is something which may be represented diagrammatically by means of points and lines. One example from everyday life is an electrical network in which the wires may be represented by lines and the electrical objects connected by the wires may be represented by points. A second example is a road map in which the roads are represented by lines, and junctions and intersections are represented by points. At a slightly more abstract level, a graph may be formed from "connections" between people, the people being represented by points and the connections between them by lines. The "connections" could e.g. be blood relationships or the existence of correspondence, possibly electronic, or a relationship within a social network such as Facebook, Twitter or LinkedIn.



In the mathematical discipline of graph theory, the points are called **vertice**s (or **node**s) and the lines are referred to as **edge**s. A **loop** is an edge connecting a vertex to itself. A **simple graph** is a graph containing no loops and with at most one edge between any pair of vertices (see Figure 1).

Figure 1: An example of a simple graph

Formally, the definitions of a (simple) graph and subgraph are:-

A **(simple) graph** is a pair (V(G),E(G)), where V(G), the vertex-set of G, is a non-empty finite set of elements called vertices or nodes and E(G), the edge-set of G, is a finite set of unordered pairs of distinct elements of V(G) called edges. Note that within this definition, it is assumed that sets are defined as not containing duplicates (thus eliminating the possibility of multiple edges between a pair of vertices) and the word distinct prohibits loops. To define a digraph, the word unordered must be substituted by ordered.

A **subgraph** of a graph G is a graph whose vertex-set is a subset of V(G) and whose edge-set is a subset of E(G).

Sometimes, it is appropriate for an edge to be given a direction, e.g. if the graph is representing a road system which includes one-way streets or a body of correspondence in which it is considered important to distinguish between the sender and the receiver(s). A graph in which the edges have direction is known as a **directed graph** or **digraph**. If it is wished to emphasise that this is not the case, the graph may be explicitly called **undirected**. Note that in the case where a graph represents a road-system including one-way streets,

an ordinary "two-way" street connecting two points may be represented by two edges between the equivalent vertices, one in each direction. In a digraph, the edges are often referred to as **arcs**. An edge joining two vertices P and Q in a graph is normally written as {P,Q} whilst the corresponding arc in a digraph is normally written as (P,Q).

In a graph or digraph, it may be appropriate to assign to each edge a non-negative real number which is called the **measure** or the **weight of the edge**. If the assignment is made for one edge, it will be made for them all, an appropriate default value being selected if necessary for edges which may not have an obvious assignment. In this case, the graph is referred to as a **weighted graph or digraph**. An **unweighted** graph or digraph may be considered a weighted graph or digraph in which each edge or arc has weight 1.

Unless otherwise stated the graphs in this report are undirected.

A **path** in a graph is a sequence of edges, one following on after another. A path may also be represented by the sequence of vertices encountered by traversing it. In the graph below P -> Q -> R is a way of getting from vertex P to vertex R and is a path given by the sequence of edges {P,Q}, {Q,R}. The edge {P,Q} is said to **join** the vertices P and Q, and P and Q are said to be **incident** to the edge {P,Q}. The **length** of a path is the number of edges in the edge sequence. A path which ends at the vertex at which it started is called a **circuit**. Given any two vertices with at least one path between them, a **shortest path** between them is a path between them containing the least number of edges. (For a **weighted graph**, the **shortest path** is defined to be a path where the sum of the weights on the edges along the path is minimal.) A graph in which there is a path between any two vertices is called a **connected graph**.

	P	Q	R	S	T
P	0	1	0	1	1
Q	1	0	1	1	1
R	0	1	0	1	0
S	1	1	1	0	1
T	1	1	0	1	0

Matrix 1: The adjacency matrix of the graph in Figure 1

Two vertices P and Q in a graph are said to be **adjacent** if the graph contains the edge {P,Q}. The **degree** of a vertex P is the number of edges to which P is incident. A vertex of degree zero is called an **isolated vertex**. If G is a graph with vertex-set $\{v_1, ..., v_n\}$, the **adjacency matrix** (Matrix1) of G (corresponding to the given labelling of the vertices) is the n x n matrix $A = (a_{i,j})$, in which $a_{i,j}$ is the number of edges in G joining v_i and v_j. The adjacency matrix of a digraph with the same vertex set is similarly defined, though in this case, a_{ij} is the number of arcs (v_i, v_j), i.e. from v_i to v_j. Note that the adjacency matrix of a (n undirected) graph is symmetric ($a_{i,j} = a_{j,i}$ for all i and j), whilst this is not necessarily the case for a digraph.

	P	Q	R	S	T
P	3	0	0	0	0
Q	0	4	0	0	0
R	0	0	2	0	0
S	0	0	0	4	0
T	0	0	0	0	3

Matrix 2: The degree matrix of the graph in Figure 1

	P	Q	R	S	T
P	3	-1	0	-1	-1
Q	-1	4	-1	-1	-1
R	0	-1	2	-1	0
S	-1	-1	-1	4	-1
T	-1	-1	0	-1	3

Matrix 3: The Laplacian matrix of the graph in Figure 1

The **degree matrix** (Matrix 2) of the graph G with vertex-set $\{v_1, ..., v_n\}$ is the n x n diagonal matrix $D = (d_{ij})$ in which $d_{ij} = 0$, if i != j, and d_{ii} is the degree of vertex v_i. From the degree matrix D and the adjacency matrix A of a graph, the **Laplacian matrix** (Matrix 3) L of the graph may be defined by

$L = D - A$

The **Laplacian matrix** of a **digraph** may be similarly defined though in this case, it is necessary to be more specific about the degree of a vertex. For the vertex v_i in G, the **out-degree** of v_i is the number of arcs of the form (v_i,v_j), arcs pointing away from v_i and the **in-degree** of v_i is the number of arcs of the form (v_j,v_i), arcs pointing towards v_i. Either the in-degrees or the out-degrees may be used in the degree matrix D which is subsequently used to define the Laplacian matrix.

The **normalized Laplacian matrix** (Matrix 4) is created from the Laplacian matrix by dividing entry (i,j) of the Laplacian matrix by $(\deg(v_i) * \deg(v_j))^{½}$. Alternatively, the **normalized Laplacian matrix** is defined as $L = (l_{ij})$, where:-

$l_{ij} = 1$, if i = j and the degree of v_i, $\deg(v_i)$, != 0,

$l_{ij} = -1/(\deg(v_i) * \deg(v_j))^{½}$, if i != j,

$l_{ij} = 0$, otherwise.

	P	Q	R	S	T
P	1	-0.288675	0	-0.288675	-0.333333
Q	-0.288675	1	-0.333333	-0.25	-0.288675
R	0	-0.333333	1	-0.353553	0
S	-0.288675	-0.25	-0.353553	1	-0.288675
T	-0.333333	-0.288675	0	-0.288675	1

Matrix 4: The normalised Laplacian matrix of the graph in Figure 1

3.2 Distance Metrics

Certain families of clustering methods, e.g., nearest neighbour clustering and hierarchical clustering, are reliant on a distance metric. These clustering methods will be among those later employed to cluster graphs so in our introduction of the general concepts, we include a formal definition and short discussion of distance functions.

The formal definition of a distance function, in this case extracted from Wikipedia [1] is as follows:-

[1] Source: http://en.wikipedia.org/wiki/Metric_%28mathematics%29

"A **metric** on a set X is a function (called the *distance function* or simply **distance**)

$d : X \times X \rightarrow \mathbf{R}$

(where **R** is the set of real numbers). For all x, y, z in X, this function is required to satisfy the following conditions:

1. $d(x, y) \geq 0$ (non-negativity, or separation axiom)
2. $d(x, y) = 0$ if and only if $x = y$ (identity of indiscernibles, or coincidence axiom)
3. $d(x, y) = d(y, x)$ (symmetry)
4. $d(x, z) \leq d(x, y) + d(y, z)$ (subadditivity / triangle inequality)"

Sometimes the term **metric** is used for the definition above and the term **distance function** is used for a function which satisfies a subset of the four conditions above. Since we need to make this distinction, we will adopt this convention.

Metrics are considered to be measures of dissimilarity between data objects. It is possible to define measures of similarity between data objects though these seem to be less rigorously defined than distance metrics. A semi-formal attempt at defining a similarity measure might be

"A **similarity measure** on a set X is a function

$s : X \times X \rightarrow \mathbf{R}$

(where **R** is the set of real numbers). For all x, y in X, this function is required to satisfy the following conditions:

1. $1 \geq s(x, y) \geq 0$ (non-negativity, or separation axiom)
2. $s(x, y) = 1$ if and only if $x = y$ (identity of indiscernibles, or coincidence axiom)
3. $s(x, y) = s(y, x)$ (symmetry)"

In relation to the definition of a distance metric, the triangle inequality has been removed and a constraint has been added that the value lies in the range 0 to 1 with 1 representing equality.

3.3 Linear Algebra

In "3.1 Graph Definitions", the degree, adjacency and Laplacian matrices of a graph were defined. Certain distance metrics defined on graphs can be expressed in terms of these matrices and we employ these matrices to calculate the distances. In addition, mathematical properties of the matrices are also used to prove that the functions defined are indeed distance metrics and thus form an integral part of the mathematical foundation for some of the graph clustering algorithms whose theory and implementation is subsequently discussed. In this section, the relevant concepts of matrix theory and pertinent properties of the particular graph matrices are introduced.

Trivially, degree, adjacency and Laplacian matrices are **square** as the number of rows and columns in each is equal to the number of vertices in the graph. For undirected graphs, all three matrices are **symmetric** ($a_{ij} = a_{ji}$ for all i and j) and the degree and adjacency matrices are **non-negative** ($a_{ij} >= 0$ for all i and j). The degree matrix is an example of a **diagonal** matrix, a matrix in which all the entries outside the main diagonal are zero ($a_{ij} = 0$ if i != j).

An **eigenvector** of a square matrix A is a non-zero vector v which, when multiplied by A, yields itself multiplied by a scalar l called the **eigenvalue** of A corresponding to v; i.e.

$Av = lv$

The **spectrum** of a matrix is the set of its eigenvalues.

The **transpose** A^T of a matrix A is the matrix in which $[A^T]_{ij} = [A]_{ji}$, i.e. the columns of A^T are the rows of A and vice versa. The transpose of a square, symmetric matrix is equal to the matrix itself.

All the matrices derived from graphs are real-valued matrices. For such matrices, a **Hermitian** matrix is a square matrix A which is identical with its own transpose A^T. As such, a Hermetian matrix is a **normal** matrix which is a matrix satisfying the equation $AA^T = A^TA$. A **unitary** matrix U is a matrix satisfying the equation $AA^T = A^TA = I$, the identity matrix and is thus a normal matrix. Real symmetric matrices (and thus our three graph matrices) are Hermetian matrices and thus satisfy the properties of these and normal matrices.

Normal matrices are important because they are the group of matrices to which the **spectral theorem** applies:-

A matrix A is normal if and only if there are diagonal and unitary matrices D and U respectively, so that

$A = UDU^*$.

The entries of the diagonal matrix D are the eigenvalues of A and the columns of U are the eigenvectors of A, the order of the eigenvalues in D corresponding to the order of the columns of U as eigenvectors.

If $v_0 = [1,1,\ldots,1]$, then $Lv_0 = 0$, the zero matrix, for each Laplacian matrix with the same number of columns as v_0 (For each row of L, the product Lv_0 adds the degree of the corresponding vertex from the diagonal to a "-1" for each neighbour of that corresponding vertex). So each Laplacian matrix has at least one zero eigenvalue. In actual fact, the number of zero eigenvalues of a Laplacian matrix corresponds to the number of connected components of the underlying graph.

A square n X n matrix A is **invertible** if and only if there is an n X n matrix B such that $AB = BA = I_n$, the n X n identity matrix. An n X n matrix A is invertible if and only if 0 is not an eigenvalue of A. So no Laplacian matrix is invertible.

The **Moore-Penrose pseudoinverse** of a matrix A is a generalization of the inverse matrix. It is defined and unique for all matrices whose entries are (real or) complex numbers including therefore Laplacian matrices (Proofs of the existence and uniqueness of the Moore-Penrose pseudoinverse for all complex number matrices may be found on Wikipedia [2]. The formal definition of the **Moore-Penrose pseudoinverse** of an m X n matrix A is that it is an n X m matrix A^+ satisfying the following four criteria:-

1. $AA^+A = A$ (AA^+ need not be the general identity matrix, but it maps all column vectors of A to themselves)
2. $A^+AA^+ = A^+$
3. $(AA^+)^* = AA^+$ (AA^+ is Hermitian)

[2] Source: http://en.wikipedia.org/wiki/Proofs_involving_the_Moore%E2%80%93Penrose_pseudoinverse

4. $(A^+A)^* = A^+A$ (A^+A is Hermitian)

A **Singular Value Decomposition (SVD)** of an m X n real matrix A is a factorization of the form

A = UDV*, where

U is an m X m (real or) complex **unitary** matrix (Matrix 5),

D is an m X n **diagonal** matrix (Matrix 6) with **non-negative real** numbers on the diagonal and

V* (the transpose of V) is an n X n real **unitary** matrix (Matrix 7).

	P	Q	R	S	T
P	-0.365148	0	-0.707107	0.408248	0.447214
Q	0.547723	-0.707107	0	0	0.447214
R	-0.365148	0	0	-0.816497	0.447214
S	0.547723	0.707107	0	0	0.447214
T	-0.365148	0	0.707107	0.408248	0.447214

Matrix 5: U in a singular value decomposition UDV* of the Laplacian matrix of the graph in Figure 1

	P	Q	R	S	T
P	5	0	0	0	0
Q	0	5	0	0	0
R	0	0	4	0	0
S	0	0	0	2	0
T	0	0	0	0	0

Matrix 6: D in a singular value decomposition UDV* of the Laplacian matrix of the graph in Figure 1

	P	Q	R	S	T
P	-0.365148	0.547723	-0.365148	0.547723	-0.365148
Q	0	-0.707107	0	0.707107	0
R	-0.707107	0	0	0	0.707107
S	0.408248	0	-0.816497	0	0.408248
T	-0.447214	-0.447214	-0.447214	-0.447214	-0.447214

Matrix 7: V* in a singular value decomposition UDV* of the Laplacian matrix of the graph in Figure 1

A Singular Value Decomposition exists for all real matrices. If A = UDV* is a singular value decomposition of A, then the Moore-Penrose pseudoinverse $A^+ = VD^+U^*$. For the diagonal matrix D, the Moore-Penrose pseudoinverse D^+ is obtained by taking the reciprocal of each non-zero element in the diagonal, leaving the zeros in place and transposing the resulting matrix. If a single value decomposition of the matrix A can be obtained, the formula above can be used to calculate the pseudoinverse A^+.

	P	Q	R	S	T
P	0.235	-0.04	-0.14	-0.04	-0.015
Q	-0.04	0.16	-0.04	-0.04	-0.04
R	-0.14	-0.04	0.36	-0.04	-0.14
S	-0.04	-0.04	-0.04	0.16	-0.04
T	-0.015	-0.04	-0.14	-0.04	0.235

Matrix 8: The Moore-Penrose pseudoinverse of the Laplacian matrix of the graph in Figure 1 given by A+ = VD+U*

We have written Java code to calculate the Moore-Penrose pseudoinverse (Matrix 8) of the Laplacian matrix of a graph. This code is based on an SVD class from the mtj linear algebra software package (See Section "5.3.3 mtj (*m*atrix-*t*oolkits-*j*ava)") which is a class which implements a single value decomposition factorization of a matrix[3].

As we have noted, the number of zero eigenvalues of a Laplacian matrix corresponds to the number of connected components of the underlying graph. The presence of at least one zero eigenvalue prevents a Laplacian matrix from being **positive-definite** but it is **positive-semidefinite** as all the eigenvalues are non-negative.

3.4 Distance Metrics on Graphs

In "Section 3.1 Graph Definitions" some basic concepts of graph theory were introduced. This was followed by a formal definition of distance metrics in "Section 3.2 Distance Metrics" and by some heavy ploughing through the relevant matrix definition and properties in "Section 3.3 Linear Algebra". Some of this introductory material will be revisited at various points in the subsequent exposition but it is necessary background to the topic of this section, distance metrics on graphs.

(Deza & Deza, 2009) is devoted to the study of distance functions and Chapter 15 in particular examines extensively distance functions in graphs. Of the several described in that chapter, we focus on a few, either because of their universality or because of their use together with graph clustering techniques.

[3] NOTE: Our linear algebra code was originally based on the Colt linear algebra software package which includes a SingleValueDecomposition class which superficially provides the same functionality as the SVD class from mtj. When we used the Colt class to calculate the pseudoinverse, many entries in the calculated pseudoinverses were clearly false. We discovered the mtj package and SVD class in the course of our attempts to debug our calculation of the Moore-Penrose pseudoinverse and found that adoption of the factorization provided by this class eliminated the obvious errors in the pseudoinverse matrices. Our code is therefore based on the Colt software with the single exception of the SVD class from mtj. There may be banal errors in the implementation of the SingleValueDecomposition class in the Colt software but the experience made us aware that neither the Single Value Decomposition factorization of a matrix nor the algorithm used to calculate such a factorization is unique.

Before looking at specific examples, it should be noted that distance metrics are normally defined on connected graphs as the distance between a non-connected pair of vertices is likely to be

- undefined or
- defined as infinite thus giving problems with otherwise finite calculations or
- defined as 0 in which case condition 2 of a distance metric, the "identity of indiscernibles", is undermined.

3.4.1 Shortest Path Length

The first and probably most common distance metric defined on graphs is the length of the shortest path between two vertices. This was defined in "Section 3.1 Graph Definitions". The shortest path distance is also known as the **geodesic** distance.

Before graph clustering became a field of study, certain problems (e.g. finding optimal routes) could be expressed in terms of shortest paths in graphs. Shortest path as a distance measure on a graph remains relevant and modern developments have generated new fields of application. For example, Google Maps employs shortest path algorithms to calculate driving directions. Social networks can be analysed using shortest paths to indicate the degrees of separation between members of the network according to particular connecting criteria; e.g. the "friend" relationship on Facebook.

The adjacency matrix A has the nice graph-theoretic property that the $(i,j)^{th}$ entry of A^k contains the number of paths of length k from vertex v_i to vertex v_j for all positive k. It follows that one way though (not necessarily the most efficient) of obtaining the length of the shortest path from v_i to v_j is to calculate increasing powers of the adjacency matrix until the first non-zero $(i,j)^{th}$ entry is observed.

An early algorithm for finding an actual shortest path (as opposed to just the length of it) is described in (Dijkstra, 1959).

Efficient algorithms for computing the shortest path, also on weighted graphs, exist e.g. the A* algorithm described in (Hart, Nilsson, & Raphael, 1968).

3.4.2 Resistance Distance

This distance metric seems to have been first defined in the brief article (Gvishiani & Gurvich, 1987) and a little later independently defined and developed in (Klein & Randic, 1993). The distance metric derives its name and inspiration from electrical network theory. A graph is regarded as an electrical network in which a fixed resistor is imagined on each edge. The resistance distance is then defined as the (effective) resistance between two vertices (when a battery is connected across them). We explain the basic idea with the three very simple graphs in the diagram below.

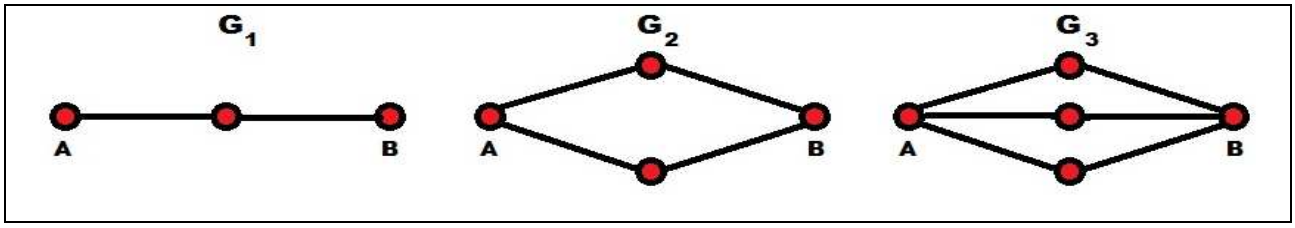


Figure 2: Three graphs with the same conventional graphical distance (Klein & Randic, 1993)

In the following figure, these three graphs have been equipped with batteries and resistors as described above.

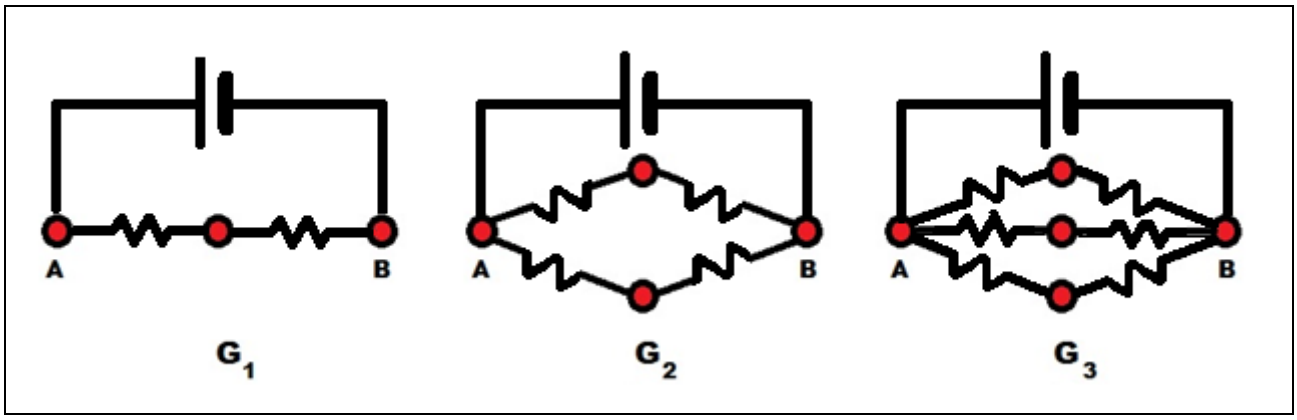


Figure 3: The three graphs of Figure 2 with resistors introduced on each edge, while a battery is linked between the a,b-pair of vertices. (Klein & Randic, 1993)

If the resistors all have a value of 1 (ohm), the resistance distances between vertices a and b in the three original graphs will be:

$\Omega_{ab} = 1 + 1 = 2$ for G_1

$\Omega_{ab} = 1/(½ + ½) = 1$ for G_2

$\Omega_{ab} = 1/(½ + ½ + ½) = 2/3$ for G_3

Section 4 of (Klein & Randic, 1993) is devoted to a proof that the resistance distance satisfies the four criteria of a distance metric. Prior to Section 4, the authors have in COROLLARY A of Section 3 expressed the resistance distance between any two vertices in terms of the graph's Laplacian matrix and what they refer to as the "generalized" inverse of this matrix. It is this expression and known properties of the Laplacian matrix which enable them, in particular, to prove the non-negativity criteria.

3.4.3 Commuting Time distance metric

Consider random walks on a graph G, where at each step the walk moves to a vertex adjacent to the current vertex randomly with uniform probability. The **hitting time** H(u,v) from vertex u to vertex v is the expected number of steps (edges) for a random walk on G beginning at u to reach v for the first time; it is 0 for u = v. The hitting time is not a distance metric because it is not symmetric as the example below illustrates. Starting at v, the random walk can only head in the direction of u although it can take steps back towards v before reaching u. Starting at u, the random walk has the same possibilities for vacillating along the u – v axis but it can also stumble around the complete subgraph of which u is a vertex and v is not. Intuitively, H(u,v) is larger than H(v,u). In fact, it can be shown that H(v,u) has $O(n^2)$ and that H(u,v) has $O(n^3)$.

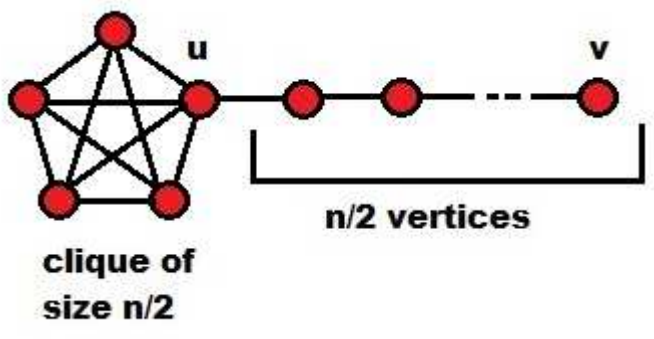


Figure 4: Diagram of a counterexample graph to the claim that the hitting time might be symmetric. (Hopcroft, 2008)

The **commuting time** C(u,v) from vertex u to vertex v is defined by C(u,v) = H(u,v) + H(v,u). The definition of the commuting time clearly overcomes the lack of symmetry preventing the hitting time from being a distance metric and is, in fact, a distance metric. This is proved as an almost trivial consequence of Corollary 2.4 in (Göbel & Jagers, 1974)

Although it is not immediately obvious, the commuting time metric and the resistance distance described in "Section 3.4.2 Resistance Distance" are intimately linked; in fact, they are identical except for a scalar factor. It is e.g. proved as THEOREM 2.1 in (Chandra, Raghavan, Ruzzo, Smolensky, & Tiwari, 1989) that:
For any two vertices u and v in a (n undirected,) connected graph G, the commuting time C(u,v) = 2mR(u,v), where m is the number of edges in G and R(u,v) is the resistance distance.

In this section and the previous section, the discussion has been limited to unweighted graphs. Both distance metrics can, however, be extended to weighted graphs. In the case of resistance distance, the fixed resistor with a value of 1 ohm imagined on each edge would be replaced by a resistor whose value in ohms reflected the weight of the edge. In the case of commuting time, the weights would be translated into probabilities of traversing an edge in a random walk, the heavier the weight, the more likely the edge to be selected as the next step in the walk. In fact, THEOREM 2.2 in (Chandra, Raghavan, Ruzzo, Smolensky, & Tiwari, 1989) is a generalization of THEOREM 2.1 to weighted graphs proving that the resistance distance and the commuting time metric are still related by a scalar factor, although this factor is no longer simply twice the number of edges in the graph.

These two distance metrics have a desirable property which the shortest path distance does not possess. They decrease when the number of paths connecting two vertices increases and when the length of any path decreases; i.e. when communication between the two vertices is facilitated. The more short paths that connect two given vertices, the closer together those vertices are. This is not the case with the shortest path distance which in general does not decrease as connections between nodes are added and which does not capture the property that "strongly connected" nodes are closer than "weakly connected" nodes. This makes the shortest path distance the least suitable of the three to employ in distance-based graph clustering methods. The property explained rather loosely here is formalized for resistance distance as Rayleigh's Monotonicity Law in e.g. (Doyle & Snell, 2000).

Towards the end of "Section 3.3 Linear Algebra", we described some of the difficulties we had encountered when calculating the Moore-Penrose pseudoinverse of the Laplacian matrix. The reason for calculating this

pseudoinverse is that the commuting times can be expressed explicitly in terms of the entries of this matrix. Specifically, if G is a connected graph with vertex-set $\{v_1, ..., v_n\}$, then

$C(v_i,v_j) = 2m(l^+_{ii} + l^+_{jj} - 2l^+_{ij})$,

where m is the number of edges of G and l^+_{ij} is the $(i,j)^{th}$ entry of the Moore-Penrose pseudoinverse L^+ of the Laplacian matrix L of G, corresponding to the given labelling of the vertices (see Matrix 9). How this formula for the commuting times is arrived at can be seen, e.g. in Appendices B and C of (Fouss, Pirotte, Renders, & Saerens, 2007).

	P	Q	R	S	T
P	0	0.475	0.875	0.475	0.5
Q	0.475	0	0.6	0.4	0.475
R	0.875	0.6	0	0.6	0.875
S	0.475	0.4	0.6	0	0.475
T	0.5	0.475	0.875	0.475	0

Matrix 9: The Commuting Times distance matrix (without the factor 2m) for the graph in Figure 1.

3.5 Distance-based clustering methods.

Having established in the previous section that various distance and similarity measures can be defined on graphs, the possibility arises of employing distance-based clustering methods to partition graphs. In this section, such clustering methods are introduced.

3.5.1 K-Means

The environment in which the K-means algorithm may be set to work is a positive integer k (the number of clusters/partitions) and a set X of data points in R^n, n-dimensional Euclidean space. The algorithm itself involves the following steps:-

1. Arbitrarily select an initial k centers $C = \{c_1, c_2,...,c_k\}$.
2. For $1 <= i <= k$, set the cluster C_i to be the set of points in X which are closer to the center c_i than to any of the other centers.
3. For $1 <= i <= k$, update the center c_i to be the center of **mass** of all points in C_i.
4. Repeat steps 2 and 3 until the set of centers C no longer changes (or until the changes are within an acceptable bound).

The K-Means algorithm is guaranteed to converge though not necessarily to a globally optimal solution. It can be slow to converge so, in practice, the algorithm is normally curtailed after a fixed number of iterations if no solution is reached before.

It is the word “closer” in Step 2 of the algorithm which reveals K-Means as a distance-based algorithm as “closer” is defined in terms of a particular distance or similarity function.

If the data set on which a distance function is defined is not a subset of R^n for some n, it can be impossible to define, let alone calculate, the center of mass (The sum of the n-dimensional vectors in C_i divided by the number of vectors in C_i.) in step 3 and this will prohibit the use of the K-Means algorithm. This is the case, e.g. with the shortest path and Commuting Time distance metrics on graphs. If, however, a distance

function is defined in terms of real vectors corresponding to nodes as it is e.g. with Euclidean Commuting Distance (See "Appendix A: Working with the Euclidean Commuting Time Distance and the Measures Presented by The Moore-Penrose Pseudoinverse being a Gram Matrix."), then the center of mass of these vectors can be computed and the K-Means algorithm may be applied.

3.5.2 K-Medoids

The problem with defining or computing the center of mass in Step 3 of the K-Means algorithm can be avoided if Step 3 is replaced by

4 For $1 <= i <= k$, update the center c_i to be the **point** in C_i which minimizes the total distance to all other points in C_i.

The center of any cluster is thus selected at any iteration as one of the points in the data set and the calculations only involve comparing sums of distances between existing points in the data set, all of which are well-defined by the existence of the distance metric.

There can, however, be some ambiguity about what "sum" or "total" means in this context. If the distance metric is Manhattan, "total" will usually be interpreted as an arithmetic sum. If the distance metric is Euclidean, "total" will most often be interpreted as "sum-of-squares". At its most mathematically stringent, the phrase "total distance" in our new Step 3 should be substituted by the phrase "potential function" which will include a definition of "total" appropriate to the underlying distance metric. The phrase "center of mass" used in Step 3 of the K-Means algorithm described in the previous section and our interpretation of it assumes the Euclidean distance metric which is a fair assumption in most cases unless the context explicitly states otherwise. If the K-Means algorithm is used with Manhattan distance, it is most often referred to as the K-Medians algorithm. If the algorithm is used with the cosine similarity measure, it is usually called Spherical K-Means. The term K-Medoids stems from the practice of referring to the cluster centers as **medoids** when they are selected from the data set itself; when the cluster centers are calculated and are not necessarily actual points in the data set, they are normally called **centroids**.

However, to return from the terminology digression, the advantage of K-Medoids being emphasized here is that it is not restricted to an R^n space of attributes on which calculations have to be made in order to obtain a center of mass. The shortest path and commuting distance metrics on graphs may be freely used together with the K-Medoids algorithm.

Our implementation of the K-Medoids algorithm, inherited from an earlier project, is the algorithm described in Section 2 of (Park, Lee, & Jun, 2006) apart from a couple of small deviations. We first present the algorithm as it appears in the article and then describe our small variations.

2. Proposed K-medoids algorithm

Suppose that we have n objects having p variables that will be classified into k ($k < n$) clusters (Assume that k is given). Let us define j-th variable of object i as X_{ij} ($i = 1,...,n;\ j = 1,...,p$). The proposed algorithm is composed of the following three steps.

Step 1 : (Select initial medoids)

1-1. Using Euclidean distance as a dissimilarity measure, compute the distance between every pair of all objects as follows:

$$d_{ij} = \sqrt{\sum_{a=1}^{p} (X_{ia} - X_{ja})^2} \quad i = 1,...,n;\ j = 1,...,n \tag{1}$$

1-2. Calculate p_{ij} to make an initial guess at the centers of the clusters.

$$p_{ij} = \frac{d_{ij}}{\sum_{l=1}^{n} d_{il}} \quad i = 1,...,n;\ j = 1,...,n \tag{2}$$

1-3. Calculate $\sum_{i=1}^{n} p_{ij}$ $(j = 1,...,n)$ at each objects and sort them in ascending order. Select k objects having the minimum value as initial group medoids.

1-4. Assign each object to the nearest medoid.

1-5. Calculate the current optimal value, the sum of distance from all objects to their medoids.

Step 2 : (Find new medoids)

Replace the current medoid in each cluster by the object which minimizes the total distance to other objects in its cluster.

Step 3 : (New assignment)

3-1. Assign each object to the nearest new medoid.

3-2. Calculate new optimal value, the sum of distance from all objects to their new medoids. If the optimal value is equal to the previous one, then stop the algorithm. Otherwise, go back to the Step 2.

Algorithm 1: The K-Medoids algorithm (Park, Lee, & Jun, 2006)

Having presented the algorithm, we describe the two variations to it in our own implementation.

- Step 1 of the borrowed algorithm begins with the phrase “Using Euclidean distance as a dissimilarity measure”. Our purpose in this context is to use the K-Medoids algorithm with distance metrics on graphs, initially, the Commuting Time metric but possibly others as well. The selected distance metric/function is used consistently in our implementation of the algorithm, also in the alternative procedure to select an initial group of medoids described next.
- Step 1 of the borrowed algorithm describes a deterministic method of selecting the initial cluster centers or medoids. The idea is to create a matrix of weighted distances which are then summed column-wise to give an intuitive idea of how centrally points are placed in the distribution with respect to the distance measure. The algorithm initially selects the most centrally placed points

which in our case should correspond to the most strongly connected graph vertices. This initialization has been programmed but as an alternative to the standard K-Means initialization so the K-Medoids class may be executed either with this deterministic selection of a set of initial cluster medoids or with an arbitrary starting set of cluster medoids whose selection stems from a random seed value. In the visualization tool described in "Section 5.4.3 The RAGraphClustering Class", the deterministic initialization of medoids corresponds to seed 0.

The algorithm we have implemented is not the most common algorithm for clustering around medoids. This algorithm is the "partitioning around medoids" algorithm which is nicely described and illustrated on Wikipedia's K-Medoids page[4] .

The authors of the article (Park, Lee, & Jun, 2006) from which our implemented algorithm is taken have compared the performance of the algorithm with the Simple K-Means algorithm and with the "partitioning around medoids (PAM)" algorithm. With artificial data generated from multivariate normal distributions and with outliers added to one of three classes, the performance of the three algorithms was measured. The performance of the two K-Medoid algorithms was very similar and significantly better than the performance of the K-Means algorithm. This is perhaps not so surprising as a K-Medoids algorithm is sometimes preferred to a K-Means algorithm even when the distance metric/function does not require it as it is considered more robust against the presence of outliers.

The implemented algorithm has complexity O(nk) which is the same as the complexity of the Simple K-Means algorithm (ignoring the number of iterations), and less than the complexity of the PAM algorithm, which is $O(k(n-k)^2)$; here, n is the total number of instances in the dataset and k is the number of clusters. The expected efficiency of the implemented algorithm was reflected in shorter execution times than the PAM algorithm could achieve.

Unfortunately, our implementation does not take advantage of the O(nk) efficiency of the algorithm. This is because we perform calculations on n x n matrices, which implies a complexity of at least $O(n^2)$. In fact, we believe the determining factor of the complexity is actually the Singular Value Decomposition needed to calculate the Moore-Penrose pseudoinverse of the Laplacian matrix; we do not know how this is implemented but the complexity of Singular Value Decomposition is often given as $O(n^3)$!

Although this is the case, our code is merged with the WEKA and JUNG packages in such a way that that an unlimited number of executions of the K-Medoids algorithm are possible after a single calculation of the Commuting Times distance matrix so in this way, the initial patience of the user may afterwards be rewarded some practical benefits from the O(nk) efficiency of the algorithm!

3.5.3 Hierarchical Clustering

Hierarchical clustering is a method of cluster analysis which seeks to build a hierarchy of clusters. Strategies for hierarchical clustering generally fall into two types:-

- **Agglomerative**: This is a "bottom up" approach. Each data object starts in its own cluster and at each step up the hierarchy, the "closest" pair of clusters is merged.

[4] Source: http://en.wikipedia.org/wiki/K-medoids

- **Divisive**: This is a "top down" approach. All the data objects start in a single cluster and splits are performed recursively as one moves down the hierarchy.

In order to decide where clusters should be merged in the case of an agglomerative strategy or where a cluster should be split in the case of a divisive strategy, a measure of similarity is required between sets of data objects. This is usually achieved by the use of a distance metric between individual objects in the data set and a means of extending this metric to sets of objects. This extension is usually referred to as a link type.

One of the advantages of hierarchical clustering is that it does not require the user to specify an optimal number of clusters. This is a considerable advantage when the number of clusters is not known and when an inappropriate choice might prejudice the workings of the clustering algorithm so as to conceal rather than reveal the "natural" clusters. The challenge with (agglomerative) hierarchical clustering is to examine the whole hierarchy of merges and to identify the significant ones. The results of hierarchical clustering are often displayed in a dendogram as illustrated below.

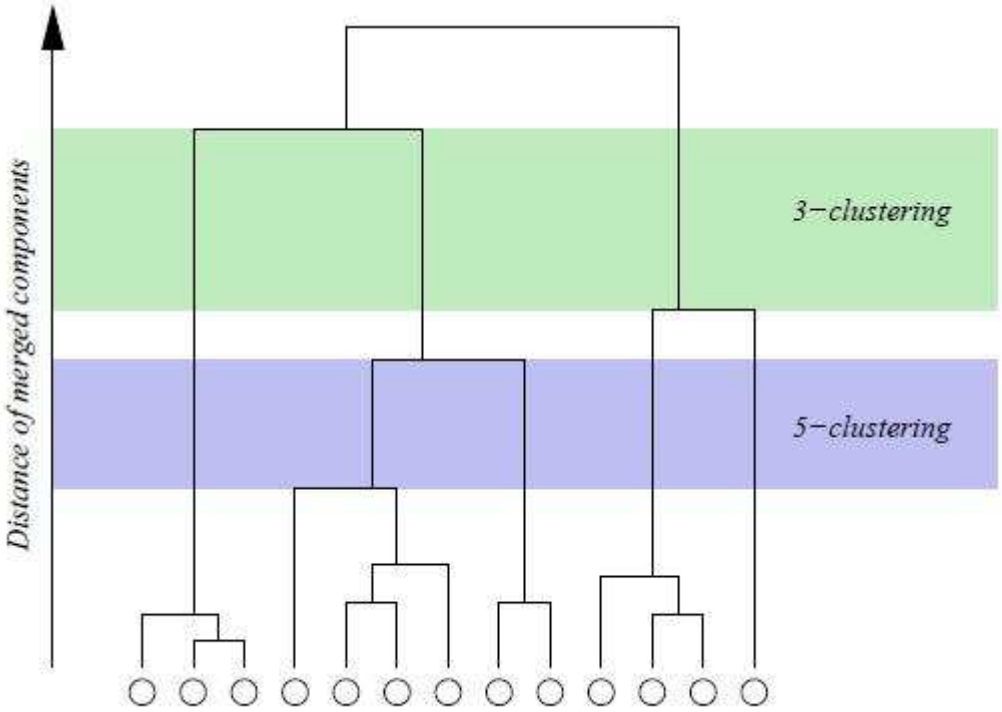


Figure 5: The results of hierarchical clustering shown as a dendogram

The circles at the bottom of the dendogram represent the individual objects in the data set being clustered and the tree structure which forms the body of the dendogram represents the order in which sets of objects are merged, the lower the join of the two branches, the earlier the agglomeration of the two subsets represented by those branches. This is achieved by allowing the y-axis of a dendogram to represent the inter-cluster distance at which two clusters are merged.

In the dendogram illustrated in Figure 5 the largest distance intervals between cluster merges are where 5 clusters are merged into 4 and where 3 clusters are merged into 2. This would suggest that the most significant clusterings for this data set may be 3 and 5 clusters. Generally, the length of the distance interval

corresponding to a specific clustering can be interpreted as a measure for the significance of this particular clustering.

For large data sets or agglomerative clusterings in which it is difficult to see distance intervals which are significantly larger than others, an extra degree of analysis may be needed. In this case, a graph may be drawn plotting the number of merges on the x-axis and the inter-cluster distance at which two clusters were merged on the y-axis. In this case, points at which the gradient of the graph undergoes a rapid change will be points at which to look for significant clusters. (An alternative would be to plot branch length (distance interval between merges) as opposed to node height on the y-axis and specific mergers on the x-axis and to look for areas in the graph where the points become more widely-spaced. This alternative appears to be considered most appropriate when the link type is **neighbour joining** (See the enumeration of link types below.)

Even this second level of distance analysis would be regarded by some as too simplistic or inaccurate an approach to the problem of extracting significant clusters from hierarchical clustering representations. In (Sander, Quin, Lu, Niu, & Korvarsky, 2003), the argument is made that significant clusters may more easily be extracted from an alternative representation of hierarchical clustering results called reachability plots. A method is then described for converting dendograms into reachability plots as a pre-processing step to extracting the significant clusters.

In this project, we have chosen to import the hierarchical clustering implemented in the data mining software package Weka (named after a bird found only on the islands of New Zealand, a flightless bird with an inquisitive nature!) (Hall, et al., 2009)

The implementation is of classic agglomerative clustering methods with the following link types available:-

- **SINGLE**: The distance d(A,B)between clusters A and B is the minimum of distances d(a,b), where a is from A and b from B. (**Single Link** is generally considered good at handling non-elliptical shapes but is sensitive to noise and outliers.)

- **COMPLETE**: The distance d(A,B)between clusters A and B is the maximum of distances d(a,b), where a is from A and b from B. (**Complete Link** is less susceptible to noise and outliers than Single Link, but it can split large clusters and it favours globular shapes.)

- **AVERAGE**: The distance d(A,B)between clusters A and B is the average of the distances d(a,b), where a is from A and b from B. (**Average** Link is considered an intermediate between the Single and Complete Links.)

- **MEAN**: The distance d(A,B) between clusters A and B is the average of the distances d(a,b), where a and b are from the merged cluster A U B.

- **CENTROID**: The distance d(A,B) between clusters A and B is the distance between the centroid of A and the centroid of B. (Centroid here is interpreted in the K-Means sense. The assumption is that the data consists of vectors of real attributes and the centroid is the vector given by taking the average of each of the components in turn.)

- **WARD**: The distance d(A,B) between clusters A and B is the change in “information” caused by merging the two clusters. Here, the “information” in a cluster is the error sum of squares between

the centroid (defined in the same way as for the CENTROID link type) of a cluster and all its members. So d(A,B) = ESS(A U B) – ESS(A) – ESS(B), where ESS(C), is the error sum of squares of the cluster C. (Ward Link minimizes the same objective or potential function as the K-Means algorithm. It can be shown mathematically that the Ward link type gives very similar results to the Average Link Type when the distance between two points is the square of the Euclidean distance between them. If this is the case and the one link type gives an interesting hierarchy of clusters, the other link type may enhance that hierarchy. If the distance between two points is other than the square of the Euclidean distance between them, the two link types no longer have the same mathematical affinity to each other.)

- **ADJUSTED COMPLETE**: The distance d(A,B) between clusters A and B is the maximum of distances d(a,b), where a is from A and b from B, the COMPLETE distance, minus the ADJUSTMENT, which is the largest within-cluster distance (d(x,y) where x and y are either both from A or both from B).

- **NEIGHBOUR JOINING**: The neighbour joining link type is not so easily described as the other link types. Its original motivation was in the area of biological classification. The link type is explained on the Wikipedia[5]. The original paper in which this link type is defined is (Saito & Nei, The neighbour-joining method: a new method for reconstructing phylogenetic trees., 1987).

In "Section 3.5.1 K-Means" we observed that the Shortest Path and Commuting Time distance metrics could not be used with the K-Means algorithm because these metrics were not defined on R^n for some n and there was no way of calculating the centroid of a set of vertices of the graph. Two of the eight link types enumerated above, **CENTROID** and **WARD**, have the same requirement that the centroid of a set of data points can be calculated. When hierarchical clustering is used with e.g. the Commuting Times metric, it must be with one of the other link types. As we also observed in "Section 3.5.1 K-Means", the Euclidean Commute Times metric is defined in terms of real vectors corresponding to graph vertices and invites the usage of all the hierarchical clustering link types (see Appendix A).

Seeing that the two link types, CENTROID and WARD, have distinguished themselves by their non-availability with our graph distance metric of choice, the Commuting Times distance, we highlight another disadvantage of these two link types compared with the others. For the other link types, the distance between merged clusters monotonically increases (or is, at worst, non-decreasing) as we proceed from singleton clusters to one all-inclusive cluster. Centroid-based link types, including CENTROID and WARD, do not share this property.

4 Clusters in Graphs

The objective of this section is to take a step back and up and review the wider field of study of clusters in graphs, why it is interesting, what the basic concepts are and which methods and algorithms are used in cluster detection. Much of the material in this section owes a direct or indirect debt to (Fortunato, 2010) which seems to us almost encyclopaedic in its scope as well as achieving for the most part a nice balance in including and omitting technical details. We have restricted our focus to concepts and, in particular, methods which have some association, albeit in some cases loose, to the programming and testing described in the subsequent sections of this report.

[5] Source: http://en.wikipedia.org/wiki/Neighbor_joining

4.1 Why look for graphs in clusters?

At the very start of "Section 3.1 Graph Definitions", we attempted to motivate the idea of graphs by giving some examples, road maps, electrical circuits and then, slightly more abstractly, relations within a social network in which members of the network can be represented by vertices and connections between them by edges. This last example was selected because it is one of the main areas in which the graph representation has proved amenable to many analytic approaches including the clustering of vertices/members. To create the graph representation much of the data is discarded. If and when clusters are revealed in this simplified dataset, the vertices in such a cluster represent an extraction of a subset of the original set of data objects which can then be studied again in the context of the original data to see if the common traits of the extraction can be deduced, patterns which might otherwise remain hidden. The transformation from network to graph is relatively easy to make and graph clustering algorithms have had some success in the area of social networks with revealing properties of and relationships between vertices that have not been available from direct observation and/or measurement. Later in this section we report some successes in connection with an explanation of the algorithms which generated them.

A second area of study where graph representation in general and graph clustering in particular has proved a useful tool is in the area of biological networks. These networks (e.g. protein-protein interaction networks, gene regulatory networks, metabolic networks) are not so easily described to a lay person as social networks can be but the principal is the same. The graph representation simplifies the data and if the simplification can be clustered, the clusters may reveal to the biologists common properties of and relationships between the clustered subsets of the data objects. The extent to which the biologists gain new insight from the reported clusters is the final measure of success of the application of the graph clustering algorithm.

4.2 What is a cluster in a graph?

Before tackling the problematic definition of a cluster, we state the easy definition of a **partition**. A **partition** is a division of a graph in clusters, so that each vertex belongs to one cluster. Even when looking for clusters, the search does not necessarily involve a partition. Particularly with large graphs, a collection of cluster subgraphs which constitute only a fraction of the original graph without any additional information about what lies outside them may be a more than satisfactory result.

Intuitively, a cluster within a graph is a densely connected subgraph relatively isolated from the rest of the graph. Expressed slightly differently, there will be relatively many edges connecting the vertices in a cluster and relatively few edges connecting the vertices in a cluster to the rest of the graph.

This implies a difference in densities between the edges connecting vertices within a cluster and edges incident or close to the vertices within it but not actually part of the cluster.

This, in turn, implies that the identification of clusters within a graph requires a graph to be "sparse". If the number of edges is much greater than the number of vertices, the distribution of edges can become too homogeneous for the identification of clusters to make sense. (Note that this is not the case if the graph is weighted.)

The problem is to quantify the intuition above, and it turns out to be a complex problem. In many cases, clusters are algorithmically defined; they are the result of the algorithm without any attempt to define the aim before it is applied. In this respect we also must plead guilty!

There is no theoretical framework defining precisely what clustering algorithms are supposed to do and until such a framework is developed and agreed upon, deciding which algorithm does the best job will remain a subjective and imprecise pursuit.

It is the quantification rather than the intuition which remains unresolved. There are three main approaches to the resolution:-

1. Local definitions

With this approach, the focus is on a subgraph, including possibly an immediate neighbourhood, evaluated independently of the graph as a whole. A cluster is often defined as a maximal subgraph satisfying some cohesive property. Some of the social networks studied today involve millions of vertices but in many of these studies much of the network would be irrelevant. It seems likely that local definitions will constitute the framework in which such studies are eventually evaluated. We give some examples of locally defined clusters:-

- **Cliques** (Luce, 1950): A social community can be defined as a subgroup in which every member is a friend of every other member. This would correspond to a complete subgraph, a **clique**, in the corresponding graph. Provided the group had a certain size, it would still be well-defined if there were one or two members who were not mutual friends. This suggests that clique is normally too strict a definition. A relaxation of it is an **n-clique** which is a maximal subgraph in which the shortest path between any two vertices has length no greater than n. (Note that a **clique** is a **1-clique**.) A disadvantage of this definition is that the shortest paths between vertices in an n-clique may include edges outside the n-clique.

- **Strong Communities** (Radicchi, Castellano, Cecconi, Loreto, & Parisi, 2004): For a subgraph C of a graph G and a vertex belonging to C, we define the **internal degree** k_v^{int} of the vertex v to be the number of edges connecting v to another vertex in C and the **external degree** k_v^{ext} of the vertex v to be number of edges connecting v to a vertex outside C. A **strong community** is a subgraph in which the internal degree of each vertex exceeds the external degree.

- **Lambda Sets** (Borgatti, Everett, & Shirley, 1990): The **edge connectivity** of two vertices in a graph is the minimal number of edges which have to be removed in order to disconnect the vertices. A **Lambda set** is a subgraph in which any pair of vertices belonging to the subgraph has a larger edge-connectivity than a pair of vertices in which only one of the two belongs to the subgraph. A disadvantage of this definition is that two vertices in a lambda set may be quite distant from each other.

- **Fitness measures**: A fitness measure is a measure expressing the extent to which a subgraph satisfies a property related to its cohesion. Fitness measures are used as quality functions for clusters. (See “Section 4.3 Cluster Quality”) The simplest example is the intra-cluster density, $\delta_{int}(C)$ which is the ratio of the number of edges in C to the total number of all possible edges in C ($n_c(n_c-1)/2$, where n_c is the number of vertices in C)

2. Global definitions

The basic idea behind global definitions is that the further away a graph is from being a random graph, the more cluster structure it is likely to possess. This idea leaves us with two properties to specify, when a graph is random and how to measure dissimilarity from a random graph.

Probably the simplest way of defining a random graph is to fix the probability that any two vertices are adjacent (Erdos & Renyi, 1959).

Another way of generating a random graph or at least a partially random graph is to create a **null model.** Intuitively, we tend to think of a good partition of a graph into clusters as one in which the number of edges between clusters is "small". In practice, "small" is dependent on the context, the number of vertices and the density of edges in the whole graph and, either consciously or less so, is related to a value judgment, an expectation, of how many edges there should be between clusters. Newman and Girvan (Newman & Girvan, 2004) formalized this idea of "expected number of edges" and defined the **modularity** Q of a partition as

Q = (number of edges within communities) – (expected number of such edges).

Determining the "expected number of edges" is equivalent to deciding how the graph would be if it were randomly constructed (without regard to clusters) and, once we have observed that equivalence, the formula for modularity is explicitly an expression of how different the actual graph is from a random graph. The random graph must have some properties in common with the actual graph for the comparison to make sense. It must at least have the same number of vertices so that we can divide the vertices into the same clusters for comparison.

If the expected number of edges between two vertices i and j is denoted by P_{ij}, the Q is the sum over all pairs of vertices in the same cluster of $A_{ij} – P_{ij}$, where A_{ij} is the appropriate entry from the adjacency matrix. What remains is to decide if there should be any further constraints on P_{ij} for the comparison with the original graph to be useful. One possibility for P_{ij} is to use the simple (Erdos & Renyi, 1959) definition referred to above. This could indeed be used but it leads to a distribution of vertex degrees which poorly reflects the actual distribution of most real-world networks. One way of attempting to reflect this distribution is to add the constraint $\sum P_{ij} = k_i$, where k_i is the degree of i and the sum is over all vertices j in the graph. This constraint expresses that the expected degree of each vertex in the graph is equal to its actual degree and is the final constraint normally used to define the **null model** against which the actual graph is compared. The simplest probabilistic model for satisfying this constraint is one in which the probability of an edge being incident to a vertex i as being dependent alone on the degree k_i of the vertex and in which the probabilities for the two ends of a single edge are independent of each other. This leads to the most widely used definition of the null model probability function as $P_{ij} = k_i k_j/2m$, where m is the number of edges in the actual graph. The description above is an abbreviated version of that to be found in (Newman M. E., 2006b)

3. Vertex similarity

The basic idea here is that clusters consist of vertices which are similar to each other so the graph is partitioned in such a way that each vertex is in the cluster whose vertices are most similar to it. We have defined three similarity measures between vertices of a graph in "Section 3.4 Distance Metrics on Graphs".

Technically, the measures defined there are dissimilarity measures where the higher the value, the less similar or more distant the vertices are. So the Commuting Times distance metric which resides at the heart of our project is a cluster definition based on vertex similarity.

The measure of vertex similarity can form the basis for a measure of cluster quality, e.g. the popular Silhouette Coefficient. This is first calculated for individual vertices and can then be averaged over the vertices in a cluster to give the cluster quality measure (and then averaged again over the clusters to give a partition quality measure. For an individual vertex v, the Silhouette Coefficient is defined by

$S = (B – A) / \max(A,B)$,

where A = the average distance from v to all points in its cluster, the **intra-cluster** distance, and

B = min(average distance of v to points in another cluster), the minimum of the **inter-cluster** distances for v.

The Silhouette Coefficient for a vertex should be positive. If it is not, then the vertex should probably be re-clustered before calculating the wider Silhouette Coefficients.

4.3 Cluster Quality

In the previous section we observed how the lack of widely agreed quantitative definitions for clusters made the evaluation of the results of a clustering algorithm an imprecise science.

A superficially alternative approach to evaluation is to agree a set of benchmark graphs with known cluster structure and to measure the performance of an algorithm on this set of graphs. In order to create such a set, practical examples of graphs with cluster structure have to be designed and the design has to be on the basis of some concept of cluster and partition which takes us back to the original problem but the alternative approach would at least allow evaluation on the basis of a multiplicity of cluster measures and definitions if the search for a unifying definition and measure continues to elude. The sets of benchmark graphs used so far, e.g. those used by Girvan and Newman when testing their algorithm, seem to be constructed according to the **planted-partition model** of (Condon & Karp, 2001).

In this model, the n vertices of a graph are divided into l groups of equal size. Two vertices within a group are connected by an edge with probability p and two vertices not in the same group are connected with probability r, where $r < p$. Such a graph implies the “natural” partition of a graph, the one that any algorithm should find.

Many graphs reflecting the real world, e.g. those describing social networks, have a hierarchical structure with larger clusters containing smaller clusters. For such graphs, a good clustering method should be able to find the hierarchy of clusters. If this is too demanding, it would at least be nice to know which of the clusters an execution of an algorithm could be expected to find. Approaching the problem from the point of view of the algorithm, some methods give a full hierarchy of clusters from the whole graph down to a set of clusters containing one vertex each or vice versa. In these cases, how can the significance of individual clusters in this vast set be gauged?

Just as it is easier to define a partition than it is to define a cluster, it seems easier to define a quality function for a partition than for a single cluster. **Coverage**, the ratio of intra-cluster edges divided by the total number of edges is one quality function for partitions. The most popular one is the **modularity** quality

function of Newman and Girvan (Newman & Girvan, 2004) which is based on their **null model** described in the previous section. Formally, this is defined by

$2m.Q = \sum (A_{ij} - P_{ij})\ \delta(C_i,C_j)$,

where m is the number of edges in the graph, where the sum runs over all pairs of vertices i,j in the graph and where $A=(A_{ij})$ is the adjacency matrix of the graph. P_{ij} represents the expected number of edges between vertices i and j in the null model. The delta function $\delta(C_i,C_j)$ is 1 if the clusters C_i and C_j to which vertices I and j respectively belong are the same and 0 otherwise.

Another approach to measuring the quality of a partition has the advantage that it can be applied for any clustering technique. It measures the stability of the partition against random perturbations of the graph structure. The idea is that a stable partition will survive small modifications of the graph whereas an unstable partition will quickly be disrupted by such changes. The perturbations could e.g. consist of the removal and addition of a small number of edges decided at random with a probability proportionate to the degrees of the vertices at their endpoints (Karrer, Levina, & Newman, 2008).

4.4 How to find clusters in graphs

4.4.1 Graph Partitioning

In the problem of graph partitioning, the vertices are defined into g groups of predefined size, such that the number of edges joining vertices in different groups is minimal and referred to as the **cut size**. It is necessary to specify both the number of clusters and their size. If the number of clusters is omitted, the cut size is trivially achieved with a partition consisting of one cluster. If the size of the clusters is omitted, the solution consists of separating a vertex of minimal degree from the rest of the graph.

Many algorithms bisect the graph and partitions into more than two clusters are achieved by iterative bisections. The constraint that the clusters have equal size is often imposed.

One of the main areas of application of this technique is in parallel computing, a successor to the problem of partitioning electronic circuits onto boards which was the motivation behind the **Kernighan-Lin algorithm** (Kernighan & Lin, 1970), one of the earliest graph partitioning algorithms and one which is still frequently used. The algorithm optimizes the benefit function Q, which represents the difference between the number of intra-cluster edges and the number of inter-cluster edges. The algorithm starts with bisection into two clusters of predefined-size. Subsets of vertices of equal size are then swapped between the clusters to maximize the increase of Q (Steps are needed in which the value of Q decreases in order to avoid converging to local maxima of Q). The performance of the Kernighan-Lin algorithm is $O(n^2 \log n)$.

Another popular graph partitioning technique is **spectral bisection** which is based on the properties of the spectrum of the Laplacian matrix of a graph, interesting to us because the Laplacian matrix lies at the heart of the Commuting Times distance metric (See “Section 3.4.3 Commuting Time distance metric”).

The basic idea is to use an index vector s whose entries are 1 or -1 according to whether the corresponding vertex is in one cluster or the other. The cut size R of the partition is given by

$4R = s^T L s$, where L is the Laplacian matrix.

The vector s may be written as the sum $\sum a_i v_i$, where $v_1,\ldots,v_n$ are the eigenvectors of L. If s is normalized, $R=\sum a_i^2 l_i$, where $l_1,\ldots,l_n$ are the eigenvalues of L. In general, the task of minimalizing R is still very hard but if the value of l_2, the smallest non-zero eigenvalue of L is sufficiently close to 0, a good approximation can be obtained by selecting s "almost" parallel to the corresponding eigenvector v_2, also known as the Fiedler vector (Fiedler, 1973). The best strategy for "almost" parallel is to match signs, $s_i = 1$ if the i^{th} component of v_2 is positive and $s_i = -1$ otherwise. In this case, the signs of the components of the Fiedler vector determine the sizes of the two components. An alternative strategy is to set the value of $s_i = 1$ for the values of i which correspond to the e.g. n_1 largest components of v_2, where n_1 is the size of one of the clusters.

4.4.2 Divisive Algorithms

The philosophy of divisive algorithms is to detect the edges that connect clusters and to remove them until the clusters are disconnected from each other.

4.4.3 Divisive Algorithms: The Girvan-Newman Algorithm

The most popular divisive algorithm is the Girvan-Newman algorithm originally described in (Girvan & Newman, 2002) and extended in (Newman & Girvan, 2004), where two additional methods of detecting the edges connecting clusters were described and tested.

It is impossible to overestimate the importance of the Girvan-Newman algorithm and the papers describing it. Many of the practical successes of graph clustering have been achieved either with the original algorithm or a subsequent refinement. In addition, the benchmark graphs generated by the authors to test the algorithm have been used to evaluate many other graph clustering algorithms. Finally, the modularity function defined in order to measure the quality of the different partitions which the divisive algorithm produces with continued edge removal has become the most widely-used partition quality function and has spawned a new family of algorithms specifically designed to optimize this function. (See 4.4.7 Optimizing Modularity)

The centrality of a vertex v in a graph had previously been defined by the sociologist Linton Freeman (Freeman, 1977) by the concept of **betweenness**, the number of shortest paths between pairs of vertices distinct from each other and from v which pass through v. It is normally divided by (n-1)(n-2)/2, where n is the number of vertices in order to give a value between 0 and 1.

Part of the innovation of Girvan and Newman was to extend this definition of centrality from vertices to edges. They defined the **edge-betweenness** of an edge to be the number of shortest paths between pairs of vertices that include it. With this definition, the steps of the algorithm are:-

1. Compute edge-betweenness for all edges
2. Remove the edge with largest edge-betweenness, or one of them at random, if there are several.
3. Recalculate edge-betweenness for edges for which it may be different after step 2 (The edges in the connected cluster in which an edge was removed or in the two connected clusters just separated by the removal of the edge if this removal split the subgraph).
4. Iterate the cycle from step 2.

The two additional centrality measures introduced in (Newman & Girvan, 2004) bear an association to the Resistance Distance and Commuting Times Distance defined in "Sections 3.4.2 Resistance Distance and 3.4.3 Commuting Time distance metric". They were:-

- Current-flow edge-betweenness: This definition relies on the transformation of a graph to an electric circuit as described in "Section 3.4.2 Resistance Distance". A unit current source and sink have also to be defined in turn for each pair of vertices. The resulting flow from source to sink will flow along a multitude of paths, those with the least resistance carrying the greatest fraction of the current. The current-flow edge-betweenness is the sum over all source/sink pairs of the absolute value of the current flowing along the edge.

- Random-walk edge-betweenness: This is the sum over all pairs of vertices of the expected number of random walks between the two vertices which includes the edge.

In "Section 3.4.3 Commuting Time distance metric", we reference a proof that the Resistance Distance and the Commuting Time distance are the same apart from a scalar constant (Chandra, Raghavan, Ruzzo, Smolensky, & Tiwari, 1989). Girvan and Newman show that these two apparently different definitions of centrality of an edge are in fact identical.

The running time for the Girvan-Newman algorithm is $O(m^2n)$, or $O(n^3)$ for sparse graphs, where m is the number of edges and n the number of vertices. Some of the subsequent refinements of the algorithm are motivated by an improvement in performance which allows the algorithm to be applied to larger graphs. In (Tyler, Wilkinson, & Huberman, 2003), the authors reduce the running time by reducing the number of edges for which edge-betweenness needs to be calculated, enabling the analysis of graphs of gene-co-occurrences which had been too large for the application of the original algorithm. An alternative enhancement (Rattigan, Maier, & Jensen, 2007) has managed to reduce the complexity to O(m) by substituting the actual calculation of edge-betweenness by a quick approximation.

We conclude this section by lauding some of the practical successes of the Girvan-Newman algorithm:-

- Girvan and Newman applied their algorithm to a network of scientific collaboration defined by co-authorship. As well as discriminating between research divisions, at least one unseen relation across divisions was revealed. In addition to the application of the algorithm, the definition and pre-processing of the data has influenced subsequent studies of collaboration.

- Functional modules in protein-protein interaction networks (PINs) were reliably detected. (Dunn, Dudbridge & Sanderson, 2005)

- Functional modules in yeast were found by (Chen & Yuan, 2006) who applied the algorithm with a modified definition of edge-betweenness.

- Functional relations between genes and associations between genes and diseases were inferred by (Wilkinson & Huberman, 2004) using the (Tyler, Wilkinson & Huberman, 2003) referred to in the previous paragraph

4.4.4 Hierarchical Clustering

Social networks are often characterized by a hierarchical community structure where smaller communities reside inside larger ones. As an example, a company may be organized in a pyramidal fashion with the

president at the top, individual workers at the bottom and departments, management, project groups etc. sandwiched in between. Part of the motivation for applying hierarchical clustering methods is the hope that such methods may reveal not just individual clusters but the whole multilevel hierarchy of communities or at least a significant portion of it. Even with data where this level of ambition seems beyond the capability of hierarchical clustering, one great advantage of the family of algorithms is that no assumptions about the number or size of the expected clusters are required prior to applying the algorithm.

As we have applied Hierarchical Clustering with the Commute Times Distance, we have discussed the details of the algorithm in "Section 3.5.3 Hierarchical Clustering". In that section we also discussed the limitations of the criteria and methods used to identify relevant partitions from the complete dendogram generated by the algorithm. We observe a second and related weakness, that it is hard to decide to what extent the results of the algorithm reflect the hierarchical structure of the graph and to what extent they are an artefact of the algorithm always generating a hierarchy. A third weakness is that vertices with a single neighbour are often classified as singleton clusters which does not make sense in most cases. The computational complexity of the algorithm is e.g. $O(n^2)$ for SINGLE LINK and $O(n^2 \log n)$ for COMPLETE and AVERAGE LINK.

We have concentrated on the weaknesses of hierarchical clustering, but it too can boast of practical successes, e.g.

- It was successfully used to reveal close connections in the network of committees in the US House of Representatives. The graph vertices were committees. Those sharing common members were linked by edges weighted by the number of common members divided by the expected number of common members if committee memberships had been randomly assigned. (Porter, Mucha, Newman, & Warmbrand, 2005) and (Porter M. A., Mucha, Newman, & Friend, 2007)
- (Rives & Galitski, 2003) had success with a hierarchical clustering technique in analysing the modular organization of a subset of the protein-protein interaction networks (PINs) of a particular yeast.

4.4.5 Partitional Clustering

In partitional clustering, the number of clusters, say K, is pre-assigned; this must be considered a deficiency for most graph clustering applications. Having said that, if the testing environment allows for rapid execution of multiple experiments and easy interpretation of the results, a claim we would make for the K-Medoids testing environment developed during the project, having to pre-assign the number of clusters becomes less of a hindrance than the basis for a collection of statistics about how often particular clusters appear with particular values of K; in this way the method is used more to reveal individual clusters than meaningful partitions.

In partitional clustering, the vertices of the graph are considered as points and a distance measure is defined on vertex pairs. The goal is to create a partition of K clusters which maximizes or minimizes a given cost function based on the distance measure. The most popular partitional technique is K-means, but the cost function requires the calculation of the centroid of a cluster which is sometimes artificial and sometimes impossible, as we explain in "Section 3.5.1 K-Means".

4.4.6 Partitional Clustering: K-Medoids

The details of our own partitional clustering method, the K-Medoids algorithm with the Commuting Times distance is described in "Section 3.5.2 K-Medoids". The cost function being minimized is the sum over all vertices of the Commuting Times distance from each vertex v to the medoid vertex of the cluster containing v.

We conclude this section with a brief survey (three articles!) of the use of the K-Medoids algorithm in graph clustering.

- (Rattigan, Maier, & Jensen, 2007) compare the performance of a modified K-Medoids algorithm with the classic Girvan-Newman edge-betweenness algorithms on large datasets. They increase the effectiveness of both algorithms by incorporating Network Structure Indices (NSIs). They construct graph data with a fixed number of vertices distributed among a fixed number of clusters and evaluate the results using two measures, a pair-wise **intra**-cluster accuracy measure and a pair-wise **inter**-cluster accuracy measure. The K-Medoids algorithm has a tendency to wrongly assign vertices on the periphery of clusters. They attempt to correct this tendency by introducing a post-processing step called **modal reassignment**. Simply described, modal reassignment is a random iteration through the vertices with a view to a possible reassignment of a vertex based upon the cluster assignment of its closest neighbours. The results of the enhanced algorithm with the incorporation of NSIs were promising, also on larger datasets than those which were the subject of earlier tests but even the enhanced algorithm is not recommended for very large datasets.

- (Firat, Chatterjee, & Yilmaz, 2007) carried out a number of experiments and concluded that a random walk distance measure, when used with distance based clustering algorithms on graphs, produced consistently better results than Euclidean distance measures. One of the clustering algorithms applied was the K-Medoids algorithm. They observed the same weakness with vertices on the periphery of clusters which (Rattigan, Maier, & Jensen, 2007) had noticed. They named their attempt at correction "Exception Bins". Exception Bins contain vertices that do not fit easily into one cluster; vertices which could equally easily belong to more than one cluster, possibly because they are positioned on the periphery of several clusters. Enhanced K-Medoids was compared with several other algorithms on a number of computer-generated datasets. The conclusion was that the effects of the enhancement could be measured but there were still concerns about the complexity of their algorithm and its ability to analyse larger real world datasets.

- In (Zhang, Zhao, & Wang, 2013) the authors have extended an agglomerative or hierarchical clustering algorithm and tested it on digraphs representing imagery data along with ten other clustering algorithms, one of which is K-Medoids. According to their results, K-Medoids is not a preferred algorithm when clustering the compact shapes which characterize this type of data. The K-Medoids algorithm was not the focus of the article but one of the algorithms against which the authors' extended algorithm was tested.

4.4.7 Optimizing Modularity

In "Section 4.4.3 Divisive Algorithms: The Girvan-Newman Algorithm", we wrote the definition of the partition quality function **modularity** which Girvan and Newman invented to evaluate the multiple partitions of their divisive algorithm. We mentioned the popularity of modularity and how new algorithms

had been devised to specifically optimize that quality measure. One of the strengths of modularity is that it unites several aspects of the Graph Clustering problem, e.g. the definition of cluster, the choice of a null model and an expression of the quality of partitions.

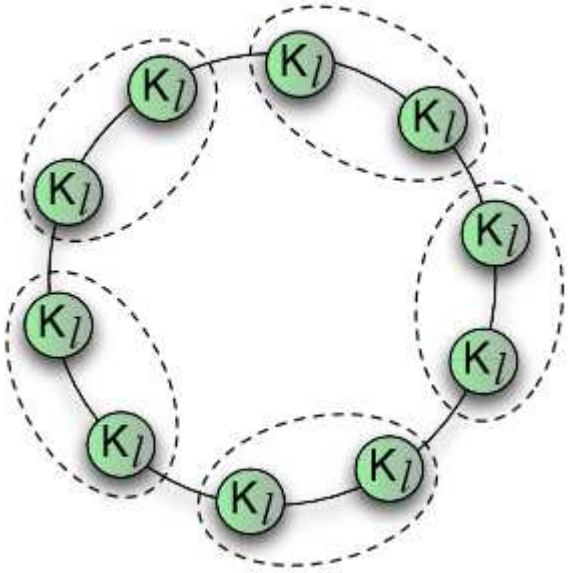


Figure 6: Modularity optimisation identifies groups of cliques instead of individual cliques (Fortunato, 2010)

The not unreasonable assumption of modularity optimization techniques is that high values of modularity indicate good partitions. This is not always the case. In the graph illustrated in Figure 6 which is a circuit of cliques joined by single edges, optimizing modularity does not as one would hope isolate the individual cliques but the groupings of them indicated by the ovals of dashes. This phenomenon begins to occur as the number of cliques n_c exceeds l^2, where l is the number of vertices in each clique. This is a striking example of a resolution problem which modularity exhibits more widely; it may be prevented in detecting small clusters even when they are as perfectly defined as cliques. Expressed more precisely, if a partition with maximum modularity includes clusters of size $m^{½}$ or less, where m is the number of edges, these clusters may in fact be unions of weakly interconnected smaller clusters.

Recently, (Good, de Montjoye, & Clauset, 2009) have made a careful analysis of modularity and its performance. They discovered that there are an exponential number of partitions whose modularity values are close to the global maximum and that the problem is particularly dramatic in graphs with a hierarchical cluster structure which is the case for many social network graphs. This explains why it is relatively easy heuristically to achieve solutions close to the global maximum but almost impossible to actually achieve it. More worryingly, partitions with modularity scores close to the global maximum may be wildly dissimilar.

In addition to the resolution and the vast numbers of near optimal solutions problems just described, there are also cases of random graphs being partitioned with high modularity scores.

The revelations of (Good, de Montjoye, & Clauset, 2009) may eventually undermine the popularity of modularity but for now, modularity optimization techniques are probably the family of techniques most widely applied. We briefly describe the first of a subfamily of these techniques called “greedy techniques”, developed by Newman himself (Newman, 2004a) and a couple of enhancements. The accuracy of greedy techniques is not that good compared to other modularity optimization techniques but it is the easiest to describe which is why it is selected here as an “appetite wetter”.

Newman’s greedy technique is an agglomerative hierarchical clustering technique where groups of vertices are joined by the addition of edges to form larger clusters in a way which maximizes the increase in modularity (maximization expressing the “greed” of the technique) compared to the previous partition

(Note that it is possible that for a single step in the algorithm, maximizing the increase could in practice mean minimizing the decrease.). The technique starts with n clusters, each containing a single vertex and no edges. Adding an edge reduces the number of clusters to n-1. The procedure finds n partitions, starting with one with n clusters, continuing with one containing n-1 clusters and so on until the last partition consists of one cluster. The complexity of the algorithm is O((m+n)n), or $O(n^2)$ for sparse graphs, where m, n are the number of edges, vertices respectively in the graph.

(Danon, Díaz-Guilera, & Arenas, 2006) suggested normalizing the variations in the modularity Q produced by the merger of two clusters by the fraction of edges incident to one of the two clusters. This normalisation improved the modularity optima, especially when the clusters involved were of widely differing sizes.

The Newman optimization algorithm involves the maintenance of a matrix $E=(e_{ij})$ expressing the fraction of edges in the current partition which join vertices in clusters i and j. (Clauset, Newman, & Moore, 2004) used data structures for sparse matrices to maintain this matrix and the matrix of modularity variations as well as a couple of other data structures to improve the performance of the algorithm. They succeeded in reducing the complexity to $O(n (\log n)^2)$ for graphs with a strong hierarchical structure. For graphs of this type, this optimization can analyse the cluster structure of graphs with up to 10^6 vertices, one of very few modularity optimization algorithms which can be successfully applied to graphs of this size.

As would be expected with such a widely applied family of techniques, modularity optimization has also had its successes:-

- (Traud, Kelsic, Mucha, & Porter, 2008) used Newman's greedy algorithm refined through additional (Kernighan & Lin, An Efficient Heuristic Procedure for Partitioning Graphs, 1970) type steps to infer relationships between the online and offline lives of students at a number of American universities. The graph clustered was constructed from anonymous Facebook data. It was discovered that communities were organized by class year in some of the universities and by house or dormitory affiliation in others.

- (Yuta , Ono, & Fujiwara, 2007) used the ultra-efficient algorithm by (Clauset, Newman, & Moore, 2004) to explain a gap in community size in a friendship network extracted from mixi.jp, the largest social networking site in Japan. Clusters of members of sizes ranging from 20 to 400 were rare. This was explained by participants forming new friendships by simultaneously closing ties with friends of friends and creating new links to individuals with similar interests.

- (Jin, Xing, Parkes, & Wolfe, 2007) used the Newman modularity optimization algorithm to cluster bidding networks created from EBay auctions connected if there was at least one common bidder. The results enabled the identification of substitute goods; i.e. goods with value for the bidder so that alternative or composite auctions can be recommended.

4.4.8 Spectral Clustering

Spectral Clustering includes all methods or techniques that partition the graph into clusters using the eigenvectors of matrices derived from the graph. We have already encountered an example; the **spectral bisection** method which was described in "Section 4.4.1 Graph Partitioning" could just as appropriately been classified and described in this section.

In spectral clustering, the set of n vertices in a graph are transformed into a set of matrix eigenvectors which are vectors in n-dimensional space. After the transformation, any techniques used to cluster n-dimensional vectors, e.g. K-Means, may be applied. Although it may seem an obscure abstraction, the representation of the graph induced by the eigenvectors can actually clarify the cluster structure of the initial data.

The first paper on spectral clustering (Donath & Hoffman, 1973) partitioned graphs using the eigenvectors of the adjacency matrix. At about the same time, Fiedler (Fiedler, 1973) realized that the eigenvector corresponding to the second smallest eigenvalue, i.e. the smallest non-zero eigenvalue, of the Laplacian matrix could be used to obtain a bipartition of the graph with very low cut size, as we have intimated in "Section 4.4.1 Graph Partitioning". The Laplacian matrix is by far the most used matrix in spectral clustering. We now attempt briefly to motivate why this matrix is especially well-suited for the task.

The Laplacian matrix has a number of zero eigenvalues equal to the number of connected components of the graph. In this case, the matrix can be written in block diagonal form, where each block is the Laplacian matrix of a connected component which therefore has a (1,1,…,1) eigenvector. These eigenvectors can be expanded into eigenvectors for the Laplacian matrix of the whole graph by adding 0's corresponding to each vertex not in the original connected component. So the connected components of the graph can be read directly from the non-zero entries of these eigenvectors.

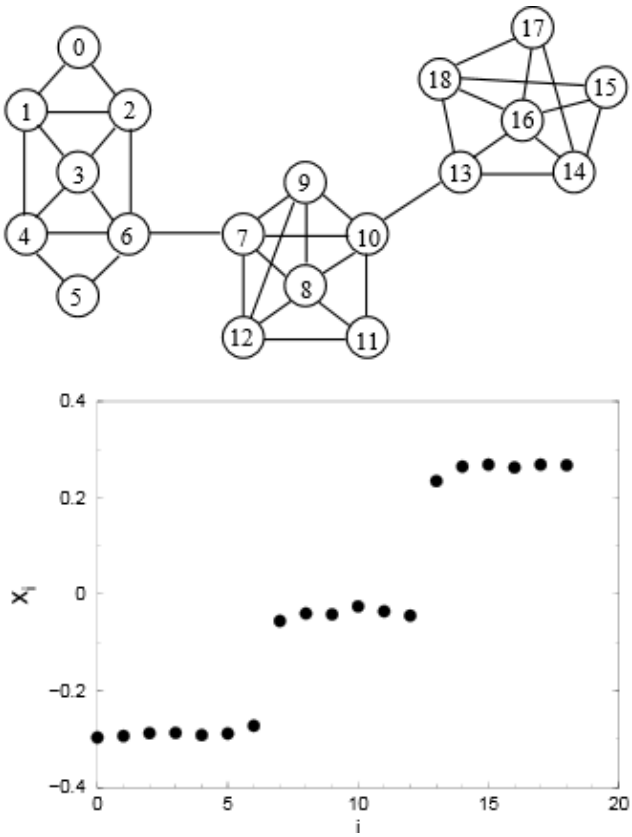


Figure 7: Components of an eigenvector reflecting clusters in a graph (Fortunato, 2010)

If the graph is connected, there is one zero eigenvalue and the rest are positive, and the Laplacian matrix can no longer be written in block diagonal form. If, however, the graph consists of k weakly linked subgraphs, the smallest k – 1 positive eigenvalues will still be close to zero and the first k eigenvectors should still enable one to distinguish the vertices in the weakly linked subgraphs. The diagram in Figure 7 illustrates the technique, albeit from a different graph-derived matrix; the components of the Fiedler vector can be divided into three sets of values, where each set of values corresponds to the vertices in a weakly-connected component.

Alternatively, the eigenvectors corresponding to vertices in the same cluster should still in general be close to each other and may be recovered using techniques like K-Means.

In general, calculating the first k eigenvectors of the Laplacian matrix is a problem of complexity $O(n^3)$. Fortunately, there are approximation techniques (Golub & Van Loan, 1989) which improve performance.

In the above discussion, the Laplacian matrix used is unnormalised. Spectral clustering using the unnormalised Laplacian matrix does not always converge and sometimes has a tendency to generate singleton clusters. Generally, the unnormalised Laplacian matrix tends to be related only to the inter-cluster edge-density whereas the normalised matrix tends to be related to both the inter-cluster and the intra-cluster edge-densities. The usual normalised matrix $L_{sym} = D^{-½}LD^{½}$, where L is the Laplacian matrix and D is the degree matrix of the graph may be used. An alternative $L_{rw} = D^{-1}L$ is, however, more reliable because the eigenvector corresponding to the lowest eigenvalues are cluster indicator vectors with non-zero entries corresponding to the vertices of a particular cluster and zero entries elsewhere if the clusters are disconnected.

Newman (Newman M. E., 2006a and 2006b) has employed the **spectral bisection** method with the modularity matrix replacing the Laplacian matrix. The hope is to optimize modularity on bipartitions.

One example of Spectral Clustering used in practice is (Sen, Kloczkowski, & Jernigan, 2006) who used Singular Value Decomposition to calculate the eigenvectors of the Laplacian matrix to identify protein clusters for yeast.

5 Implementing Clustering Based on the Commuting Times Metric

A primary goal of the project has been to implement the Commuting Times Distance Metric and to ally it to a K-Medoids implementation and to Hierarchical Clustering with an array of link types and then to test these distance-based clustering methods on an array of graph data. In this section we describe individual elements of the implementation we ourselves have coded and how these are merged with external software components to create a structure in which to achieve this.

5.1 The Programming Environment

Eclipse was selected as the programming environment and the java code was stored in the Google code repository.

Standard Windows PCs were used for the project so all programming, compiling and testing was carried out in this environment.

5.2 Accessing and Executing the Project Code

5.2.1 Execution

An executable jar file and dependent external Jars can be found on the Google code website[6] that hosts the project code under the distribution (dist) directory. The contents of the folder can be downloaded.

Alternatively the jar file can be found in the distribution (dist) directory that accompanies this project report.

Masters Project\graphClustering\dist directory.

In order to run the GUI double click on the graphClustering jar file.

[6] Google Code web site: http://code.google.com/p/graph-clustering-ra/

5.2.2 Accessing the source code and test material.

The source code can be found on the Google code website under (visual) or in the subdirectories of the code delivered with this report **Masters Project\graphClustering\visual.**

The Pajek NET files (See "Section 6.1 Pajek NET Format") on which the code was tested can be found on the Google code website under (src) or in the code delivered with this report **Masters Project\graphClustering\src\main\resources\datasets**

The external packages used in this project can be found on the Google code website (lib) subdirectories of the code delivered with this report **Masters Project\graphClustering\lib**

This source code accompanying this project has only been tested within a Windows environment.

A large part of the software is based on standard, freely available Java packages and frameworks as described in the following section.

5.3 Imported Libraries and their Function.

Where possible, we have tried to use standard software packages in order to take advantage of their stability and proven usefulness as well as allowing ourselves more time to concentrate on the original aspects of our work. Nevertheless, it has required a substantial amount of time, effort and occasionally ingenuity to unite the four frameworks described below and our own software into a compatible whole.

5.3.1 JUNG

The basis for the part of the implementation which reads, processes and displays graph data is the JUNG framework. The contents of the box below are how the authors describe their software. It is taken from the Jung website[7] .

> JUNG — the Java Universal Network/Graph Framework--is a software library that provides a common and extendible language for the modelling, analysis, and visualization of data that can be represented as a graph or network. It is written in Java, which allows JUNG-based applications to make use of the extensive built-in capabilities of the Java API, as well as those of other existing third-party Java libraries.

JUNG's Clustering Demo class has been selected as the basis of the implementation GUI. This class is intended to demonstrate the Girvan-Newman clustering algorithm. In the implementation the class has been renamed RAGraphClustering and extended to facilitate the visualization of both weighted and unweighted graphs stored in the Pajek NET format (Batagelj & Mrvar, Ljubljana). In addition to demonstrations of the Girvan-Newman clustering algorithm the K-Medoids clustering algorithm can also be demonstrated.

Jung is a recognized and well used graph manipulation package. The package contains a number of examples that show how to use the Jung framework.

The JUNG software contains an implementation of the ground breaking Girvan-Newman algorithm[8]. In this description of the EdgeBetweennessClusterer class, the implemented algorithm is described as a slight

[7] Source: http://sourceforge.net/apps/trac/jung/wiki/JUNGManual

modification to the Girvan-Newman algorithm, but the modification referred to is just for the purposes of the visual demonstration of the effects of edge-removal for varying numbers of edges. As a result, we were able to use this software to apply the Girvan-Newman algorithm to our datasets.

5.3.2 Colt

A large part of our implementation involves matrix manipulation as the adjacency and degree matrices of a graph are constructed and then manipulated to eventually arrive at the Commuting Times distance matrix. To this end, we use, in particular, the Linear Algebra classes (e.g. sparseDoubleMatrix) and algorithms (e.g. Eigenvalue Decomposition) from the Colt software framework. As with our acknowledgment of JUNG in the previous section, we quote from the internet[9] in order to allow those responsible for the software to describe it.

> This distribution provides an infrastructure for scalable scientific and technical computing in Java. It is particularly useful in the domain of High Energy Physics at CERN: It contains, among others, efficient and usable data structures and algorithms for Off-line and On-line Data Analysis, Linear Algebra, Multi-dimensional arrays, Statistics, Histogramming, Monte Carlo Simulation, Parallel & Concurrent Programming. It summons some of the best concepts, designs and implementations thought up over time by the community, ports or improves them and introduces new approaches where need arises. In overlapping areas, it is competitive or superior to toolkits such as STL, Root, HTL, CLHEP, TNT, GSL, C-RAND / WIN-RAND, (all C/C++) as well as IBM Array, JDK 1.2 Collections framework (all Java), in terms of performance (!), functionality and (re)usability.
>
> Copyright (c) 1999 CERN - European Organization for Nuclear Research.

5.3.3 mtj (*m*atrix-*t*oolkits-*j*ava)

In spite of the claims of competitiveness and even superiority claimed in the previous section by the developers of Colt, we were unable to overcome or work around apparent errors in the algorithms used to calculate the Moore-Penrose pseudoinverse of a matrix and here the **m**atrix-**t**oolkits-**j**ava package provided us with an alternative implementation in which we have yet to encounter errors. Colt and mtj have their own matrix representation classes. As we had already written a considerable amount of code based on the Colt matrix classes, we chose to write conversion routines between the mtj and Colt matrix classes as opposed to re-writing all our code with mtj matrix classes. This is likely to offend purist readers of our code and may well be refactored should the code be revisited at a later stage. There is, of course, also a performance penalty involved in copying and converting matrices between the two representations but we have as yet not encountered practical performance problems, even with the Enron dataset, so the motivation to reduce or eliminate this penalty is low.

We have no reservations in expressing our gratitude to mtj for giving us a viable algorithm for the calculation of the Moore-Penrose pseudoinverse. Here we allow Bjørn-Ove Heimsund and Samuel Halliday to describe their toolkits[10].

---

[8] Link to the Jung example:
http://jung.sourceforge.net/doc/api/edu/uci/ics/jung/algorithms/cluster/EdgeBetweennessClusterer.html
[9] Source: http://acs.lbl.gov/software/colt/
[10] Source: https://github.com/fommil/matrix-toolkits-java/blob/master/README.md)

Java linear algebra library powered by BLAS and LAPACK

MTJ is designed to be used as a library for developing numerical applications, both for small and large scale computations.

The library is based on BLAS and LAPACK for its dense and structured sparse computations, and on the Templates project for unstructured sparse operations.

MTJ uses the netlib-java project as a backend, which can be set up to use machine-optimised BLAS libraries for improved performance of dense matrix operations, falling back to a pure Java implementation. This ensures perfect portability, while allowing for improved performance in a production environment.

5.3.4 Weka

In a previous project we implemented a cosine distance function to use in conjunction with a K-Means algorithm and with our implementation of a K-Medoids algorithm which in this project is tested together with the Commuting Times distance function. This earlier development was an extension of the Weka data mining software package. This package contains a hierarchy of distance function classes and interfaces which new distance functions, including the cosine distance mentioned earlier and our new Commuting Times distance function, can implement and extend. It is through our extension of the Weka framework that the Commuting Times distance function is made available to the K-Medoids algorithm. Weka also includes an implementation of hierarchical clustering with several link types (See "Section 3.5.3 Hierarchical Clustering") and this implementation can thus also be applied with the Commuting Times distance function. We continue with our habit from the previous sections of quoting from the internet [11] to allow those responsible behind Weka to briefly describe it (Hall, et al., 2009).

Weka is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from your own Java code. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well-suited for developing new machine learning schemes.

Found only on the islands of New Zealand, the Weka is a flightless bird with an inquisitive nature.

Weka is open source software issued under the GNU General Public License.

5.4 The Structure of our Programming

We have attempted to observe good object-oriented programming practice by creating interfaces which define a contract and which the classes implementing those interfaces are then required to maintain.

Each of the classes we have designed and programmed contains a main method which allows it to be tested, as far as possible, independently of other classes as if it were an independent application.

[11] Source: http://www.cs.waikato.ac.nz/ml/weka/

In the previous section we described the software packages we have imported during the programming of this project. We now emphasize our own contribution, the classes and methods where we claim authorship of the code. The following diagram (Figure 8) illustrates the class dependency for these classes.

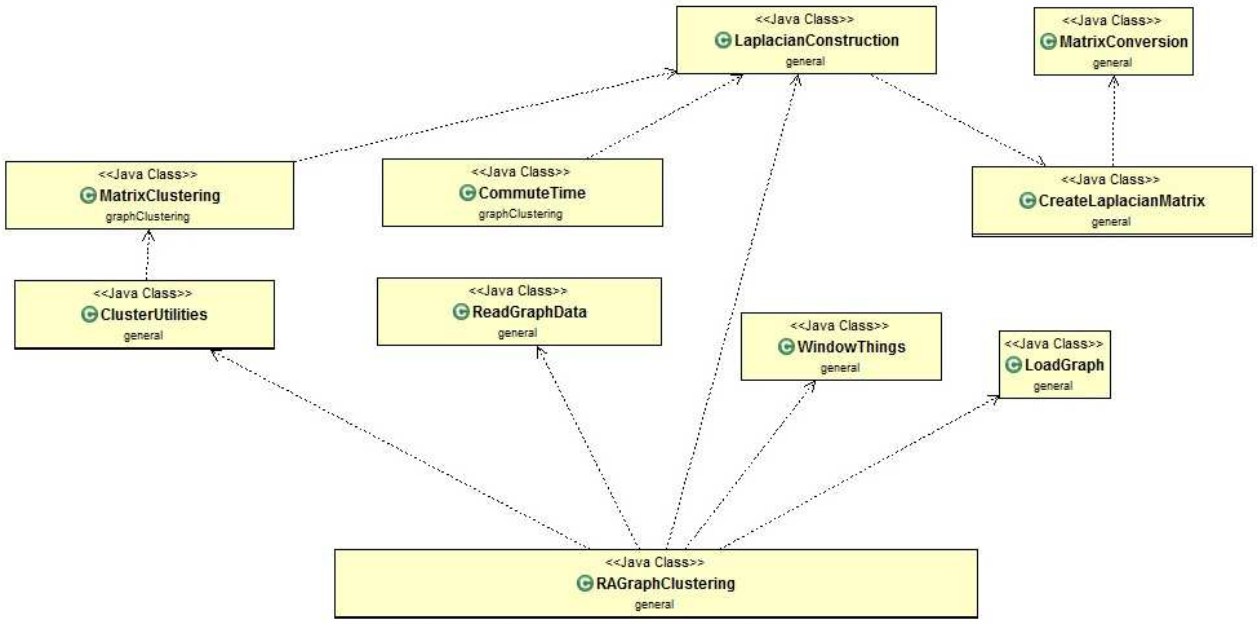


Figure 8: Project specific classes shown in a class dependency diagram

5.4.1 Using the CommuteTime Class to Illustrate the Programming Structure

To illustrate this principle, consider the CommuteTime class which returns the Commuting Time between any two vertices in a graph. This class is an extension of the Weka EuclidianDistance class.

In order for the CommuteTime distance to return distances between the vertices of a graph, the main method of the class presents the following open file dialog window.

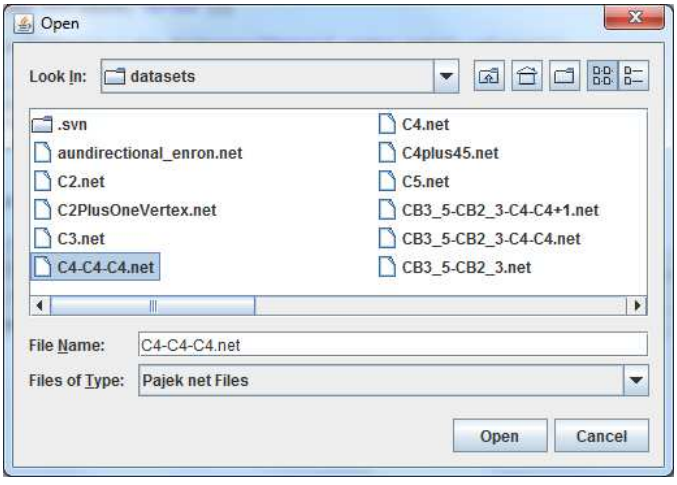


Figure 9: The open Pajek .net dialog box

The dialog box (see figure 9) allows the selection of a file containing graph data in Pajek NET format (Batagelj & Mrvar, Ljubljana). The file selected is read into a parameterized object of type Graph by the project specific class LoadGraph and a second project specific class LaplacianConstruction creates the Laplacian Matrix of the graph. This class also contains a getCommuteTimes method which loads the graph's Commuting Times distances into an object of the Colt SparseDoubeMatrix2D class.

An ArrayList of objects (actually a single object) of the Weka class attribute is created and this pseudo list of attributes is used to generate an object of the Weka class instances which contains a number of instances corresponding to the number of columns in the Laplacian matrix which is, of course, the number of vertices in the loaded graph. For each instance in the collection, the value of the single (pseudo)attribute is set to the index of the vertex in the Graph object. This enables us to define the distance between two instances in the collection as the appropriately indexed entry of the matrix returned by the getCommuteTimes method referenced in the previous paragraph.

The CommuteTime class can now be tested by printing the commute times between every combination of vertices using nested loops to run through all pairs of vertices. The code which implements this description in the main method of the CommuteTime class is in the box below.

```
ReadGraphDataI myGraphData = new ReadGraphData();
LoadGraphI myGraphLoader = new LoadGraph();
Graph<Number, Number> graph = null;
SettableTransformer<Number,Number> edgeWeights = null;
SparseDoubleMatrix2D matrix;

try {
                graph = myGraphLoader.loadGraphPnr(myGraphData.openFile(null,null));
                edgeWeights = myGraphLoader.getEdgeWeights();
   } catch (IOException e) {
                e.printStackTrace();
   }
LaplacianConstructionI myLaplacianmatrix = new LaplacianConstruction();
myLaplacianmatrix.constructMatrix(graph, edgeWeights);
matrix = myLaplacianmatrix.getCommuteTimes();
ArrayList<Attribute> vertex = new ArrayList<Attribute>(1);
vertex.add(new Attribute("Vertex"));
Instances ourInstances = new Instances("Matrix",vertex,matrix.columns());

for(int i = 0; i < matrix.columns();i++){
   Instance inst = new DenseInstance(1);
   inst.setValue(0,i);
   ourInstances.add(inst);
}

CommuteTime cd = new CommuteTime(ourInstances,matrix);
System.out.println("");
System.out.println(ourInstances.toSummaryString());
System.out.println("");

for(int i = 0; i < matrix.columns();i++){
   for(int j = 0; j < matrix.columns();j++){
  System.out.println("Commute time from vertex : "+ i  + " to  vertex   : " + j + "   is: " + cd.distance(i,j));
  System.out.println("Distance from instance   : "+ i  + " to  instance : " + j + " is: " +
cd.distance(ourInstances.get(i),ourInstances.get(j)));
System.out.println("");
 }
}
```

Code Example 1: Class CommuteTimes' main method

And the start of the resulting printout might look like this (in order to check that the commute time from $vertex_x$ to $vertex_y$ is the same as the value contained in the commute time matrix for the $instance_x$ to $instance_y$, both the commute time and the distance are printed):-

Distance with 2 parameters called:
Commute time from vertex : 0 to vertex : 2 is: 0.5
Distance from instance : 0 to instance : 2 is: 0.5

Distance with 2 parameters called:
Commute time from vertex : 0 to vertex : 3 is: 0.5000000000000004
Distance from instance : 0 to instance : 3 is: 0.5000000000000004

Distance with 2 parameters called:
Commute time from vertex : 0 to vertex : 4 is: 1.5000000000000013
Distance from instance : 0 to instance : 4 is: 1.5000000000000013

In the following sections we will describe the classes that have been specifically constructed for this project.

5.4.2 The CommuteTime Class.

The K-Medoids and Hierarchical clustering algorithms cluster with the help of a nearest neighbour search algorithm where the concept of "nearest" is defined in terms of a distance function between the instances being clustered. The CommuteTime class provides the distance function in our implementation.

It is an incongruity that the CommuteTime class extends the Weka EuclideanDistance class, particularly as we went to some length to describe in "Section 3.4.3 Commuting Time " the limitations in applying the Commuting Times distance function with clustering algorithms precisely because it was not a Euclidean distance. Intuitively, CommuteTime would be better placed alongside the Weka EuclideanDistance class and implementing the parent class NormalizableDistance. It could have been constructed in this way but having it extend the Euclidean class instead saved us from having to copy essentially irrelevant code from the EuclideanDistance class.

We have been sufficiently faithful to the implementation of distance in Weka to enable the Commuting Times distance function to be used with the K-Medoids and Hierarchical clustering algorithms without the need to modify them.

To summarize, the CommuteTime class makes available the Commuting Time distance between any two vertices in a graph by creating a collection of instances corresponding to these vertices and essentially indexed in the same way. We describe the steps by which this is facilitated.

When a commute time matrix has been calculated by

```
matrix = myLaplacianmatrix.getCommuteTimes();
```

it is passed on to the CommuteTime class and can be used to extract commute times from the matrix.

The WEKA distance object is defined on pairs of members of an Instances object which is a collection of instance objects, each of which has an ArrayList of attributes.

In order for the Commuting Distance to be used by the WEKA clustering algorithms without modification, it has to be expressed in terms of attributes belonging to two instances in a collection. This is done by adding to each instance a pseudo attribute ArrayList vertex containing just one actual attribute Vertex.

```
ArrayList<Attribute> vertex = new ArrayList<Attribute>(1);
vertex.add(new Attribute("Vertex"));
```

The collection of instances objects is then created with the label “Matrix” and the ArrayList vertex of attributes.

```
Instances ourInstances = new Instances("Matrix",vertex, matrix.columns());
```

For all instances in the collection the value of the one attribute Vertex is now set to the index of the corresponding vertex index in the Commuting Times matrix.

```
for(int i = 0; i < matrix.columns();i++){
            Instance inst = new DenseInstance(1);
            inst.setValue(0,i);
            ourInstances.add(inst);
}
```

The collection of instance objects and the Commuting Times matrix are used to define a distance object CommuteTime.

```
CommuteTime cd = new CommuteTime(ourInstances,matrix);
```

The actual distance between two instances first and second is obtained by reading from their only attribute the value of the indices with which they are represented in the indexing of the Commute Times matrix.

```
distance = ourMatrix.getQuick((int)first.valueSparse(0)  , (int)second.valueSparse(0));
```

```
InstanceComparator instanceComparator;
 instanceComparator = new InstanceComparator();
 if (instanceComparator.compare(first, second) == 0) {
    distance = 0;
 } else {
 distance = ourMatrix.getQuick((int)first.valueSparse(0),(int)second.valueSparse(0       ))
 }
 return distance;
```

Code Example 2: Selecting a commute time from a commute time matrix

An instance comparator is used to test if the two instances (or vertices) are equal, if they are equal (a distance of 0) then the value 0 is returned otherwise the distance between the two vertices is returned from the commute time matrix.

5.4.3 The RAGraphClustering Class

As we described in ”Section 5.3.1 JUNG”, the class RAGraphClustering is derived from the JUNG ClusteringDemo class which is used to demonstrate an EdgeBetweennessClusterer (actually, the Girvan-Newman algorithm) on the Zachary dataset. Not only have we used the demo but we have extended it by

adding a number of new buttons, sliders and dropdowns in order to expand its functionality. It is now able to open and process any graph represented in Pajek .net format. Furthermore it is now able to use more than one algorithm. The graph visualization classes and vertex colouring classes and functions, along with the mouse mode dropdown are all that remain from the original demo app. Much of the original functionality has been re-factored to reduce the amount of duplicate code.

Our extension class can read and process any graph stored in a Pajek NET format (Batagelj & Mrvar, Ljubljana). Furthermore, it can demonstrate the original EdgeBetweennessClusterer and our own K-Medoids clusterer.

A graph is read by invoking an instance of the class ReadGraphData (See “Section 5.4.4 The ReadGraphData Class”). The buffered reader returned by ReadGraphData is passed on to the method setUpView which loads the graph into memory using the class LoadGraph. (See “Section 5.4.5 The LoadGraph Class”) The class LaplacianConstruction (See “Section 5.4.6 The LaplacianConstruction Class”) creates a Commuting Times distance matrix for the graph which in this context is used by the K-Medoids clustering algorithm.

A Jung visualization viewer (see Figure 10) is created which adds mouse behaviour and other rendering functionality to the graph visualization object. The method then builds the Graphical User Interface and assigns functionality to the individual elements within this GUI, buttons, sliders and dropdown boxes.

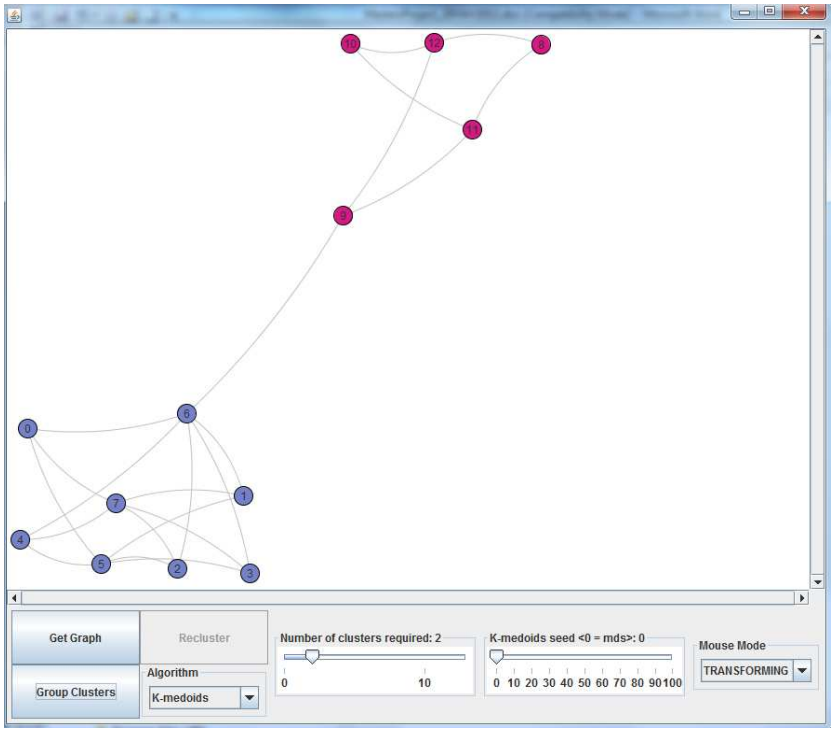


Figure 10: A typical view of a Graph after a clustering run

The button ‘Get graph’ allows the user to open a new graph in Pajek net format (Batagelj & Mrvar, Ljubljana). The button uses the classes ReadGraphData and LoadGraph to open and read the new Pajek net file as described above.

The button 'Recluster' is only active if the K-medoids seed is set to random (100 on the seed slider). When activated the K-medoids algorithm is re-run using a randomly generated seed. This seed is a randomly generated integer.

The button 'Group Clusters' (see Figure 11) groups or ungroups the clusters found by using the ClusterUtilities class. (See "Section 5.4.8 The ClusterUtilities Class")

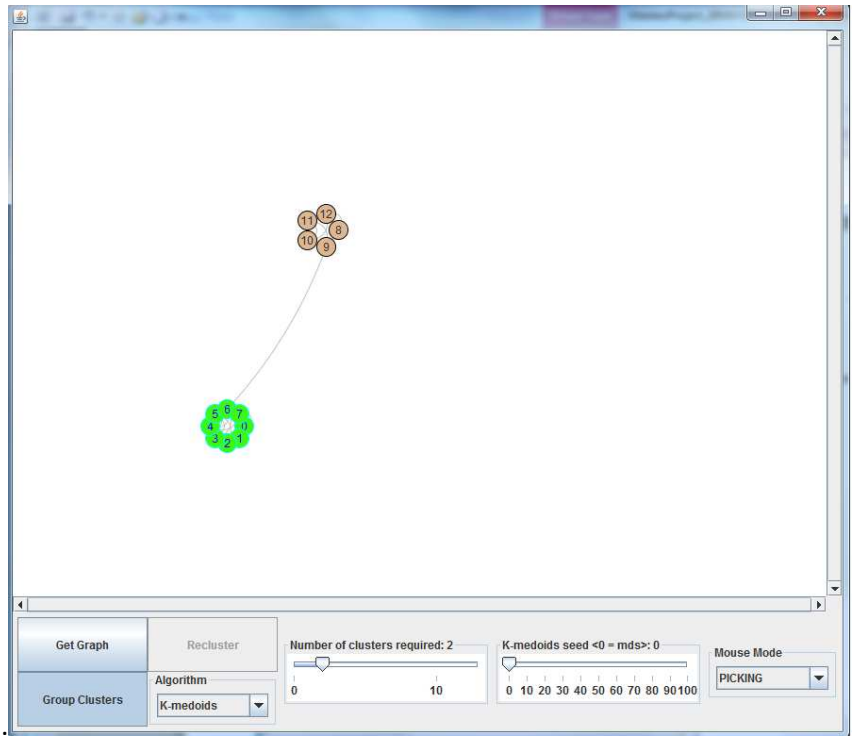


Figure 11: A typical view showing the grouping of individual clusters

The 'algorithm' dropdown box facilitates the choice of clustering algorithm to be used. When a choice is made, the ClusteringUtilities class is used to recluster the graph data, passing the number of edges to be removed or the number of clusters required as appropriate to the chosen algorithm. The visualization viewer is then used to update the Gui.

```
clustersOrEdgesToRemove = source.getValue();
  myClusterUtilities.clusterAndRecolor(layout, clustersOrEdgesToRemove,
  groupVertices.isSelected(), matrix, algorithm);
  if (algorithm == "Girvan-Newman"){
 edgeBetweennessSlider.setMaximum(graph.getEdgeCount() );
        sliderBorder.setTitle(boxTitle[0] + + edgeBetweennessSlider.getValue());
  } else {
                    edgeBetweennessSlider.setMaximum(graph.getVertexCount());
sliderBorder.setTitle(boxTitle[1] + edgeBetweennessSlider.getValue());
  }
  eastControls.repaint();
  vv.validate();
  vv.repaint();
```

Code Example 3: Illustrating the selection of clustering algorithm

The slider "Edges removed for clusters: "/"Number of clusters required: " is assigned according to the chosen algorithm. The value chosen is then passed to the clusteringUtilities object and the clusters are re-calculated and the Gui updated.

The slider 'K-Medoids seed' is used to assign a value to the seed used by the K-Medoids algorithm. A seed value of 0 is interpreted as no seed and instead the Minimal Distance Sums initialization method of the K-Medoids algorithm is applied (See Section "3.5.2 K-Medoids"). A seed value of 100 is interpreted as a random seed and a random seed will be generated and then applied for the K-Medoids algorithm.

The dropdown box 'Mouse Mode' uses the JUNG framework to alter the mouse actions.

The next improvements we would like to the Gui would be:

- Expand the Gui so that 4 clustering results can be seen at the same time, this will enable quicker comparisons of differing algorithms and algorithm parameters.
- Save clusters in a file format for post processing, cluster number, vertex id.

5.4.4 The ReadGraphData Class

The class possesses a single method which displays an 'open file dialog' box. This box allows the user to navigate down a directory hierarchy to select a file in Pajek NET format. The method returns a buffered reader.

5.4.5 The LoadGraph Class

The class contains two methods:-

- **loadGraphPnr**, which reads the Pajek net file from the buffered reader into a Jung Graph object. If edge weights are present, these are read into a JUNG SettableTransformer object. The load method in the JUNG PajekNetReader class fills out the Graph object with data from the Pajek file which loadGraphPnr returns.

```
PajekNetReader<Graph<Number, Number>, Number,Number> pnr =
   new PajekNetReader<Graph<Number, Number>, Number,Number>(vertexFactory, edgeFactory);
   final Graph<Number,Number> graph = new SparseMultigraph<Number, Number>();
   if (br == null){
     return null;
   }
   pnr.load(br, graph);
   edgeWeights = pnr.getEdgeWeightTransformer();
return graph;
```

Code Example 4: Loading a graph in the Pajek .net format

- **getEdgeWeights**: this method returns the object containing the edge weights.

5.4.6 The LaplacianConstruction Class

This class contains two methods:-

- **constructMatrix**, which receives an instance of Graph as a parameter and calculates the Commuting Times for the instance using the class createLaplacianMatrix (See "Section 5.4.7 The CreateLaplacianMatrix Class") by invoking in turn the methods generateLaplacianMatrix, createMPPseudoInverse and createCommuteTimesMatrix.

```
CreateLaplacianMatrix ourMatrix = new CreateLaplacianMatrix();
SparseDoubleMatrix2D ourSparseDM = ourMatrix.generateLaplacianMatrix(graph, null, Weights);
SparseDoubleMatrix2D ourMPInverse = ourMatrix.createMPPseudoInverse(ourSparseDM);
SparseDoubleMatrix2D ourCommuteTimes = ourMatrix.createCommuteTimesMatrix(ourMPInverse);
commuteTimes = ourCommuteTimes;
```

Code Example 5: Creating a Commuting Times matrix

- **getCommuteTimes**, which does what it says.

5.4.7 The CreateLaplacianMatrix Class

This class has a plethora of methods, each one either executing a step in the calculation of the Commuting Times distances or providing a general matrix method used in one or more of the steps:-

- **generateLaplacianMatrix**, which receives an instance of Graph and an instance of SettableTransformer, containing the edge weights of the graph, as parameters and returns the Laplacian matrix of the Graph instance.

```
edgeWeights = Weights;
 if (nev == null)
    nev = new ConstantMap<E,Number>(1);
 numVertices = G.getVertices().size();
 SparseDoubleMatrix2D matrix = new SparseDoubleMatrix2D(numVertices,numVertices);
 BidiMap<V,Integer> id = Indexer.<V>create(G.getVertices());
 int i=0;
 for(V v : G.getVertices()){
    Number sumOfWeights = 0;
    for (E e : G.getOutEdges(v)) {
         Number myWeight = getEdgeWeight(nev.get(e));
         sumOfWeights = sumOfWeights.floatValue() + myWeight.floatValue();
         V w = G.getOpposite(v,e);
         int j = id.get(id.getKey(w));
         matrix.set(id.get(id.getKey(v)), j,   matrix.getQuick(id.get(id.getKey(v)),j) - myWeight.doubleValue());
    }
    matrix.set(id.get(id.getKey(v)), id.get(id.getKey(v)), sumOfWeights.floatValue());
    sumOfWeights = 0;
    i++;
      }
 return roundedMatrix(matrix);
```

Code Example 6: The creation of a weighted Laplacian matrix from a complete graph

- **createMPPseudoInverse**, which receives a matrix as parameter and creates the Moore Penrose Pseudo Inverse of that matrix, using the formula $A^{+} = VD^{+}U^{*}$, where A = UDV* is a singular value decomposition of A. A singular value decomposition of a matrix A is obtained by creating an instance of the mtj class SVD and invoking the factor method from that class. (See "Section 5.3.3 mtj (*m*atrix-*t*oolkits-*j*ava)")

```
if (mySVD == null) {
 createSingleValueDecomposition(A);
 }
 SparseDoubleMatrix2D ourUTransposed = Transpose(getSVD_U());
 SparseDoubleMatrix2D ourSReciprocal = createS_MP(getSVD_S());
 SparseDoubleMatrix2D ourV = Transpose(getSVD_Vt());
 return roundedMatrix(mult(ourV, mult(ourSReciprocal, ourUTransposed)));
```

Code Example 7: creation of the Moore-Penrose Inverse matrix

- **getSVD_S, getSVD_U and getSVD_Vt** return the three matrices in the singular value decomposition of a matrix, Vt being shorthand for V transposed or V*as we and others usually denote it. Note that the matrices returned from the mtj SVD class have to be converted through a denseToSparse method in order for them to be Colt SparseDoubleMatrix2D matrices. Analogously, the createSingleValueDecomposition method invokes a sparseToDense method on a Colt matrix in order that it can be processed in mtj.

- **createCommuteTimesMatrix**, which calculates and returns the Commuting Times distance matrix form the formula $C(v_i,v_j) = 2m(l^{+}_{ii} + l^{+}_{jj} - 2l^{+}_{ij})$, where m is the number of edges of G and l^{+}_{ij} is the $(i,j)^{th}$ entry of the Moore-Penrose pseudoinverse L^{+} of the Laplacian matrix L of G. It is understood that the matrix fed as parameter to the method is the Moore-Penrose pseudoinverse of the Laplacian matrix of a graph (the parameter is named APlus in the method!).

```
SparseDoubleMatrix2D CTmatrix = new SparseDoubleMatrix2D(numVertices,numVertices);
for (int i=0;i<numVertices;i++) {
   for (int j=0;j<numVertices;j++) {
   CTmatrix.set(i, j, APlus.getQuick(i, i) + APlus.getQuick(j, j) - 2 * APlus.getQuick(i, j));
   }
}
return roundedMatrix(CTmatrix);
```

Code Example 8: Creation of the Commute Time matrix

In order to maintain a level of consistency the matrices returned by the differing methods have been rounded up to a predefined number of decimal places. This has been achieved by using the method roundedMatrix:

- **roundedMatrix,** which rounds small entries (absolute value less than 1.0E -6) in a matrix down to zero. This is an effort to combat extremely small entries that sometimes occur in the matrix calculations because of rounding errors inherent in Java. These entries should really be zero and an almost zero entry instead of a zero entry can have enormous consequences, e.g. when calculating the inverse of a matrix eigenvalue.

5.4.8 The ClusterUtilities Class

The constructor of this class expects a visualisation viewer as a parameter. The visualisation viewer is accessed by other methods within the class. This class is the result of refactoring and expanding the original Jung clustering demo application.

The class has four methods:-

- **clusterAndRecolor**, this method has 6 parameters:-
  - a layout object (the JUNG defined layout),
  - the number of edges to remove or the number of clusters required, depending on whether the algorithm is Girvan-Newman or K-Medoids,
  - A Boolean indicating whether or not the visualization is to be grouped,
  - the Commuting Times matrix (i.e. the distance function),
  - which algorithm is to be used and
  - a seed used by the K-Medoids algorithm to calculate the starting values of the medoids.

  The method removes the graph being processed from the layout object and resets the layout object. If the Girvan-Newman algorithm has been selected, the JUNG implementation of this algorithm is used to recalculate the graph clusters which are emphasized visually by assigning a common colour to all the vertices within a cluster.
  Alternatively, if the chosen algorithm is the K-Medoids algorithm, the MatrixClustering class (See "Section 5.4.10 The MatrixClustering Class") is used to create an object that allows the application of the K-Medoids algorithm from the WEKA package. Again, the clusters are emphasized visually by the assignment of a common colour to all the vertices within a cluster.

  The individual colours used to paint the vertices in distinct clusters are retrieved via the method getColor which returns a randomly-generated colour.

```
MatrixClustering clusterer = new MatrixClustering();
clusterer.runKmedoids(matrix, clustersOrEdgesToRemove);
int assignments[];
assignments = clusterer.getAssignments();
Set<Number> vertices = new  HashSet<Number>();
for (int i = 0; i < assignments.length; i++) {
                    if (assignments[i] == j ){
                                        vertices.add(i);
                    }
}
Color c = getColor();
colorCluster(vertices,c);
if ( groupClusters == true ) {
   groupCluster(layout, vertices);
}
```

Code Example 9: Grouping the clusters before selecting colours for each cluster

- **ColorCluster**, which assigns a colour to each vertex according to which cluster the vertex belongs to.
- **groupCluster**, which groups the clusters within the assigned layout and repaints the view using the visualization viewer.

5.4.9 The MatrixConversion Class

This class is used to convert a matrix from the Colt package format to the mtj package format. The class has two methods:-

- **SparseToDense** converts an object of Colt class SparseDoubleMatrix2D to an object of mtj class DenseMatrix.
- **DenseToSparse** converts an object of mtj class DenseMatrix to an object of Colt class SparseDoubleMatrix2D

5.4.10 The MatrixClustering Class

This is the class that brings together the K-Medoids and hierarchical clustering algorithms. This class has its own main method where it is possible to open a graph in the Pajek net format and cluster the resulting commute time matrix using either the K-Medoids or the Hierarchical Clustering algorithm.

The class contains three instance variables with methods to get and set them:-

- **sizes**, an array of cluster sizes
- **assignments**, an array indexed by the vertex indices and containing values which indicate which cluster a vertex is currently assigned to.
- **numberOfClusters** is the number of clusters the K-Medoids algorithm should find.

The main method opens a dialog box where it is possible to choose a graph file. The selected graph file is opened and its Commuting Distances matrix generated. A WEKA Instances object is created containing an instance corresponding to each vertex in the graph. The data from the Commuting Times matrix is then fed into a CommuteTime object which in itself is an instance of the WEKA EuclideanDistance class. The Instances and the CommuteTime objects are then passed on to the appropriate clustering algorithm.

```
ReadGraphDataI myGraphData = new ReadGraphData();
LoadGraphI myGraphLoader = new LoadGraph();
Graph<Number, Number> graph = null;
SettableTransformer<Number,Number> edgeWeights = null;
SparseDoubleMatrix2D matrix;

try {
    graph = myGraphLoader.loadGraphPnr(myGraphData.openFile(null,null));
    edgeWeights = myGraphLoader.getEdgeWeights();
    if (graph == null){
      System.exit(0);
    }
} catch (IOException e) {
}
LaplacianConstructionI myLaplacianmatrix = new LaplacianConstruction();
myLaplacianmatrix.constructMatrix(graph, edgeWeights);
matrix = myLaplacianmatrix.getCommuteTimes();

ArrayList<Attribute> vertex = new ArrayList<Attribute>(1);
vertex.add(new Attribute("Vertex"));
Instances ourInstances = new Instances("Matrix",vertex,matrix.columns());

for(int i = 0; i < matrix.columns();i++){
                    Instance inst = new DenseInstance(1);
                    inst.setValue(0,i);
                    ourInstances.add(inst);
}

CommuteTime cd = new CommuteTime(ourInstances,matrix);
//kMedoids(ourInstances,cd,100,4,11,false);
Hierarchical(ourInstances,cd,0,0,7);
int mySizes[] = getSizes();
int myAssignments[] = getAssignments();
```

Code Example 10: Main method of ClusterMatrix devised to test the K-medoids and Hierarchical algorithms

The K-Medoids method runs the K-Medoids algorithm using the data in Instances and the CommuteTime distance object. It is also possible to choose the number of iterations, the number of expected clusters and an initialization seed as well as initiating the algorithm with medoids calculated from minimum distance sums. (See “Section 3.5.2 K-Medoids”) The K-Medoids algorithm is included in the source code and was created by us in a previous project. The algorithm is programmed in accordance with the WEKA guidelines but it is not part of the official WEKA software.

When using the Hierarchical Clustering method, it is possible to alter the number of iterations and the link type, (Single, Complete, Average, Mean, Centroid, Ward, Adjusted Complete, Neighbour Joining) which are represented with the digits 0..7. The Hierarchical Clustering algorithm is available as part of the WEKA package. (Note that although eight link types are available, Centroid and Ward are not compatible with the Commuting Times distance. The reasons for this are explained in “Section 3.5.3 Hierarchical Clustering”.)

Both methods print their results to the console. K-Medoids lists the clusters and the number of vertices within each cluster, while the Hierarchical Clusterer creates and prints its results in the Newick format. In order to visualize the resulting clustering hierarchy, this Newick string must be fed to a viewer that can

parse it and convert the contents to e.g. a dendogram. Our preferred parser is referenced in “Section 7.1.2 Results of Hierarchical Clustering”.

In order to be able to use the K-medoids from within the Gui framework the method runKmedoids has been added. This method requires three parameters, a Commuting Times distance matrix, the number of clusters desired and an initialization seed. The method creates a WEKA Instances object and a CommuteTime object and then calls the K-Medoids method with the number of iterations set to 100. The number of iterations required has not been incorporated into the Gui. Our tests thus far indicate that the K-medoids algorithm converges to a stable collection of medoids and clusters after a small number of iterations, so 100 seems an ample bound for the number of iterations.

6 Test Material

As stated in the introduction to the previous section, the primary goal of our project has been to implement the Commuting Times Distance Metric and to ally it to a K-Medoids implementation and to Hierarchical Clustering with an array of link types and then to test these distance-based clustering methods on an array of graph data. In this section we describe the graph data collected for testing.

6.1 Pajek NET Format

An early decision was to store the graph data in the Pajek NET format (Batagelj & Mrvar, Ljubljana). This format or, perhaps more accurately, small family of formats is appealingly simple and seems to be fairly widely adopted. The JUNG graph processing software package (See “Section 5.3 Imported Libraries and their Function.”) accepts graph data in Pajek NET formats and this was a prerequisite for our selection.

A Pajek file is a line-based text file describing a single graph. It consists of a list of vertices followed by a list of edges. The definition of the vertices starts with the declaration “*Vertices N”, where N is the number of vertices in the graph. Each of the following lines describes a single vertex/node with a mandatory unique identifier and an optional label, e.g.

*Vertices 34

1 “1”
2 “2”
3 “3” etc.

An even simpler version allows the vertices to be defined by the declaration line alone, provided the subsequent definition of the edges is compatible with an implicit indexing of the vertices from 1 to N. We have adopted this simpler version in the graphs we have ourselves constructed and which are described in “Section 6.2.1 Constructing Graphs for Testing Purposes.”

The end of the list of vertices and the start of the list of edges is marked by an edges declaration line. Each of the following lines defines a single edge in terms of a pair of integers interpreted as the indices of the vertices at either end of the edge. If the line contains a third number, it is interpreted as the weight of the edge.

*Edges 1 2 1 3 1 4	*Edges 1 2 14 1 3 7 1 4 99
A simple list of non-weighted edges	*A simple list of weighted edges*

6.2 Three levels of size and complexity

It is not trivial to verify an implementation of a graph clustering algorithm. If the implementation of the algorithm contains errors, these will cloud the analysis of the results given by applying the algorithm on a particular dataset and may lead to inaccurate or even spurious conclusions. It would seem necessary to make attempts to test the implementation of an algorithm before applying it to "real" data. Our testing attempts have centred on applying the algorithm to artificially constructed graphs of limited size with "natural" clusters ("Section 6.2.1 Constructing Graphs for Testing Purposes.") though we have become aware that what these tests actually reveal is more complex than whether the algorithm implementation does or does not contain programming errors. Nevertheless, such graphs have enabled us to painstakingly validate entries of both intermediate matrices and the final Commuting Times distance matrix in our implementation and to check that the K-Medoids and Hierarchical Clustering algorithms were fed these distance measures correctly.

At some stage an algorithm must be tested with real data if it is to prove its worth. Again, with the aim both of verifying the implementation as well as learning about the appropriateness of the algorithms to the field of application, two small datasets were selected ("Section 6.2.2 Small sets of real data (Zachary, Dolphins)"):-

- Zachary, a dataset with 34 vertices and 78 edges representing the members of a karate club
- Dolphin, a dataset with 62 vertices and 159 edges representing the members of a pod of dolphins being studied in a particular location off the coast of New Zealand.

Finally, a dataset consisting of e-mails between 150 employees of the now defunct Enron Corporation was the largest set of data on which our implementation has been tested ("Section 6.2.3 Larger set of real data (Enron)").

We conclude this section with some remarks about problems which may arise when testing with much larger datasets ("Section 6.2.4 Large sets of data.").

6.2.1 Constructing Graphs for Testing Purposes.

Initially, as a means of verifying the implementation of our Commuting Times distance and K-Medoids code as well as the Girvan-Newman implementation in JUNG and the hierarchical clustering algorithm with different link types in WEKA, we constructed test graphs of limited size with "natural" clusters. Such graphs were constructed by linking standard "dense" graphs with single edges so that the natural clusters are the standard subgraphs of the composite whole. As dense subgraph building blocks, we used complete graphs and complete bipartite graphs which are defined and illustrated below.

A **complete graph** is a simple graph in which every pair of distinct vertices is adjacent. The complete graph on n vertices is usually denoted by K_n and in it, each vertex has degree n-1. The six complete graphs K_2 through to K_7 are illustrated below [12] .

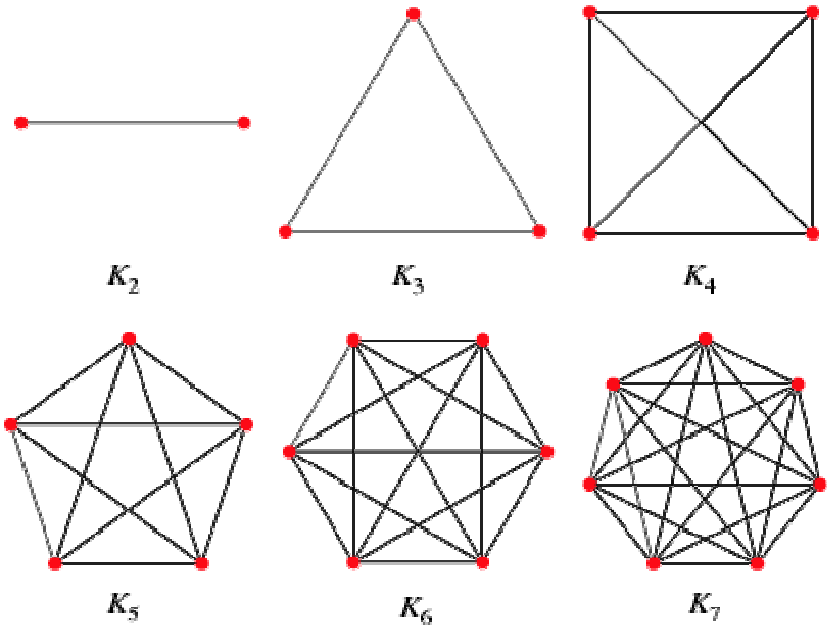


Figure 12: The six complete graphs K2 through to K7

A **bipartite graph** is a graph G in which the vertex set V(G) may be divided into two disjoint sets V_1 and V_2 in such a way that every edge of G joins a vertex of V_1 to a vertex of V_2. If every vertex of V_1 is joined to every vertex of V_2 and G is simple, G is called a **complete bipartite graph**. A complete bipartite graph is usually denoted by $K_{m,n}$, where m and n are the number of vertices in V_1 and V_2. The graphs $K_{3,4}$ and $K_{4,4}$ are illustrated below[13].

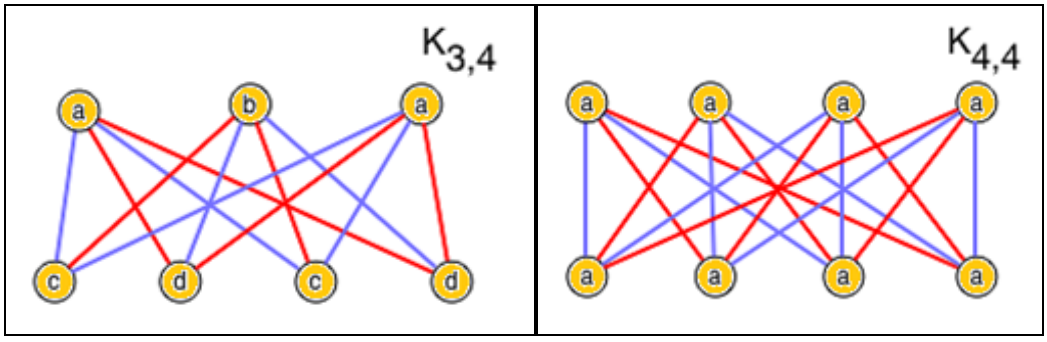


Figure 13: The complete bipartite graphs K3,4 and K4,4

The first step in creating test material for our graph clustering algorithm(s) was to manually create a small library of composite test graphs using the building blocks and technique described above. These graphs are to be found on the Google code website under (src) or in the code delivered with this report **Masters**

[12] Source: http://mathworld.wolfram.com/CompleteGraph.html
[13] Source: http://gilleain.blogspot.dk/

Project\graphClustering\src\main\resources\datasets [14]. In this library, the naming conventions adopted are Cn for the complete graph K_n, CBm_n for the complete bipartite graph $K_{m,n}$ and "-" as a separator. Thus, C4-C4-C4(.net) represents a graph comprising three K_4 subgraphs, where there are single edges linking the first subgraph to the second and the second to the third. CB3_5-CB2_3-C4-C4(.net) represents a graph composed of four subgraphs, $K_{3,5,}$ $K_{2,3}$, K_4 and K_4 where each subgraph in the sequence is linked to the next by a single edge.

Note that the construction technique described above results in a "chain" of subgraphs. The testing revealed that the addition of a single edge linking the last subgraph in the chain to the first sometimes made a difference to the discovery of the "natural" clusters (this has also been observed and noted in (Rattigan, Maier, & Jensen, 2007)), particularly in tests where an algorithm was asked to find a number of clusters other than the "natural" number. The suffix "+1" was adopted in the naming conventions to indicate the addition of the extra edge which converted the composite graph from a "chain" of subgraphs to a "circuit" of subgraphs. Adding such an extra edge to the composite graph contained in the file CB3_5-CB2_3-C4-C4(.net) will thus result in a file named CB3_5-CB2_3-C4-C4+1(.net)

With hindsight, the "circuit" of subgraphs in which each subgraph has the same connection pattern to the rest of the composite graph seems the more appropriate test graph in the majority of circumstances.

The level of ambition achieved in this project for constructed test graphs has been a small library of composite graphs as described above. As a natural extension to this, we have been considering programming a composite graph generator. The motivation for this and what it might generate and how are briefly described in (Rattigan, Maier, & Jensen, 2007).

6.2.2 Small sets of real data (Zachary, Dolphins)

The first small set of real data used to test our implementation is the Zachary data set is available in the Pajek format from several sources on the internet[15] and generated in connection with the study behind the article (Zachary, 1977). Zachary observed 34 members of a karate club over a period of two years. During the course of his study, a disagreement developed between the administrator of the club and the club's instructor, which ultimately resulted in the instructor leaving and starting a new club, taking about half of the members of the original club with him. Zachary constructed a network of friendships between members of the club, using a variety of measures to estimate the strength of ties between individuals. We use a simple unweighted version of his network containing 78 edges and with the instructor and the administrator represented by the first and last vertices respectively.

The second small set of real data[16] used in our testing is also available on the internet in the Pajek format. The underlying data was generated in connection with the study behind the article (Lusseau, et al., 2003). The associations in the title of the article were based on observations about how often individual dolphins were seen together in a school and comparing this frequency with the expected frequency of the two appearing in a school, should the schools of dolphins in the population be formed randomly. Only dolphins which survived the first year of the study were included in the statistical analysis. A threshold was defined for when the frequency of the two dolphins in a school was considered significant and this pair of dolphins

[14] Code repository: http://code.google.com/p/graph-clustering-ra/source/browse/
[15] Source: http://networkdata.ics.uci.edu/data/karate/
[16] Source: http://vlado.fmf.uni-lj.si/pub/networks/data/mix/mixed.htm

was then considered a “preferred companionship”. The graph data maps the preferred companionships. It consists of 62 nodes representing the dolphins in the population and 159 edges.

A community of dolphins is often referred to as a pod. We adopt this nomenclature and extend it referring to a singleton dolphin as an Ipod.

6.2.3 Larger set of real data (Enron)

Our implementation was also tested on a dataset significantly larger than the two described in the previous section. The original data consisted of 619,446 e-mail messages emanating from 158 e-mail addresses. The contents of the box below are quoted from an introduction to the dataset to be found on the internet [17]

> This dataset was collected and prepared by the CALO Project (A Cognitive Assistant that Learns and Organizes). It contains data from about 150 users, mostly senior management of Enron, organized into folders. The corpus contains a total of about 0.5M messages. This data was originally made public, and posted to the web, by the Federal Energy Regulatory Commission during its investigation.
>
> The email dataset was later purchased by Leslie Kaelbling at MIT, and turned out to have a number of integrity problems. A number of folks at SRI, notably Melinda Gervasio, worked hard to correct these problems, and it is thanks to them that the dataset is available. The dataset here does not include attachments, and some messages have been deleted "as part of a redaction effort due to requests from affected employees”. Invalid email addresses were converted to something of the form user@enron.com whenever possible (i.e., recipient is specified in some parseable format like "Doe, John" or "Mary K. Smith") and to no_address@enron.com when no recipient was specified.

A detailed description of the dataset is to be found in (Klimt & Yang, 2004).

This dataset was downloaded as a MySQL dump file from the internet [18]. This file was loaded into a MySQL database and the contents of the tables analysed and pre-processed before being converted to graph data.

As a first pre-processing step, the subject and content of the e-mails were discarded. A different analysis than ours might e.g. have concentrated on e-mails whose subject and content contained certain key-words.

Our objective was to analyse the social network of Enron employees. To that end, only e-mails sent to an Enron employee, characterized by an e-mail address of the form “*@enron.com” where * represents a user name, were retained and recipients of these e-mails external to Enron were removed. In addition, e-mails sent by employees to themselves were discarded the data to avoid loops in the converted graph data. From the resulting data, Enron e-mail addresses which neither sent nor received one of the retained e-mails were discarded (though they may in themselves be useful as a first identification of “outliers”) so that the converted data generated a connected graph. When these steps were complete, the data consisted of 150 employees and 40,490 e-mails.

It should be noted that the large number of e-mails in the dataset and the subsequent large number of edges (1526) in the graph derived from the dataset mitigates against the extraction of clusters. As we

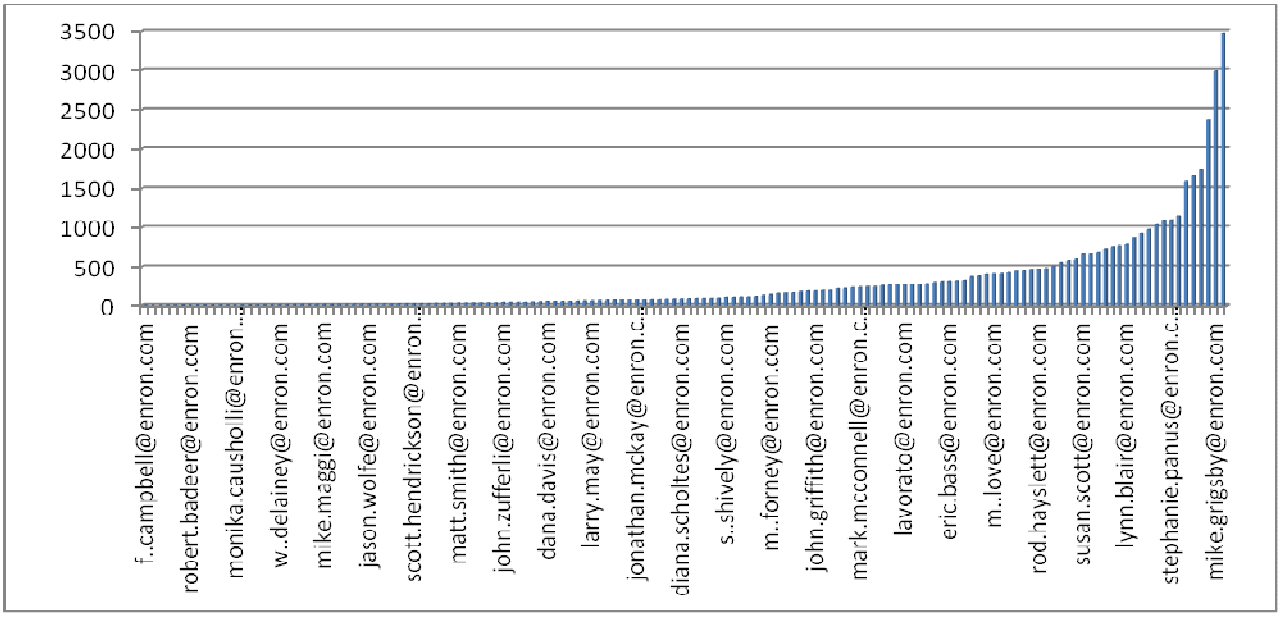
[17] Source: http://www.cs.cmu.edu/~enron/
[18] Source: http://www.isi.edu/~adibi/Enron/Enron.htm

observed in “Section 4.2 What is a cluster in a graph?“, if the number of edges in a graph is much larger than the number of vertices, the distribution of edges among the vertices has a tendency to become more homogeneous and potential clusters less distinct from the surrounding graph.

In our analysis of the Enron social network, we have chosen not to distinguish between the sender and the recipient of an e-mail. When the data is converted, the graph thus formed is undirected.

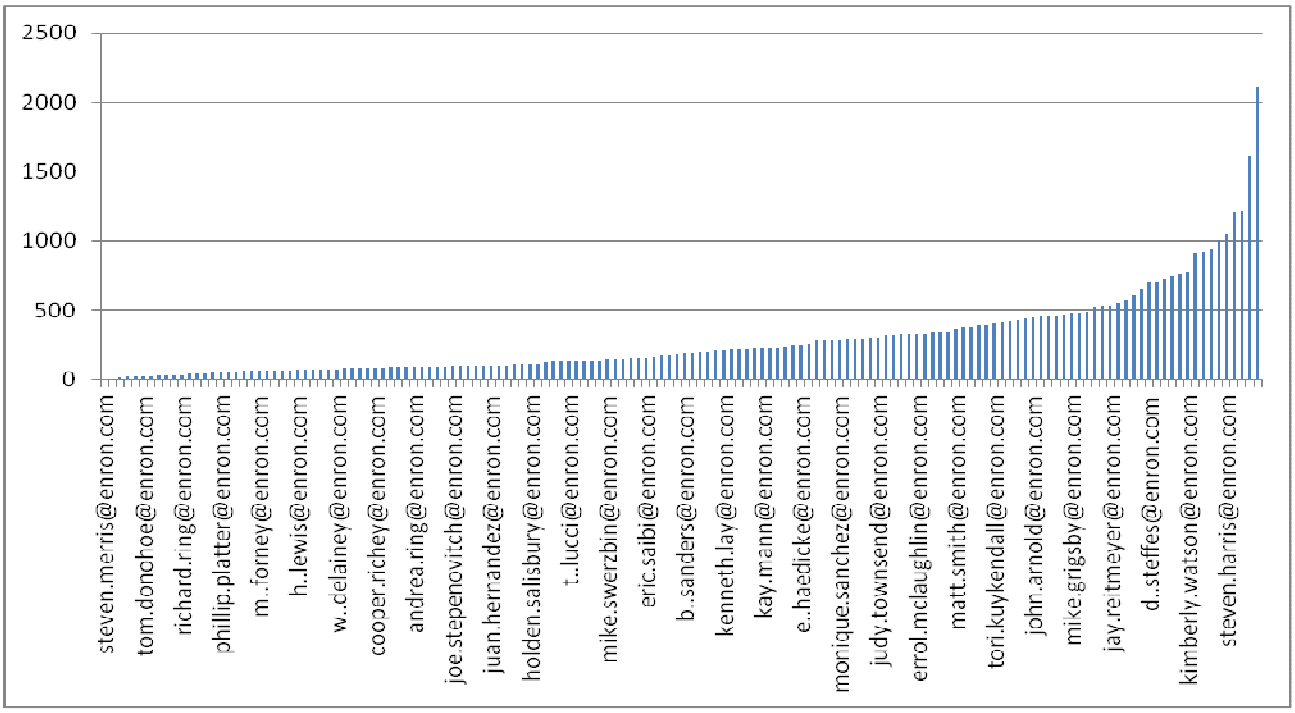
Initially, we ignored the number of e-mails between employees and created an unweighted graph with an edge joining any pair of employees who had exchanged one or more e-mails. As our work with this data progressed, we became interested in how much information had been discarded by this strategy and analysed the distribution of e-mails between employees. The charts below illustrate respectively the distribution of e-mails sent and received.

Distribution of outgoing e-mails.



Histogram 1: Distribution of outgoing e-mails from the Enron data set

Distribution of incoming e-mails.



Histogram 2: Distribution of incoming e-mails from the Enron data set

The number of e-mails sent by an individual Enron employee ranges from 2 to 3400 and the range of the number of e-mails received by an individual employee is from 2 to 2100. Speculation about why these variations were so considerable led to the thought that it might be influenced by the period of employment and that it might be an idea to counter the effect of variations in this by restricting our attention to e-mails sent during a particular time interval. This we have not done but we have converted the data a second time to create a weighted graph in which each edge is annotated with a weight corresponding to the number of e-mails exchanged between the employees whose vertices are incident to the edge.

The ER (Entity-Relationship) diagram below (Figure 14) depicts the table structure used in the pre-processing of the Enron dataset.

- The four table employeelist, message, recipientinfo, referenceinfo were imported from the MySQL data dump.
- The table u_graph was created to enable the extraction of both unweighted and weighted edges between employee nodes/vertices. It removes the direction from sender to receiver. In this table, both columns node1 and node 2 could either be a sender or a receiver but there is only one line in the table for any particular value of the node1-node2 pair (i.e. If there is a line in u_graph with node1 = eid1 and node2 = eid2, there is no line in the table with node1 = eid2 and node2 = eid1.)
- To populate the table u_graph with data the view employee_send_receive was constructed.

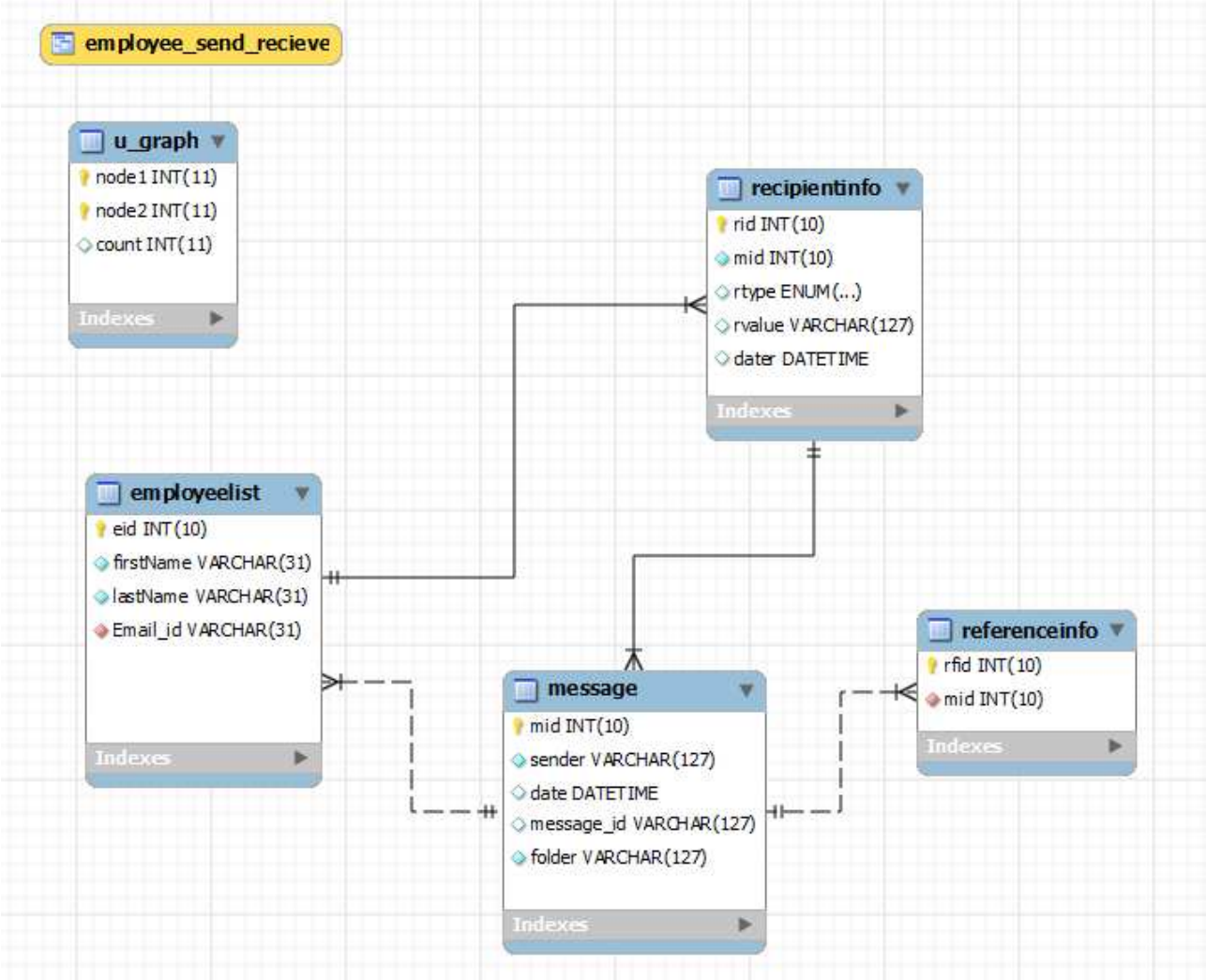


Figure 14: ER diagram of the table used to extract the Enron test data from the Enron dataset

We give a brief description of the tables and views in the data and the relevant columns:

- Table employeelist
  - o eid = unique employee id.
  - o Email_id = a string that contains the employees email address (Indexed to improve performance)
- Table message
  - o mid = unique message id(Email)
  - o sender = the email address for the originator of the message (Foreign key Email_id from employeelist) (Indexed to improve performance)

- Table recipientinfo
    - rid = the employee id of the recipient of a message (Foreign key eid from employeelist)
    - mid = the message id (Foreign key mid from message)(Indexed to improve performance)
- Table referenceinfo (not used)
- View employee_send_receive: This view is the result of a query which selects the sender and receiver of all messages. The objective was to create a flexible view used in the first instance to populate the table u_graph. (A print-out of a small portion of this view is shown below together with the SQL code used to populate it Figure 15 and Figure 16.)
    - seid = the employee id of the message sender
    - sender = email address of the sender
    - m_mid = message id
    - r_mid = recipient message id (used for verify that the message sent is the message received)
    - rvalue = email address of the recipient
    - reid = the employee id of the recipient

seid	sender	m_mid	r_mid	rvalue	reid
1	robert.badeer@enron.com	847	847	mike.grigsby@enron.com	17
1	robert.badeer@enron.com	848	848	mike.grigsby@enron.com	17
1	robert.badeer@enron.com	217876	217876	jeff.dasovich@enron.com	73
1	robert.badeer@enron.com	218035	218035	jeff.dasovich@enron.com	73
1	robert.badeer@enron.com	219852	219852	jeff.dasovich@enron.com	73
2	kevin.hyatt@enron.com	868	868	bill.rapp@enron.com	30
2	kevin.hyatt@enron.com	873	873	darrell.schoolcraft@enron.com	28

Figure 15: Sample data seen in the employee_send_recieve view

```
select distinct
(select `e`.`eid`
    from `enron`.`employeelist` `e`
    where (`e`.`Email_id` = `m`.`sender`)) AS `seid`,
`m`.`sender` AS `sender`,
`m`.`mid` AS `m_mid`,
`r`.`mid` AS `r_mid`,
`r`.`rvalue` AS `rvalue`,
(select `e`.`eid`
    from `enron`.`employeelist` `e`
    where (`e`.`Email_id` = `r`.`rvalue`)) AS `reid`
from (`enron`.`message` `m` join `enron`.`recipientinfo` `r`)
    where (`m`.`sender` in (select `enron`.`employeelist`.`Email_id`
                            from `enron`.`employeelist`)
    and (`r`.`mid` = `m`.`mid`)
    and `r`.`rvalue` in (select `enron`.`employeelist`.`Email_id`
                            from `enron`.`employeelist`))
order by `m`.`mid`
```

Figure 16: The sql select used to define the employee_send_recieve view

- Table u_graph (The sql query used to populate the table is shown below Figure 17)
    - node1 = the employee id of a message sender or receiver
    - node2 = the employee id of a message receiver or sender
    - count = the number of emails sent or received by node1 or node2

```
insert into u_graph
SELECT  distinct s.seid, s.reid,
( select count(*) from employee_send_recieve x1 where x1.sender = s.sender and x1.rvalue = s.rvalue)
+
( select count(*) from employee_send_recieve x1 where x1.sender = s.rvalue and x1.rvalue = s.sender)

FROM `enron`.`employee_send_recieve` s
where s.sender not like s.rvalue
order by s.sender, s.rvalue
```

Figure 17: The sql query used to populate the u_graph table

To create a file from the Enron data in Pajek format, user data was extracted from the employeelist table and correspondence data was extracted from the u_graph table. The data extracted from these two tables was saved in comma separated files which were then merged into a single file and supplied with vertex and edge declarations to complete the Pajek format.

6.2.4 Large sets of data.

As outlined in (Fouss, Pirotte, Renders, & Saerens, Random Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation, 2007) the very nature of large datasets, containing large numbers of vertices and edges, and the resulting commute time calculations prohibit the use of pseudoinverse matrices for large datasets. The authors propose the use of several iterative techniques that can be used to reduce the reliance on a pseudoinverse matrix.

The way in which we calculate the final commute time matrix is a memory intensive operation. We started to investigate the limitations of the current implementation with regards to the performance of the algorithms. The practical problem that our experiments encountered was an exhaustion of memory. Our first step towards finding the problem would be to attempt to isolate the location of the problem, either in our own software or the mtj package.

7 Test Results

In the previous section we described the implementation of the Commuting Times distance metric and how this metric was allied to implementations of K-Medoids and Hierarchical Clustering algorithms. In “Section 6 Test Material”, we described the datasets to which these Graph Clustering techniques have been applied. We are now ready to discuss the results of the tests themselves but before we do this, we discuss briefly the presentation of these results.

7.1 Presentation of Test Results

The results of subjecting the test material described in ”Section 6.2 Three levels of size and complexity” to the K-Medoids, Hierarchical and Girvan-Newman algorithms are summarized below. More details are given

in the three Excel arks GirvanNewman.xslx, Hierarchical.xlsx and KMedoids.xlsx included in the electronic form of the project delivery. **Masters Project\Clustering results**[19]

7.1.1 K-Medoids results

As described in "Section 5.4.3 The RAGraphClustering Class", we have extended the clustering demo visualization GUI in the JUNG framework to display the results of the K-Medoid algorithm for any number of clusters selected by the operator. This extension of the JUNG GUI has enhanced enormously our ability to test the K-Medoids algorithm. As we conduct tests and become aware of the clusters identified by the K-Medoids algorithm, these clusters can be emphasized by re-arranging the pattern of vertices on the screen to group them together visually and to separate them from the other vertices. We can then fix a value of K and run through the range of seeds, noting for which seed values the algorithm reveals certain clusters. This in itself allows us to run efficiently through a range of experiments for a particular K but the pre-processing of the geometry on the screen enables us to spot both previously identified clusters and new ones much more easily than would otherwise be the case. The fact that K-Medoids experiments are much quicker to execute and to interpret than they previously were enables us to consider different forms of simple statistical analysis when collecting and interpreting the results (e.g. percentage measures for a particular cluster appearing for a particular value of K for a fixed number of different seed values).

The graph diagrams illustrating the K-Medoids results are screen dumps of two diagram formats generated from the adaptation of this GUI.

7.1.2 Results of Hierarchical Clustering

The WEKA implementation used for Hierarchical clustering generates output in Newick format. This format was worked out in the agreeable setting of the Dover, New Hampshire branch of Newick's Lobster House[20] and is described e.g. on the Wikipedia [21].

It is a format, or perhaps more accurately, a family of formats to represent graph-theoretical trees, of which the dendograms described in "Section 3.5.3 Hierarchical Clustering" are a particular example. In terms of the descriptions on the Wikipedia page, the format generated is the "distances and leaf names" variant (minus the semi-colon suffix!). The resulting string has been fed to the online proWeb Tree Viewer[22] to generate a dendogram which has been rotated 90 degrees to give the same x-y orientation as described in "Section 3.5.3 Hierarchical Clustering". We have adopted the recommendations to be found in the Weka Hierarchical Clustering class program notes and elsewhere to use branch-length on the y-axis for the neighbour joining link type and node-height for all other link types. We have considered creating two-dimensional graphs plotting the distance representation on the y-axis against number of merges on the x-axis in order to assist in the extraction of significant clusters from the whole hierarchy but have not as yet taken this step. It may be informative to do so, in particular with the Enron dataset, as the dendograms produced by the proWeb Tree Viewer are too large to be easily read and analysed. We have looked at other Newick format parsers but with limited success.

[19] Google Code web site: http://code.google.com/p/graph-clustering-ra/
[20] The Lobster House: http://newicks.com/
[21] Source: http://en.wikipedia.org/wiki/Newick_format
[22] Proweb Tree Viewer: http://www.proweb.org/treeviewer/

The only alternative we have found in any way useful was the DrawTree parser[23] which we have used to give a form of overview of the Enron data. The results of the two parsers are superficially compatible as one would hope. The problem with the DrawTree viewer is the opposite of that with the proWeb viewer; it gives an overview of possible clusters on a single screen but only allows a zooming factor of 400% which is too small to see the numbers labelling the nodes and edges in the visual representation.

We conclude with a proWeb generated dendogram of a size where the details can easily be seen. It is the dendogram corresponding to the ADJUSTED COMPLETE link type and the graph CB3_5-CB2_3. In this dendogram the vertices in the graph are indexed from 0(.0) to 12(.0). The other numbers represent difference in node height between the new cluster and the two clusters being merged; e.g. when the vertex 1(.0) is merged with the cluster consisting of the vertices (2(.0), 5(.0), 6(.0),7(.0)), it occurs at a node height of 0.00267 and at a height of 0.00178 above where merging of the cluster of four vertices occurred.

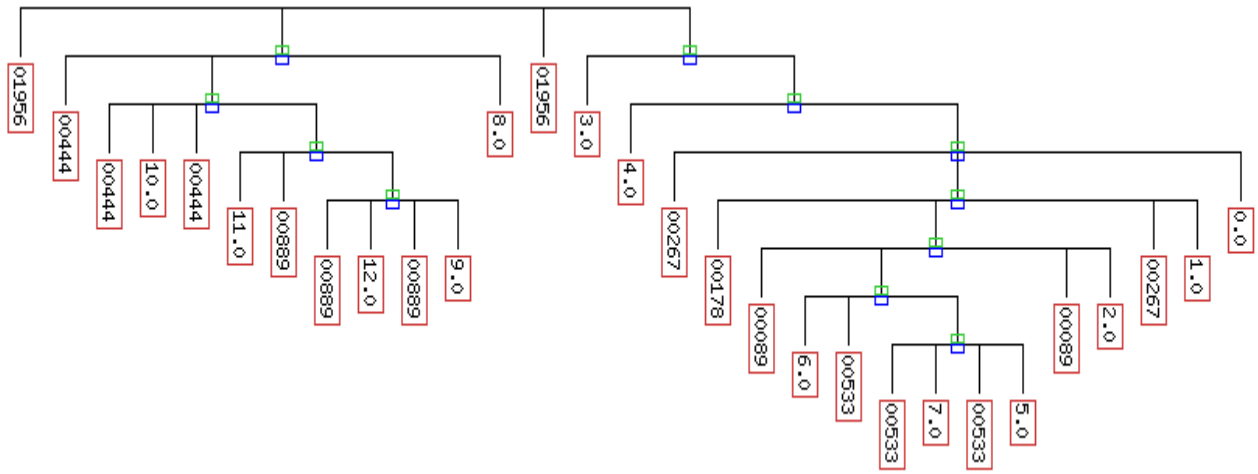


Figure 18: Example of a proWeb generated dendogram

7.2 The Actual Test Results Summarised.

7.2.1 Small graphs with natural clusters (K-Medoids).

The objective with testing K-Medoids with the small graphs with natural clusters which were created manually was to see if the algorithm could find the natural clusters in these graphs when K was set to the appropriate value. The results were mixed. Selecting the appropriate K and varying the seed, the natural clusters were sometimes found and sometimes not. Initially, the reason was thought to be that there was not enough density difference between the smallest clusters (e.g. CB2_3) and the edges connecting them. Experiments with graphs containing larger natural clusters (e.g., C6, C7) seemed to reduce the problem but not eliminate it. In the end, it was concluded that with the way the test graphs were constructed, the algorithm had difficulty moving a medoid from one natural cluster to another. If the initialization of medoids distributed two or more in the same natural cluster, then the algorithm was likely to terminate with one or more of the natural clusters split and others joined in one larger than expected cluster. We are

[23] Draw Tree Parser: http://www.phylogeny.fr/version2_cgi/one_task.cgi?task_type=drawtree

considering ways of measuring the ability of the K-Medoids algorithm to move a medoid from one natural cluster to another with graphs constructed in this and other ways. We conclude with a couple of examples of K-Medoids failing to find the natural clusters (Figure 19) in one of our manually constructed graphs and a couple of examples where it finds the natural clusters (Figure 20).

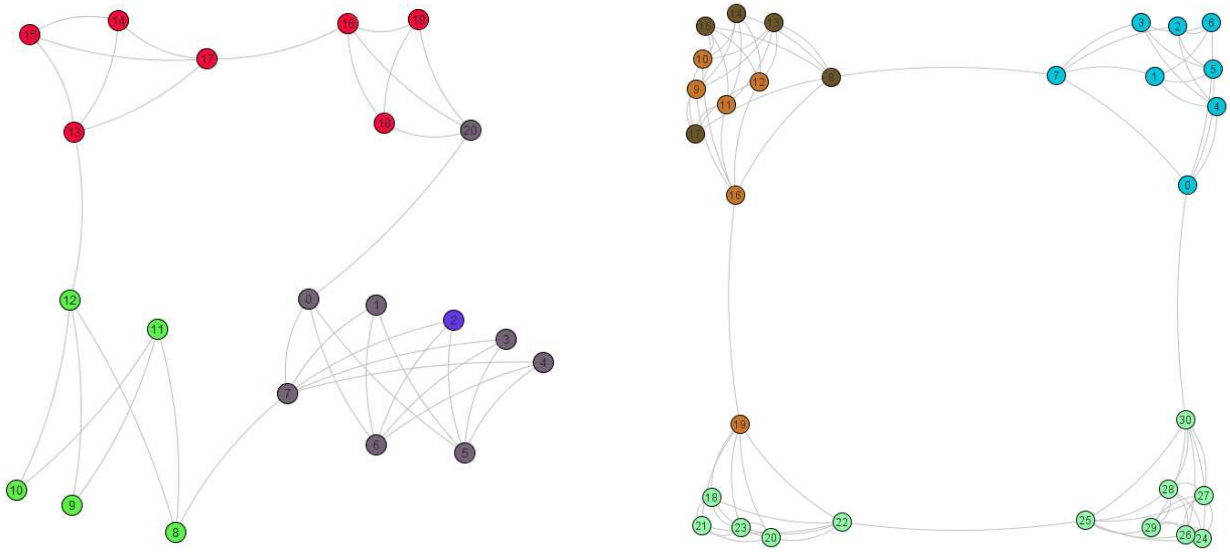


Figure 19: K-Medoids fails to find completely the four clusters in CB3_5-CB2_3-C4-C4+1 and CB4_4-CB5_5-C6-C7+1

.

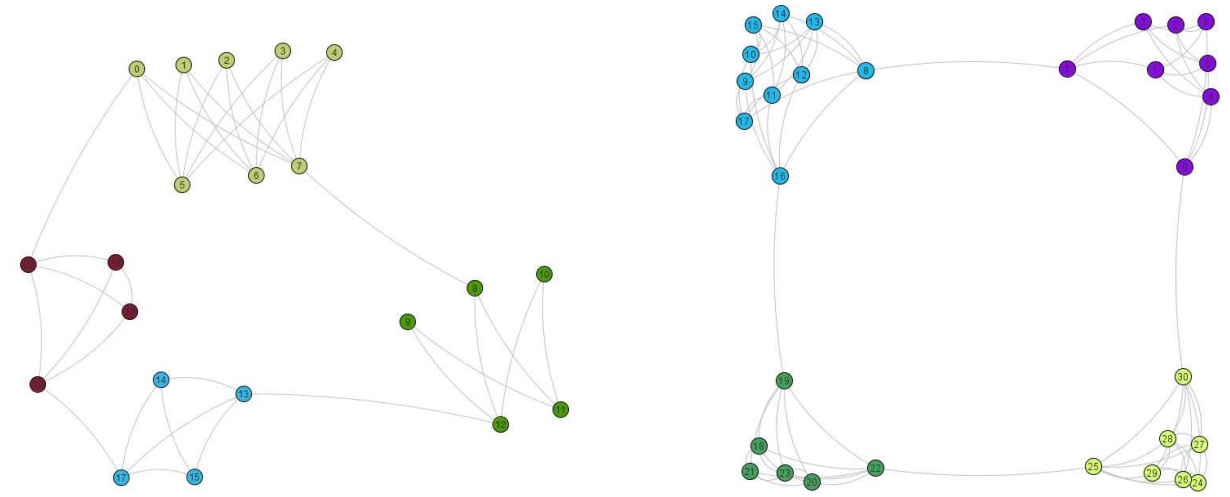


Figure 20: K-Medoids finds the four clusters in CB3_5-CB2_3-C4-C4+1 and CB4_4-CB5_5-C6-C7+1

7.2.2 Small graphs with natural clusters (Hierarchical Clustering)

The objective with testing Hierarchical Clustering with the small graphs with natural clusters which we created manually was to see if the algorithm could find the natural clusters in these graphs at the right level in the hierarchy, e.g. if the algorithm would have identified CB4_4, CB5_5, C6 and C7 when it had agglomerated the vertices in CB4_4-CB5_5-C6-C7+1 to four clusters. As described in “Section 3.5.3

Hierarchical Clustering “, we were able to test with six link types, SINGLE, COMPLETE, AVERAGE, MEAN, ADJUSTED COMPLETE and NEIGHBOR JOINING. Of these six, five were found to be good at identifying the correct clusters at the appropriate level in the hierarchy, the exception being NEIGHBOR JOINING. That NEIGHBOR JOINING should fail to perform well in these circumstances was perhaps not surprising as it a link type which seems to have been developed for the specific purpose of biological classification and we would not claim that our test graphs bear any resemblance to those that might be found in that area of application. There were single minor exceptions but otherwise the five other link types clustered close to perfectly! We conclude by including the dendogram generated by the proWeb viewer from the Newick String corresponding to graph CB4_4-CB5_5-C6-C7+1 and link type COMPLETE.

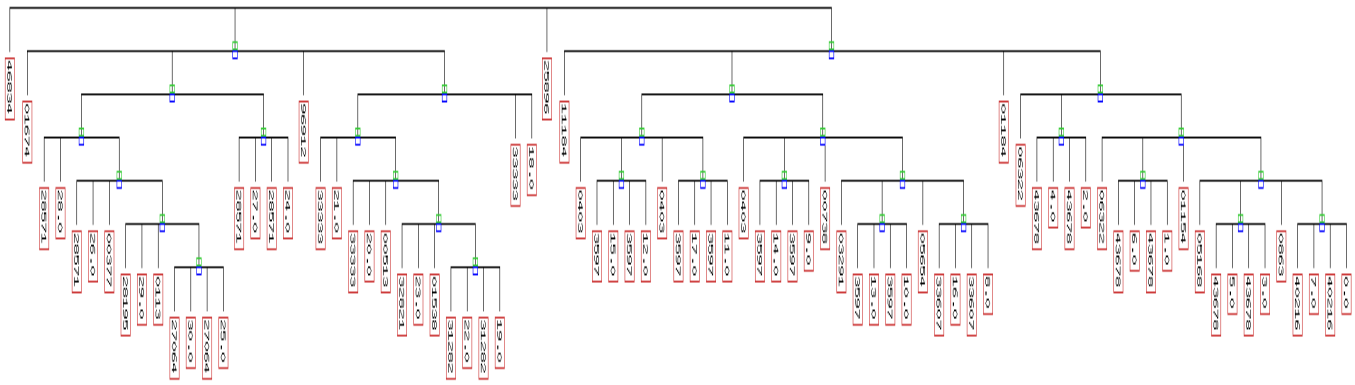


Figure 21: Dendogram generated by the proWeb viewer from the Newick String corresponding to graph CB4_4-CB5_5-C6-C7+1 and link type COMPLETE

7.2.3 Small Graphs with natural clusters (Girvan-Newman).

The simple test for the Girvan-Newman algorithm with the graphs generated with natural clusters was to see if it would remove all inter-cluster edges before it started removing edges from within clusters. This it did faultlessly. We illustrate the performance with the crucial point in a couple of edge-removing sequences:-

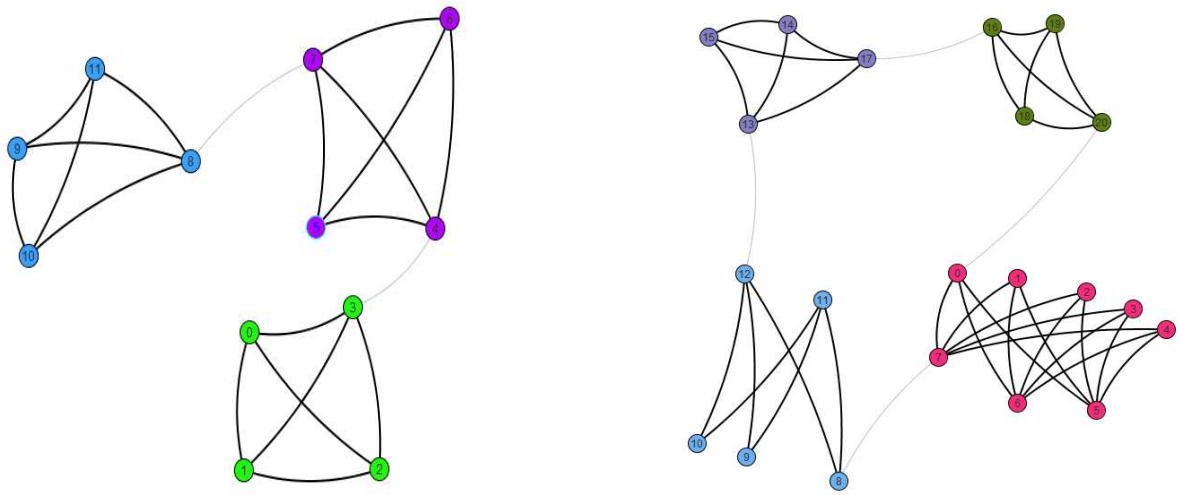


Figure 22 CB3_5-CB2_3-C4-C4+1.net with four edges removed and C4_C4_C4.net with two edges removed..

7.2.4 K-Medoids and the Zachary dataset

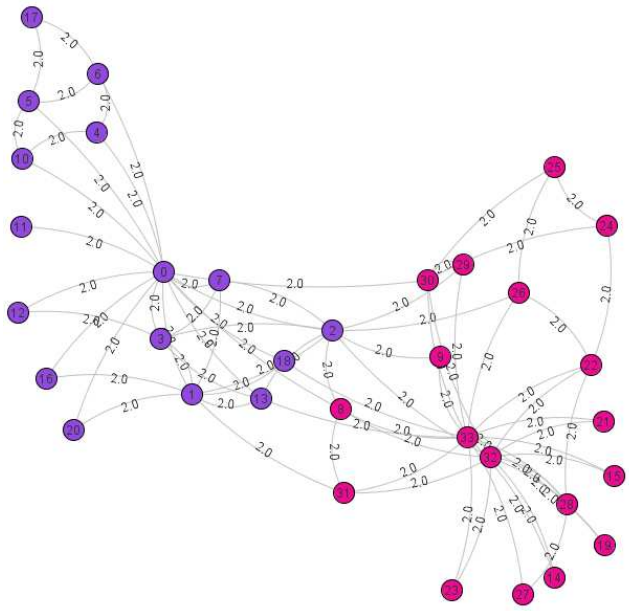


Figure 23: K-Medoids with 2 clusters on the Zachary dataset

The K-Medoids algorithm was able to identify with K=2 the two main clusters normally associated with the Zachary dataset (Figure 23), a cluster of size 18 including vertex 33, the Karate club administrator and a cluster of size 16 containing vertex 0, the club trainer for about 31 of the 100 seeds tested. With the deterministic start, these two clusters were identified. Furthermore, with K=3, the clusters found were these two clusters minus one or two vertices which formed the third cluster. There was a single difference with respect to the two clusters found by the Girvan-Newman algorithm (See "Section 4.4.3 Divisive Algorithms: The Girvan-Newman Algorithm"); vertex 2 was placed in the "trainer" cluster by K-Medoids and in the "administrator" cluster by Girvan-Newman.

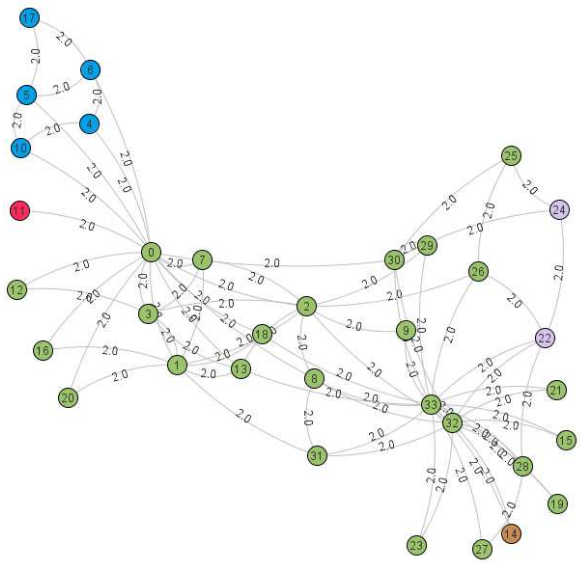


Figure 24: K-Medoids finding a cluster of 5 vertices on the Zachary dataset

A second cluster identified by the algorithm when K=2 was the cluster of size 5 (Figure 24) consisting of the vertices 4, 5, 6, 10 and 17. These are characterised by only being adjacent to some other members of the cluster and to the trainer. This cluster was found by 16 of the 100 seeds when K=2 and 20 of the 100 seeds when K=3. The following illustration shows that the cluster continues to be identified for seeds when K=5.

The least stable members of this cluster of size 5 are the vertices 4 and 10 which, however, have a strong tendency to be grouped together, either as a cluster in their own right or as members of the larger "trainer" cluster.

For larger values of K, the "administrator" cluster tends to disintegrate by losing individual vertices which form singleton clusters.

CONCLUSION: The clusters by the K-Medoids algorithm which would be considered for further analysis would be:-

1) The cluster of size 18 popularly designated the "administrator" cluster.
2) The cluster of size 16 popularly designated the "trainer" cluster.
3) The cluster of size 5, the five vertices 4,5,6,10,17, the five vertices only linked to the rest of the graph through vertex 0.

When we compare our 2 cluster results (Figure 23) with the results discussed in (Fortunato, 2010) we can see that the clustering is in full agreement with Zachary's bisection of the data. (Note that our vertex numbering begins with 0 and the numbering in (Fortunato, 2010) begins with 1.) According to (Fortunato, 2010) the misclassification of vertices 3, 9 and 10 (2, 8 and 9 in our dataset) which we avoid is prevalent in many clustering algorithms.

7.2.5 Hierarchical Clustering and the Zachary dataset

Hierarchical Clustering did not perform well with the Zachary dataset. In particular, no link type produced identifiable clusters which resembled the large "trainer" and "administrator" clusters identified by the K-Medoids and Girvan-Newman algorithms. Four of the link types (The exceptions were SINGLE and ADJUSTED COMPLETE.) clustered the five vertices (4,5,6,10,17) which were also identified as clusters by the two other algorithms.

The link type ADJUSTED COMPLETE identified (0,2,8,32,33) as a cluster which includes both the "trainer" and "administrator" vertices together with another "hub" (vertex 32) and two vertices which have "balanced" adjacency to vertices in the "trainer" and "administrator" clusters.

The link type NEIGHBOR JOINING identified (22,24,25,26,29,30) as a cluster which is a subcluster of the "administrator" cluster.

With this dataset and with all link types the hierarchical clustering tends overwhelmingly to agglomerate by the addition of single vertices to an existing cluster. In these cases the dendogram representation highlights visually the clusters which break the "add one vertex at a time" pattern. Thus, these may easily be regarded as the significant clusters as opposed to those which are formed prior to the largest jumps in node height.

CONCLUSION: In the light of the failure of Hierarchical Clustering to identify something resembling the "trainer" and "administrator" clusters, the conclusion may be to try other better suited algorithms than to further analyse the clusters identified by Hierarchical Clustering. Adopting a less negative viewpoint, the two clusters most likely to be considered for further analysis would be:-

1) The cluster of size 5, the five vertices 4,5,6,10,17, which was identified by four of the six link types.

2) The cluster of size 6 (22,24,25,26,29,30) identified by NEIGHBOR JOINING. Has this link type often considered as a specialist link type for biological classification revealed a grouping which the more standard link types and alternative algorithms failed to extract from the data?

7.2.6 Girvan-Newman and the Zachary dataset

Note that the two key personnel in the Zachary dataset, the instructor and the administrator, are represented by vertices 0 and 33 respectively.

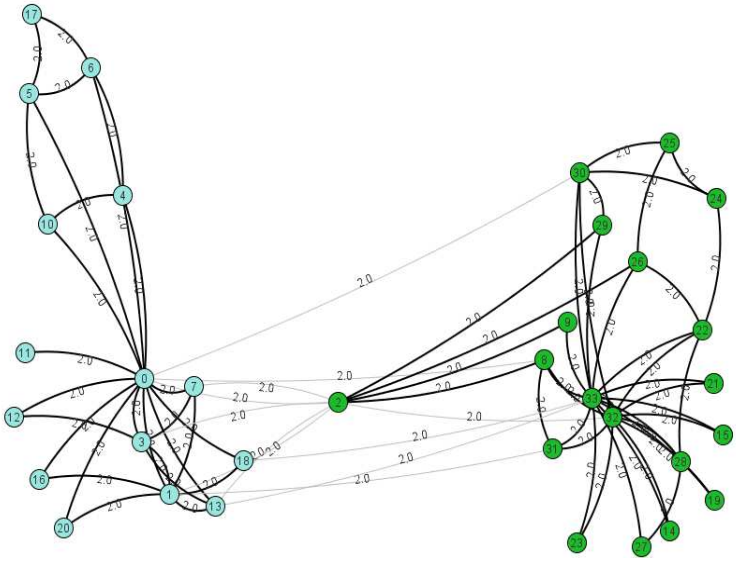


Figure 25: Girvan-Newman on the Zachary dataset with 2 clusters

- Number of clusters 2: Edges Removed 11: The clusters created are close to the most 2 cluster split generated by K-Medoids with vertex 2 clustered with the larger rather than the smaller cluster, giving clusters with 19 and 15 vertices. Vertex 2 has the appearance of a hub with strong connections to both clusters which probably explains why K-Medoids links it with the one cluster and Girvan-Newman with the other.

- Number of clusters 3: Number of Edges Removed 15: The third cluster is a singleton, vertex 9, from the cluster of size 19.

- Number of clusters 4: Number of Edges Removed 18: The fourth cluster is a division of the cluster of size 15 into two clusters of size 10 and 5, the cluster of size 5 also identified by K-Medoids (The five vertices 4,5,6,10,17, the five vertices only linked to the rest of the graph through vertex 0).

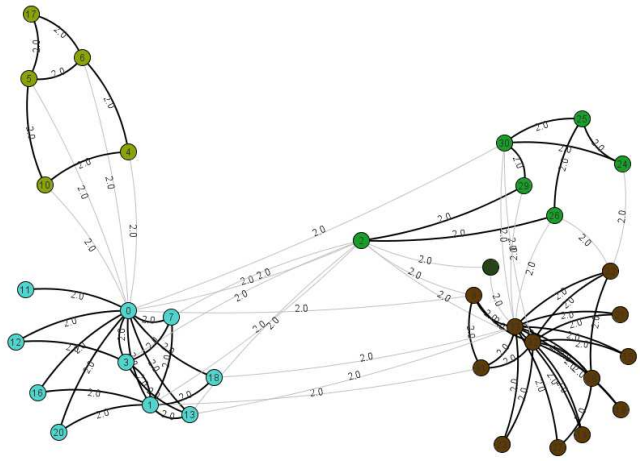


Figure 26: Girvan-Newman on the Zachary dataset with 5 clusters

- Number of clusters 5: Number of Edges Removed 24: The 5th cluster is a cluster of size 6 (Vertices 2, 24, 25, 26, 29, 30) separated from the cluster of size 19.

- The disintegration continues with extraction of singletons.

- Number of clusters 16: Number of Edges Removed 45: The five vertices 4,5,6,10,17, the five vertices only linked to the rest of the graph through vertex 0 are finally split into two clusters of size 2 and 3, a split which was observed several times in the K-Medoids experiments.

CONCLUSION: The clusters revealed from Girvan-Newman which would be considered for further analysis would be:-

1) The cluster of size 19 at the split into two clusters.
2) The cluster of size 15 at the split into two clusters.
3) The cluster of size 5, the five vertices 4,5,6,10,17, the five vertices only linked to the rest of the graph through vertex 0.
4) The cluster of size 6 at the split into five clusters, a cluster not identified by the K-Medoids analysis.

7.2.7 K-Medoids and the Dolphin dataset

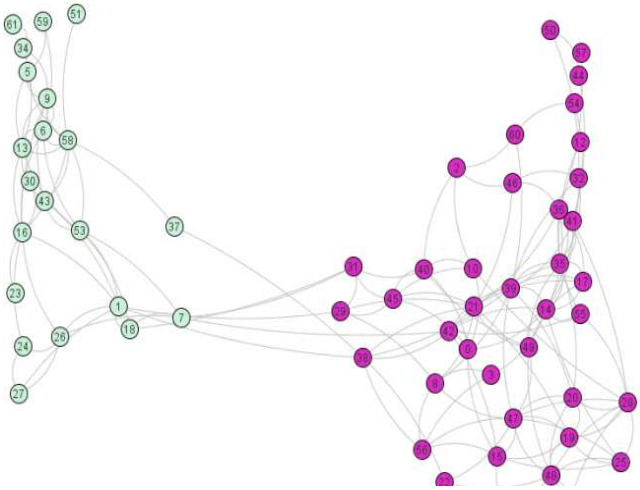


Figure 27: K-Medoids with 2 clusters on the Dolphin dataset

With K=2 the K-Medoids algorithm terminated with two clusters of sizes 41 and 21 for 51 out of 100 experiments with different seeds. These two clusters were also the result of the deterministic start.

With this division into two clusters (Figure 27), the vertex 7 is the least stable member of the smaller group. It is adjacent to the same number of vertices in both clusters and is the only vertex in the smaller cluster which was grouped with members of the larger cluster in a significant number of KMedoids experiments.

Other groupings which turn up regularly are:-

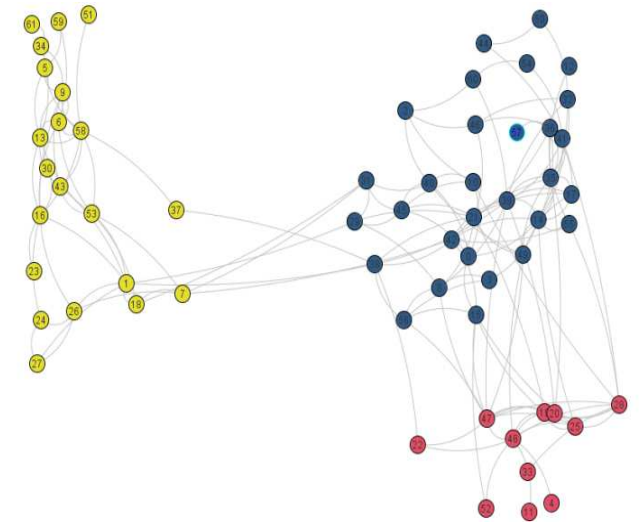


Figure 28: K-Medoids with 3 clusters on the Dolphin dataset

- 4,11,19,20,22,25,28,33,47,48,52, a cluster of size 11 (Figure 28) (with 2% of seeds with K=2, 6% with K=3, 8% with K=4 and 7% with K=5; with K=5, variants with one vertex less appear in an additional 9% of experiments.)

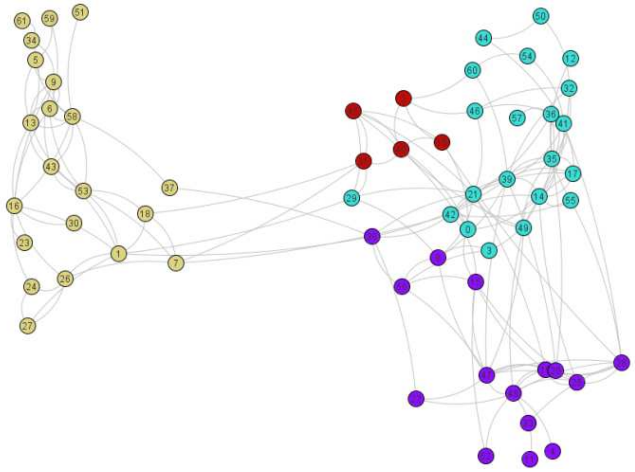


Figure 29: K-Medoids with 4 clusters on the Dolphin dataset

- 10,31,40,45, a cluster of size 4 (Figure 29) sometimes augmented with vertex 2, sometimes with vertex 29 (with K=3, 8% of seeds give the 2-variant and 2% the 29-variant, with K=4, 11% give the 2-variant and 4% the 29-variant, with K=5, 11% give the 2-variant and 8% the 29-variant)

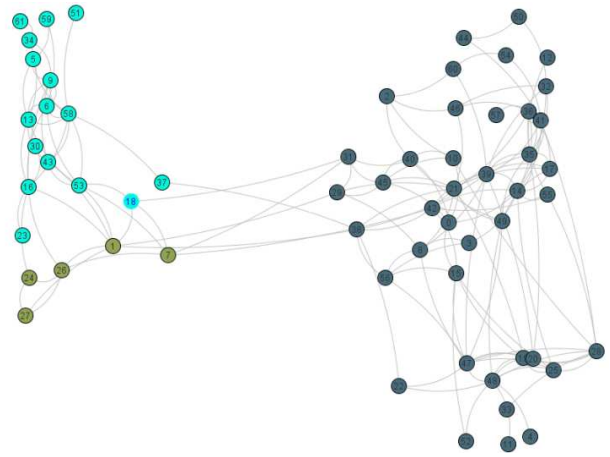


Figure 30: K-Medoids with 3 alternative clusters on the Dolphin dataset

- 1,7,24,26,27, a cluster of size 5 (Figure 30) sometimes augmented with 3, 4 or 5 extra vertices (with K=3, 5% of seeds give the cluster of size 5, with K=4, 8% of seeds give that cluster, with K=5, also 8% of seeds give that cluster).

The two cluster variants (**2**,10,31,40,45) and (10,**29**,31,40,45) are a simple example of a feature of the results generated by the K-Medoids algorithm which requires further consideration. The four vertices which are the common core of these two variants hardly ever appear as an identified cluster but the two variants appear sufficiently frequently that together they would represent the most statistically significant clusters for values of K less than 6 after the two large clusters of sizes 41 and 21.

The K-Medoids algorithm typically generates small variations of groupings which may or may not be individually significant and where the common core of these groupings and/or the set-theoretic union of these groupings or something in between may or may not have significance.

Biologically, there may be a reason why dolphins 10, 31, 40 and 45 are almost always in the company of either dolphin 2 or dolphin 29 but almost never both of them. On the other hand, it is also possible that the four dolphins share common traits which dolphins 2 and 29 exhibit to a lesser extent, so that the “real-life” cluster is the core of four. When considering which groupings to analyse further, it may not be sufficient

just to consider the most frequently appearing clusters; it may be necessary to analyse in some way clusters that are closely related to each other before then deciding which groupings are most promising for application-related post-processing.

CONCLUSION: The clusters by the K-Medoids algorithm which would be considered for further analysis would be:-

1) The cluster of size 41 appearing in over 50% of the experiments with K=2.
2) The cluster of size 21 appearing in over 50% of the experiments with K=2
3) The two cluster variants (**2**,10,31,40,45) and (10,**29**,31,40,45).
4) The cluster of size 11 consisting of vertices 4, 11, 19, 20, 22, 25, 28, 33, 47, 48 and 52.
5) The cluster of size 5 consisting of vertices 1, 7, 24, 26 and 27, possibly with some of the cluster augmentations.

7.2.8 Hierarchical Clustering and the Dolphin dataset

The K-Medoids and Girvan-Newman algorithms both seem to reveal a division of the Zachary dataset and a division of the dolphins dataset into two clusters. As we discussed in "Section 7.2.5 Hierarchical Clustering", the Hierarchical Clustering algorithms were unable to extract these two clusters with the Zachary dataset. Though none of the link types managed to extract the two dolphin clusters agreed upon by the other algorithms exactly, only the ADJUSTED COMPLETE link type failed to find any cluster that resembled one of the two. The most significant clusters found by the other link types were:-

- SINGLE: Two largish subclusters of the clusters found by the other two algorithms were extracted, one containing 27 vertices from the cluster of size 41 and the other 12 from the cluster of size 21.
- COMPLETE: Perhaps the most successful link type with the dolphins dataset (Figure 31). It extracted a cluster of size 22 which contained 20 of the 21 vertices in the smaller of the two large clusters found by K-Medoids and Girvan-Newman supplemented with the two vertices 29 and 31; the one vertex missing from the smaller cluster was vertex 61 which is an outlier whose only adjacent vertex is 34. The COMPLETE link type also extracted a variant of the (1,7,24,26,27) cluster identified by K-Medoids; vertex 1 was missing and vertices 18, 29 and 31 were added. Vertex 18 was in a number of experiments also added to the cluster by K-Medoids. This was not the case with vertices 29 and 31 which K-Medoids often placed in a separate cluster of size 5 and did not group together with this size 5 cluster.
- AVERAGE: Extracted a cluster containing 13 of the vertices from the cluster of size 21 in the two-cluster divide favoured by K-Medoids and Girvan-Newman.
- MEAN: Extracted a cluster containing 19 of the vertices from the cluster of size 21 in the two-cluster divide favoured by K-Medoids and Girvan-Newman; the missing vertices were 37 (adjacent to one vertex in each of the two clusters) and 51 (an outlier just connected to vertex 58).

- NEIGHBOR JOINING: Extracted a cluster of size 24 containing all the vertices from the cluster of size 21 in the two-cluster divide favoured by K-Medoids and Girvan-Newman together with vertices 29, 31 and 38. Note that this cluster is very similar to the one extracted by the COMPLETE link type.

CONCLUSION: Hierarchical clustering seems to have had a reasonable amount of success in identifying clusters closely related to the cluster of size 21 in the two-cluster divide favoured by K-Medoids and Girvan-Newman. In particular, the two link types COMPLETE and NEIGHBOR JOINING largely agree on a slight variant of this cluster and these two variants of this cluster are the most promising for further analysis. The family of algorithms seems less successful than the other two algorithms in detecting smaller clusters.

As with the conclusion reached in the previous section we are unable to compare our results with the results discussed in (Lusseau, et al., 2003), again they have reduced the dataset to 40 dolphins while we have operated with a dataset comprising 62 dolphins.

The following dendogram shows the clustering results with Hierarchical clustering using average link.

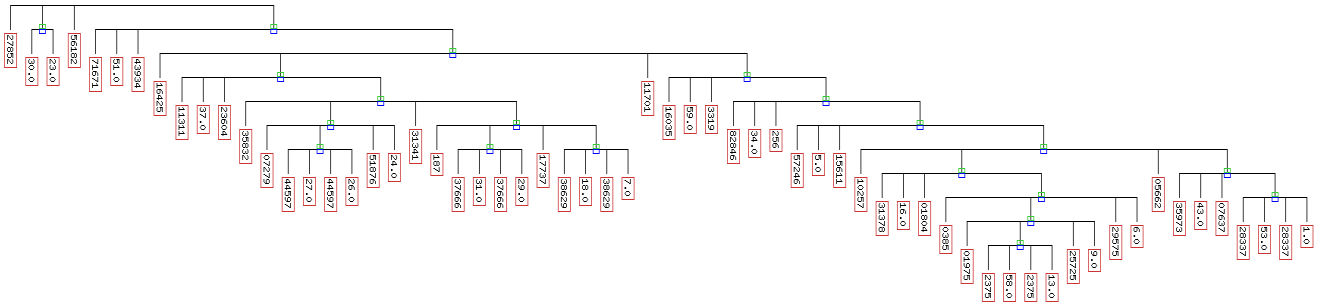


Figure 31: The Dolphin dataset hierarchical clustering using the Complete Link

7.2.9 Girvan-Newman and the Dolphins dataset

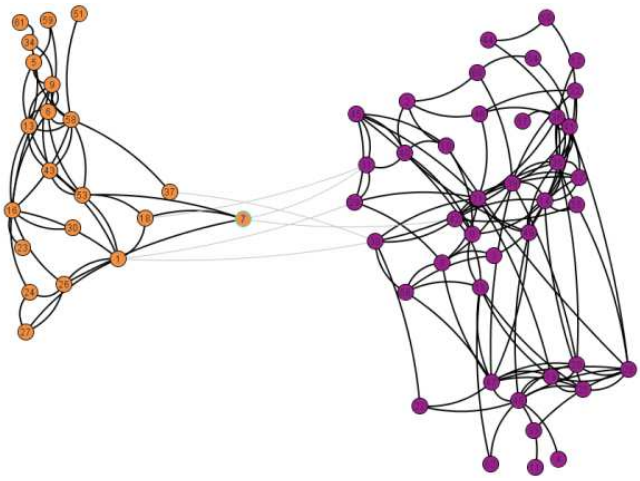


Figure 32: Girvan-Newman on the Dolphin dataset with 2 clusters

- Number of Clusters 2: Number of Edges Removed 6: The first six edges removed are the six edges connecting the two groupings of 21 and 41 vertices which appeared most frequently in the K-Medoids selections of two clusters.

- Number of Clusters 3: Number of Edges Removed 12: The 3rd cluster contains vertices 54 and 60
- Number of Clusters 4: Number of Edges Removed 21: The 4th cluster consists of a cluster identified by the K-Medoids algorithm (vertices 2,10,31,40,45) augmented with vertices 0 and 29.

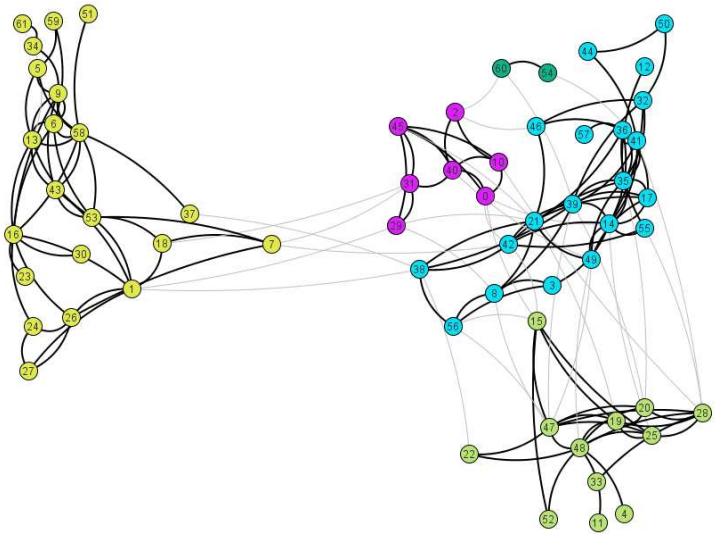
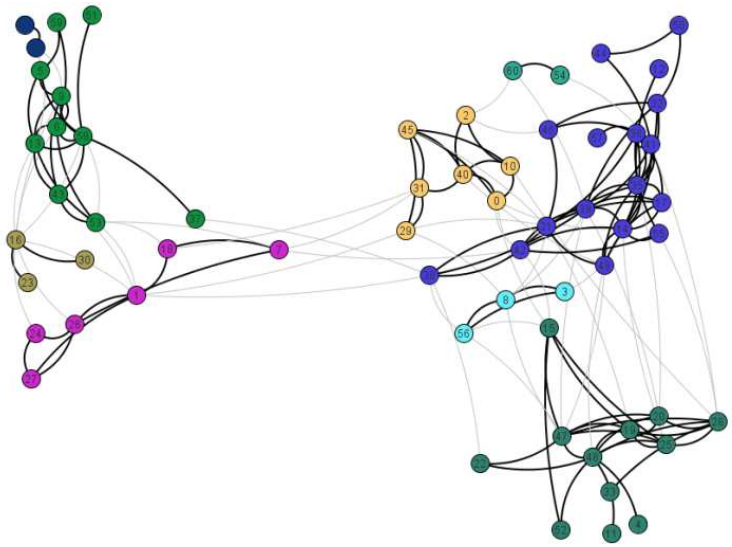


Figure 33: Girvan-Newman on the Dolphin dataset with 5 clusters

- Number of clusters 5: Number of Edges Removed 33: The five clusters are:-
  - The smaller of the first two clusters.
  - The cluster of size 5 identified by K-Medoids augmented with 2 vertices.
  - The cluster of size 2 consisting of vertices 54 and 60
  - A cluster of size 11 identified by the K-Medoids algorithm augmented with vertex 15
  - A cluster of size 20 from the earlier cluster of size 41 which was observed more ambiguously with K-Medoids.

- Number of Clusters 6: Number of Edges Removed 34: Cluster of size 2 splits from cluster of size 21.
- Number of Clusters 7: Number of Edges Removed 39: Cluster of size 3 splits from cluster of size 20
- Number of Clusters 8: Number of Edges Removed 45: Cluster of size 3 splits from cluster of size 19 formed with number of clusters 6.



- Number of Clusters 9: Number of Edges Removed 50: Cluster of size 5 identified by K-Medoids (vertices 1,7,24,26,27) augmented with vertex 18.
- Disintegration continues with IPods and a single cluster of size 2.

Figure 34: Girvan-Newman on the Dolphin dataset with 9 clusters

CONCLUSION: The clusters revealed from Girvan-Newman which would be considered for further analysis would be:-

1) The cluster of size 41 at the split into two clusters.
2) The cluster of size 21 at the split into two clusters.
3) The cluster of size 2 at the split into three clusters. (Here a dialogue could be imagined in which the characteristics of this cluster of size 2 (e.g. a mother-child pair) could be fed back into the data to identify the correlation between these characteristics and the clustering in an effort to prevent these characteristics from veiling other clustering phenomena. For example, it might be instructive to reconstruct the relationship graph by merging mother-child vertex pairs into a single vertex. The point here is that dialog between the clustering experts and the application experts, in this case, those who swim with dolphins, could lead to refinements of the network data and better evaluation of the competing clustering techniques.
4) The cluster of size 7 at the split into four clusters. (If the K-Medoids algorithm has also been run, the cluster of size 5 contained in this cluster could be reported as an alternative, also to help evaluate the two algorithms with this dataset.)
5) The cluster of size 12 at the split into five clusters. Again, it is relevant to compare this cluster with the slightly smaller cluster identified by the K-Medoids algorithm.
6) Likewise, the cluster of size 6 at the split into nine clusters. As in the previous two cases, there is a comparison with a slightly smaller cluster from K-Medoids.
7) The three cases above illustrate a weakness of the divisive nature of the Girvan-Newman and other edge-removing algorithms. The three clusters are all slightly larger than similar clusters identified by K-Medoids. It is possible that later in the edge-removing process they would be identical with the K-Medoids clusters or more similar. There is no obvious way to decide where the significant splits occur in the edge-removing process. In their original paper, Girvan and Newman do not

address this issue; they addressed it in a subsequent paper with the concept of modularity. The temptation is to regard the process as analogous to glacial breakdown; it is most interesting when big lumps fall off. An alternative criteria could be to use robustness through further divisions as a measure of significance; the longer a cluster remains intact, the more valid the grouping it represents.

7.2.10 K-Medoids and the Enron dataset

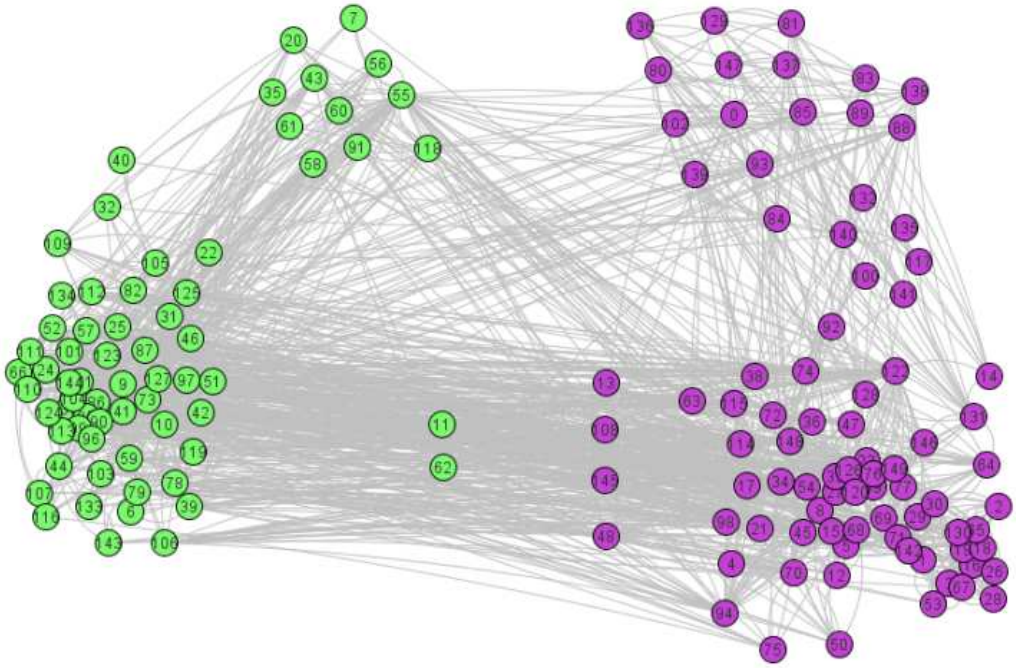


Figure 35: K-Medoids clustering of the Enron dataset

The diagram (Figure 35) above indicates in terms of colour and geometry the basic clusterings of the Enron dataset as revealed by the K-Medoids algorithm.

The most basic division is indicated by colour as well as by a vertical divide down the middle of the diagram. For experiments with 100 seeds, this basic division is reflected in 3 results with K=2, 4 with K=3, 5 with K=4, 4 with K=5, 4 with K=6, 2 with K=7 and 3 with K=8. The division continues to be reflected in results for values of K larger than 8. By “reflected in”, we mean that slightly reduced versions of these clusters appear when K is greater than two. Many experiments result in one dominant cluster and a group of singletons or occasionally, a cluster of size two.

Apart from this one dominant cluster and one exception which we describe next, we have not experienced any other clusters which take vertices from both sides of the divide as depicted above.

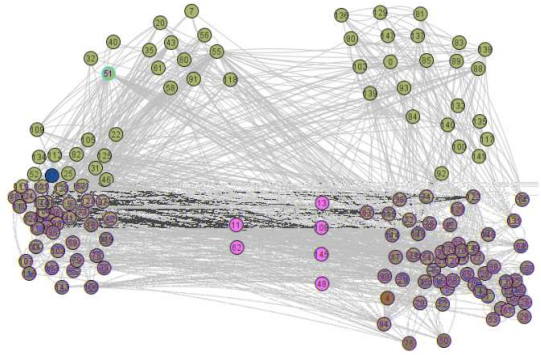


Figure 36: K-Medoids finds a cluster of 6 on the Enron dataset

The exception referred to is depicted in Figure 36 by two vertical lines of vertices towards the middle of the diagram, vertices 11 and 62 towards the left and vertices 13, 48, 108 and 145 towards the right. These six vertices seem to form a cluster, where vertex 62 is sometimes included and sometimes excluded. Whether or not vertex 62 is present, the cluster crosses the divide with the inclusion of vertex 11. For experiments with 100 seeds, the two variants of this cluster appear three times with K=4, twice with K=5 and K=6, three times with K=7 and once with K=8. The cluster appears six times without vertex 62 and five times with in the experiments enumerated. It continues to appear for values of K greater than 8.

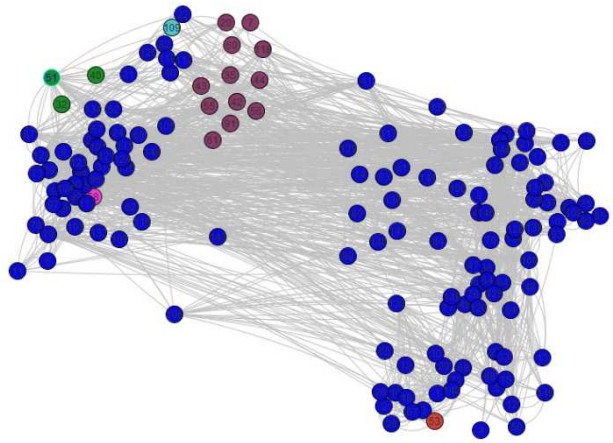


Figure 37: K-Medoids finds a cluster of 13 on the Enron dataset

The third cluster identified by the K-Medoids algorithm is a group of thirteen vertices (7,20,35,42,43,44,55,56,58,60,61,91,118) to the left of the divide towards the top of the diagram and towards the middle (Figure 37). This cluster also appears in variants where one of the vertices is missing but not always the same one. For experiments with 100 seeds, the cluster appears once with K=3, twice with K=4 and K=5, three times with K=6 and K=7, four times with K=8 and continues to appear with values of K greater than 8. For the experiments enumerated, the cluster appears 8 (Luce , 1950) The last and smallest cluster identified by the K-Medoids algorithm consists of the three vertices 32, 40 and 51 and is depicted just to the left of the cluster described in the previous paragraph. It was first identified in our experiments for K=5. It is identified for six seed values when K=5, for three when K=6 and once each for K=7 and K=8. It also continues to be identified for values of K greater than 8.

CONCLUSION: The two clusters most clearly visible in the diagram above, the cluster of size 5 or 6 containing vertex 11, the cluster of size 12 or 13 containing vertex 7 and the cluster consisting of vertices 32, 40 and 51 are all considered candidates for further analysis.

Our intuition about the Enron data before we began to apply algorithms to it and the results of our experiments suggest that there is no “correct value” of K when applying the K-Medoids algorithm to this dataset. It is something of a paradox, therefore, that the algorithm points to a clear division of the vertices into two groups, albeit with experiments from a number of differing values of K.

We conclude this section with a selection of figures depicting possible clusters (Figure 38).

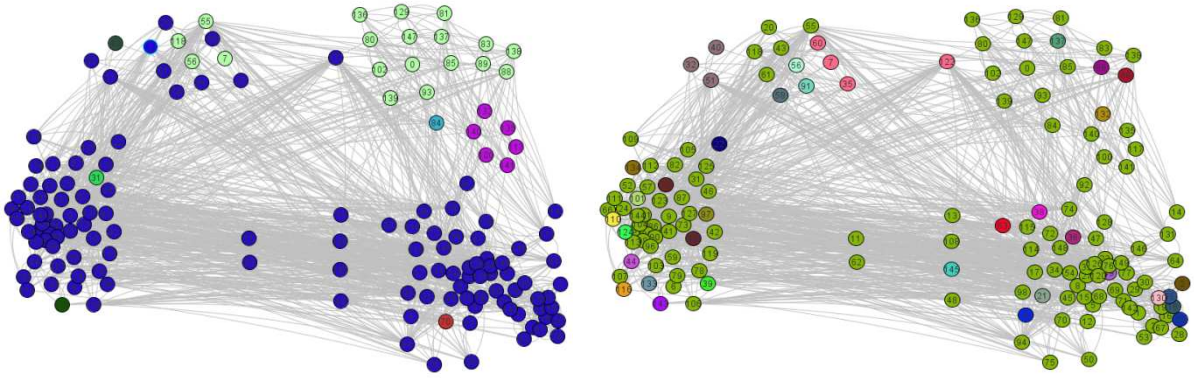


Figure 38: K-Medoids clustering on the Enron dataset

7.2.11 Hierarchical Clustering and the Enron Dataset

For Hierarchical Clustering with the Enron dataset, the three link types SINGLE, MEAN and ADJUSTED COMPLETE essentially built up one big cluster by merging vertices into it one at a time.

The three other link types included more promising clusters in the hierarchies built up but these have proved difficult to interpret with the tools we have so far been able to assimilate.

As described in "Section 7.1.2 Results of Hierarchical Clustering", the dendograms produced by the proWeb Tree Viewer are too large to be easily read and analysed. The only alternative online Newick Tree interpreter we have found in any way useful was the DrawTree parser[24] which has given us a form of overview of the Enron data but which will not allow a zooming factor whereby it is possible to see the numbers labelling the nodes and edges in the visual representation.

The results of the two parsers are superficially compatible as one would hope. We include here (Figure 39) the DrawTree diagrams for the SINGLE and COMPLETE link types, both to illustrate the homogeneous nature of the SINGLE link clustering of the Enron data and the several larger clusters that appear in the COMPLETE link clustering.

[24] Source: http://www.phylogeny.fr/version2_cgi/one_task.cgi?task_type=drawtree

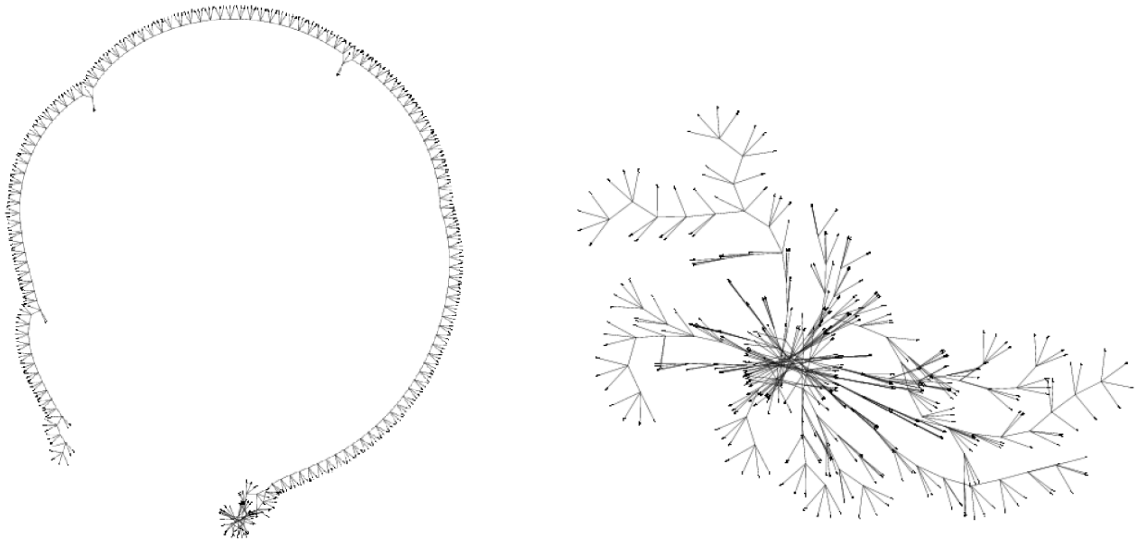


Figure 39: (a) Hierarchical single link clustering and (b) complete link clustering of the Enron dataset

More detailed analysis is needed for the three link types which produce potentially interesting clusters with the Enron data. The following possibilities are available though all of them seem time-consuming:-

1) Continue a survey of Newick string parsers and viewers extending the search from online interactive tools to those which may be downloaded and adapted.

2) Take the time to work with the proWeb Tree viewer screen by screen and attempt to isolate the significant clusters, partly by plotting (number of merges, node-height) co-ordinates onto a graph and looking for sudden changes in gradient.

3) Examine alternative representations of hierarchical clustering such as the reachability plots utilized in "J. Sander, X. Qin, Z. Lu, N. Niu, A. Kovarsky, Automatic Extraction of Clusters from Hierarchical Clustering Representations, Advances in Knowledge Discovery and Data Mining, Lecture Notes in Computer Science, Volume 2637, 2003, pp. 75-87".

It seems noteworthy that one of the two link types which superficially gives the most promising Hierarchical Clustering analysis of the Enron dataset is NEIGHBOR JOINING as was the case with the Dolphins dataset. Perhaps this link type is more widely useful than just in the area of biological classification.

7.2.12 Girvan-Newman and the Enron dataset

- Number of Clusters 2: Number of Edges Removed 9: First outlier, vertex 135.
- Number of Clusters 3: Number of Edges Removed <=79: The new cluster is composed of two clusters identified by the K-Medoids algorithm (vertices 0,137,138,102,147,89,83,80,136,81,129) and (vertices 140,141,117) augmented with vertices 14 and 131.

- Number of Clusters >= 8: Number of Edges Removed <= 176: A new cluster of size 17 is formed (1,2,3,16,18,19,26,28,29,30,53,67,69,71,130,142,(65)). All the vertices apart from 65 were contained in a cluster identified by K-Medoids.

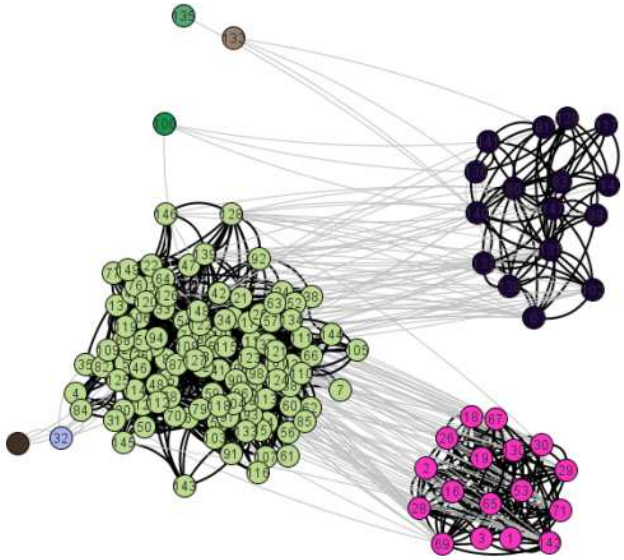


Figure 40: Girvan-Newman on the Enron dataset with 8 clusters (3 large clusters)

- Number of Clusters >> 8: Number of Edges Removed <= 914: A new cluster of size 7 consisting of 2 vertices (78,79) from the "dominant left" cluster and 4 vertices (12,34,50,70) from the "dominant right" cluster identified by the K-Medoids algorithm colour coded in Figure 35.

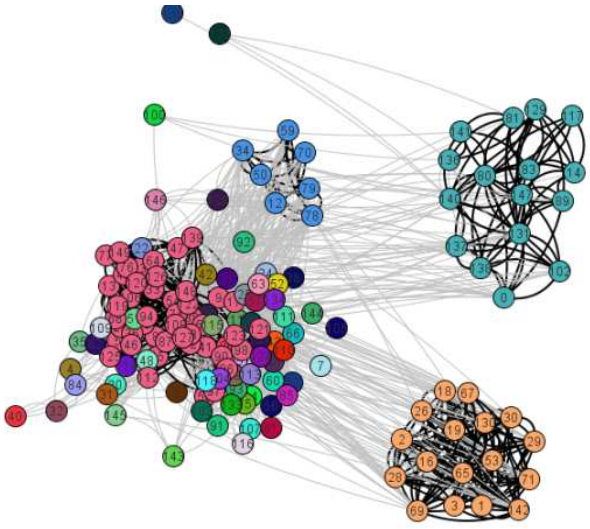


Figure 41: Girvan-Newman on the Enron dataset with more than 8 clusters (4 large clusters)

- Number of Edges Removed 1442: No new clusters of any size. Slow disintegration by extraction of single vertices.

CONCLUSION: The clusters revealed from Girvan-Newman which would be considered for further analysis would be:-

1) The cluster with vertices 0,137,138,102,147,89,83,80,136,81,129;140,141,117;14,131 identified at split into three clusters; if K-Medoids is also being used, the kernel of this cluster and the K-Medoids cluster, vertices 0,137,138,102,147,89,83,80,136,81,129 would probably be the cluster analysed.

2) The cluster with vertices 1,2,3,16,18,19,26,28,29,30,53,67,69,71,130,142,(65) identified at the split into four clusters.

8 Conclusions

8.1 Evaluation of Commuting Times Distance Metric and the Algorithms using it

The summary evaluation of the Commuting Times distance metric from our experiments using it in conjunction with K-Medoids and Hierarchical Clustering is that it is worth further investigation.

The combination of K-Medoids algorithm together with the Commuting Times distance metric seemed to identify the important clusters from the known Zachary dataset and potentially interesting clusters from within the less familiar Dolphins dataset, suggesting that the combination can at least compete with other graph clustering methods.

The stand-out success of the combination in the context of our project is, perhaps, to identify the two large clusters in the Enron dataset if, indeed, they turn out to be significant. This division was not extracted by the Girvan-Newman algorithm and we have not found any source on the Internet claiming to have found such a clustering within that dataset. Potentially, therefore, K-Medoids together with the Commuting Times distance has discovered large clusters in the Enron dataset which have previously escaped notice and that in itself would seem justification for persevering with that combination of algorithm and distance metric on further examples of graph data, both known in order better to understand the capabilities and limitations of the combination and unknown in order to give the combination the chance to show its worth alongside competing methods where the challenge is unfamiliar.

The conclusions regarding the Commuting Times metric with Hierarchical Clustering are more ambiguous. It is not much of a commendation that not one of the six link types could come close to identifying the two “administrator” and “trainer” clusters from the Zachary dataset and it would be a help in understanding when and when not to apply Hierarchical Clustering with Commuting Times if this could be explained.

On the other hand, the results from the Dolphins dataset were much more promising, and potentially significant variants of the cluster containing 21 vertices were uniquely identified by Hierarchical Clustering among the methods applied in this project.

We believe the results produced by Hierarchical Clustering and the Commute Times distance to be potentially useful as well, but here we are hampered by the lack of practical interpretive tools to process and display data on that scale.

Our modification of the JUNG GUI enables K-Medoids experiments to be executed rapidly and compared easily.

The WEKA implementation of hierarchical clustering generates the Newick strings efficiently but we lack an interpretive tool that can both give an overview of the cluster hierarchy and focus on significant smaller portions of it as required. The lack of such a tool places Hierarchical Clustering with any distance metric at a disadvantage for datasets above a certain size.

8.2 Datasets Revisited

The ultimate goal of clustering algorithms is to unveil properties of and relationships within the original data that are not available from direct observation and/or measurement or from other more immediate analysis of the data. Even if a clustering algorithm by its own evaluation criteria has performed admirably and can display groups of tightly bunched vertices separated by areas of space thinly interrupted by a few black edges, the real value of the process is in terms of the new insight it gives in the original data. Most clustering algorithms will always generate a result even on truly random data which has no clustering structure whatsoever and it would, of course, eventually be disheartening to spend time attempting to interpret clusters that were in fact artefacts of the clustering algorithm.

In many cases, the clustering expertise will be separated from the specialist knowledge and insight required to interpret the results of the clustering. This separation can indeed be an advantage in approaching the data without the prejudice which can be hard to avoid with familiarity. There will, e.g. not be many involved in graph clustering who will themselves be able to give a qualified explanation of the results in terms of a protein-protein interaction network belonging to a particular yeast identified by its Latin name!

In other cases, e.g. studies of collaborative work patterns, the ambition of the clusterers themselves interpreting the results in terms of the original data seems more feasible.

Having identified clusters in the Enron dataset, we took up the challenge of trying to determine if we could deduce what they might say about the original data. The approach was two-fold, firstly to return to the original dataset to see if the clusters could be interpreted in terms of the attributes which had been discarded when transforming the data into a graph and secondly to search the literature for others who had attempted to cluster or in other ways analyse the Enron data and to see if our clusters could be interpreted from those results.

Our basic problem with the original data was that there were no attributes (e.g. title, department) which described the position or role of an employee within the company. A clustering of the vertices representing the employees would probably need to be interpreted in terms of this sort of personal information. We have since learned that some data of this sort is available from other sources. An incomplete list of employee positions is available on the internet[25]. Another not entirely consistent list can apparently be obtained from the records of the Federal Energy Regulatory Commissions (FERCs) records from their investigation of Enron as can an incomplete list of the five regions (Canada, Central, East, Texas, West) at which Enron employees worked, but our attempts to retrieve these documents have so far been without success. Table 1 in (Diesner & Carley, 2006) contains a list of how the 151 employees in the Enron dataset may be classified according to position and observe that many of these employees seem to be in the upper echelons of the Enron Hierarchy. It seems feasible, therefore, that, with some effort, attempts could be made to interpret the results of the clustering of the Enron data in terms of position and location.

[25] Source: http://www.isi.edu/~adibi/Enron/Enron.htm

As regards literature, most of the analysis is of the contents of the e-mails rather than the people who sent and received them. We have found two exceptions to this:-

- (Diesner & Carley, 2006) have managed to locate the positions of individuals within the company which was not present in our data. Their focus was, however, time-based. They were interested in how the network of e-mails within the company changed as the crisis which eventually led to the company's demise developed and their article contains no direct clustering information.

- (CHAPANOND, KRISHNAMOORTHY, & YENER, 2005) use graph theoretic and spectral analysis techniques to cluster the data superficially as we did. They record a number of clusters generated by the different algorithms applied to the data and we have been able to compare their results with ours as they label the indices of their vertices with the names of the individuals they correspond to. There is little resemblance between the various clusters their techniques identified and those extracted by our experiments. One explanation for this may be in differences in pre-processing of the data. Their focus is an "internal" comparison of the results of the set of algorithms applied rather than an interpretation of these results in terms of the original data.

Having failed to interpret the clusters found in the Enron data, we were nonetheless spurred to attempt something similar with the dolphin data. We downloaded this data in Pajek format so there were no discarded attributes to examine but, in contrast to the Enron dataset, the data stemmed from an academic study, the results of which freely available in an article describing the study(Lusseau, et al., 2003). In addition, the word on the web was that this dataset had been used to test graph clustering algorithms almost as often as Zachary had so it should be possible to find other clustering results. The two main clusters of this dataset, or small variations of them, seemed well known but what about smaller groupings. Which were known and how did they compare with the results of our own experiments?

Unfortunately, the data on which the results of the original study were based and the dataset which seems widely available for analysis, and which we have adopted, seem incompatible. (Lusseau, et al., 2003) publishes results on a sub(pod) of 40 dolphins while the dataset available online[26] contains 62 dolphins. We have not managed to correlate our larger dolphin population with Lusseau's. Lusseau records the existence of three clusters and that the smallest of these clusters, a cluster of size 5, is a subcluster of a larger cluster with 13 members. Looking at the results we have achieved (See Figure 30) a cluster of size 5 does exist and this cluster is also associated with one of the larger clusters but we are unable to verify whether this cluster coincides with Lusseau's.

(Fortunato, 2010) discusses and illustrates a bisection of the dolphin data into two main clusters which correspond closely (a single deviation!) to the primary clustering we too have experienced. Apparently, the division into two groups was triggered by the disappearance of a single dolphin but the role in the group of the unifying dolphin and the characteristics of the division which that individual's presence managed to prevent remain unexplained, at least to us. Fortunato also mentions several internal cliques (in a non graph-theoretic sense!) within the two main groups but how closely these correspond to the clusters to be seen in Figure 28, Figure 29 and Figure 30 remains as yet unknown.

[26] The Dolphin dataset: http://vlado.fmf.uni-lj.si/pub/networks/data/mix/mixed.htm

8.2.1 Alternative Pre-Processing Strategies for the Enron Data.

In the light of the e-mail distributions illustrated in Histogram 1 on page 56 and Histogram 2 on page 57, it seemed likely that the weighted graph created from letting each edge be valued according to the number of e-mails that passed between the employees whose vertices are incident to the edge would contain important information discarded by the unweighted graph. Although we verified that the Commuting Times were different for the weighted graph, the results of our few experiments were so close to the results of the experiments with the unweighted data as to cast suspicion upon our methodology.

(CHAPANOND, KRISHNAMOORTHY, & YENER, 2005) follow the precedent of (Tyler, Wilkinson, & Huberman, 2003) in using the numbers of e-mails passing between individual members of the network to establish two thresholds, both of which have to be exceeded before an edge is added to the graph. Firstly there is a threshold for how many e-mails pass between the two members which sets a criterion for when an e-mail relationship is significant. Secondly, there is a threshold for how many e-mails each of the two individuals must have sent which obstructs the inclusion of "one-way" relationships. This is appropriate because the graph created after the two thresholds have been applied is an (unweighted) undirected graph. Our theory is derived on the basis of connected graphs so we would have to include in our pre-processing the isolation of connected components of this reduced graph. This could, for example, be achieved by allowing the JUNG GUI to display the original graph and extending the code to rewrite the connected components in Pajek format. One advantage is that the thresholds could be set high initially to verify the methodology on small graphs.

One of the social network theories which seems to be under examination regarding Enron is that a highly segmented workforce with little cross communication was a characteristic of the organization which allowed fraud to thrive. The hypothesis, if correct, is not incompatible with the existence of "natural" communication clusters. (Diesner & Carley, 2006) have observed that the traffic of e-mails peaked as the crisis broke. To investigate the speculation above, it would be helpful to isolate the e-mail traffic from a "normal" period of Enron activity. Indeed, (Diesner & Carley, 2006) have themselves compared a "crisis" month (October 2001) with a "pre-crisis" month (October 2000) though not with the purpose of testing the hypothesis above. Our proposal would be to isolate one or more periods of "normal" activity and to create and cluster the graph of e-mails belonging to such periods to see what they might reveal about a "highly segmented workforce".

One of the inappropriate practises which lead to Enron's bankruptcy was the siphoning off of losses in the finances through "special purpose entities (SPEs)". An example of such an SPE was Raptor, a liaison of Enron executives who bought shares in two companies with stock money loaned from Enron. Our proposal here is to analyse the e-mail network which may have had something to do with SPEs. This would involve creating a list of keywords which could identify content about SPEs and then to search the e-mails for these keywords. The smaller group of e-mails would then be used to create a network in the same way as the original set was. This is, of course, a single example of the idea of filtering out irrelevant e-mails to create a graph based on a particular communication theme.

Bibliography

Batagelj, V., & Mrvar, A. (Ljubljana). *Reference Manual, List of Commands with short explanation, version 2.05.* 2011.

Borgatti, S., Everett, M., & Shirley, P. (1990). LS sets, lambda sets and other cohesive subsets . *Social Networks*, 337-357.

Chan, T. F., Ciarlet, P., & Szeto, W. K. (1997). On the Optimality of the Median Cut Spectral Bisection Graph Partitioning Method. *SIAM J. Scientific Computing, 18*(3), 943-948.

Chandra, A. K., Raghavan, P., Ruzzo, L. W., Smolensky, R., & Tiwari, P. (1989). The Electrical Resistance of a Graph Captures its Commute and Cover Times. *Journal of Computational Complexity, 6*(4), 312-340.

CHAPANOND, A., KRISHNAMOORTHY, M. S., & YENER, B. (2005). Graph Theoretic and Spectral Analysis of Enron Email Data. *Computational & Mathematical Organization Theory*, 265–281.

Chen, J., & Yuan, B. (2006). Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, 2283-2290.

Clauset, A., Newman, E. J., & Moore, C. (2004). Finding community structure in very large networks. *Physical review* .

Condon, A., & Karp, R. M. (2001). Algorithms for Graph Partitioning on the Planted Partition Model. *Journal of Random Structures and Algorithms*, 116-140.

Danon, L., Díaz-Guilera, A., & Arenas. (2006). The effect of size heterogeneity on community identification in complex networks. *Journal of Statistical Mechanics: Theory and Experiment 2006.11* .

Deza, M. M., & Deza, E. (2009). *Encyclopedia of Distances.* Berlin: Springer.

Diesner, J., & Carley, K. M. (2006). Exploration of Communication Networks from the Enron email corpus. *SIAM International Conference on Data Mining: Workshop on Link Analysis, Counterterrorism and Security, .* CA.

Dijkstra, E. W. (1959). A Note on Two problems in Connexion with Graphs. *Numerische Mathematik 1*, 259-261.

Ding, C., & He, X. (2004). K-Means. *ICML '04 Proceedings of the twenty-first international conference on Machine learning*, (s. 225-232). New York.

Donath, W. E., & Hoffman, a. J. (1973). Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 420-425.

Doyle, P. G., & Snell, J. L. (2000). Random Walks and Electric Networks. *The Mathematical Association of America*.

Dunn, R., Dudbridge, F., & Sanderson, C. M. (2005). The Use of Edge-Betweenness Clustering to Investigate Biological Function in Protein. *BMC bioinformatics*.

Erdos , P., & Renyi, A. (1959). On random Graphs. *Publ. Math, 6*, 290-297.

Fiedler. (1973). Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 298-305.

Firat, A., Chatterjee, S., & Yilmaz. (2007). Genetic clustering of social networks using random walks. *Computational Statistics & Data Analysis*, 6285-6294.

Fortunato, S. (February 2010). Community detection in graphs. *Physics Reports, 486*(3-5), s. 75-174.

Fouss, F., Pirotte, A., Renders, J.-M., & Saerens, M. (2007). Random-Walk Computation of Similarities between Nodes of a Graph with Application to Collaborative Recommendation. *Knowledge and Data Engineering, IEEE Transactions*, 355-369.

Freeman, L. C. (1977). A Set or Measures of Centrality Based on Betweness. *Sociometry*, 35-41.

Girvan, M., & Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, *99 No. 12*, s. 7821–7826.

Golub, G. H., & Van Loan, C. F. (1989). *Matrix computations.* John Hokins Univ. Press.

Good, B. H., de Montjoye, Y.-A., & Clauset, A. (2009). Performance of modularity maximization in practical contexts. *Physical Review*.

Gvishiani, A. D., & Gurvich, v. A. (1987). Metic and ultrametric spaces of resistances. *COMMUNICATIONS OF THE MOSCOW MATHEMATICAL SOCIETY*(6), 187-188.

Göbel, F., & Jagers, A. (1974). Random Walks on Graphs. *Stochastic Processes and their Applications, 2*(4), 311-336.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (1. June 2009). The WEKA Data Mining Software. *ACM SIGKDD Explorations Newsletter*, s. 10-18.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimal Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on, 4*(2), 100-107.

Hopcroft, P. J. (2008, March 5). *CS 683: Advanced Design and Analysis of Algorithms: Lecture Notes 20*. Retrieved from http://www.cs.cornell.edu/courses/cs683/2008sp/lecture%20notes/lec20notes.pdf

Jin, R., Xing, K., Parkes, D. C., & Wolfe, P. (2007). Analysis of bidding networks in eBay: aggregate preference identification through community detection. *Proceedings of AAAI workshop on plan, activity and intent recognition*.

Karrer, B., Levina, E., & Newman, M. E. (2008). arXiv:0709.2108v1 [physics.data-an] 13 Sep 2007. *Karrer, Brian, Elizaveta Levina, and Mark EJ Newman. "Robustness of community structure in networks." Physical Review E 77.4 (2008): 046119.*

Kernighan, B. W., & Lin, S. (1970). An Efficeient Heuristic Procedure for Partituoning Graphs. *Bell System Tech*, 291-308.

Kernighan, B. W., & Lin, S. (1970). An eflicient heuristic procedure for partitioning graphs. *Bell system technical journal*.

Klein, D. J., & Randic, M. (1993). Resistance Distance. *Journal of Mathematical Chemistry, 12*, 81-95.

Klimt, B., & Yang, Y. (2004). Introducing the Enron Corpus. *First conference on email and anti-spam.* Pittsburgh PA15213: CEAS.

Luce , R. D. (1950). Connectivity and generalized cliques in sociometric group structure. *Psychometrika 15*, (s. 160-190).

Lusseau, D., Schneider, K., Boisseau, O. J., Haase, P., Slooten, E., & Dawsen, S. M. (2003). The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiologyh, 54*(4), 396-405.

Luxburg, U., Radl, A., & Hein, M. (2011). Hitting and commute times in large graphs are often misleading.

Newman, M. E. (2004a). Analysis of weighted networ. *Physics Review E*.

Newman, M. E. (2006a). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, (s. 8577-8582).

Newman, M. E. (2006b). Finding community structure in networks using the eigenvectors of matrices. *Physical review*.

Newman, M. E., & Girvan, M. (2004). arXiv:cond-mat/0308217v1 [cond-mat.stat-mech] 11 Aug 2003. *Phys. Rev*.

Park, H. S., Lee, J. S., & Jun, C. H. (2006). A K-means like algorithm for K-medoids clustering and its performance. *ICCIE.*

Porter, M. A., Mucha, P. J., Newman, M. E., & Friend, A. J. (2007). Community structure in the united states house of representatives. *Physica A: Statistical Mechanics and its Applications*, 414-438.

Porter, M. A., Mucha, P. J., Newman, M. E., & Warmbrand, C. M. (2005). A network analysis of committees in the US House of Representatives. *Proceedings of the National Academy of Sciences of the United States of America*, (s. 7057-7062).

Radicchi, F., Castellano, C., Cecconi, F., Loreto, V., & Parisi, D. (2004). Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, (s. 2658-2663).

Rattigan, M. J., Maier, M., & Jensen, D. (2007). Graph clustering with network structure indices. *Proceedings of the 24th international conference on Machine learning* , (s. 783-790). New York.

Rives, A. W., & Galitski, T. (2003). Modular organization of cellular networks. *Proceedings of the National Academy of Sciences*, (s. 1128-1133).

Saito, N. (7. May 2012). *MAT 280: Harmonic Analysis on Graphs & Networks.* Retrieved from www.math.ucdavis.edu: https://www.math.ucdavis.edu/~saito/courses/HarmGraph/lecture11.pdf

Saito, N., & Nei, M. (1987). The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol. Biol. Evol, 4*(4), 406-425.

Sander, J., Quin, X., Lu, Z., Niu, N., & Korvarsky, A. (2003). utomatic Extraction of Clusters from Hierarchical Clustering Representations. *Advances in knowledge discovery and data mining* (s. 75-87). Springer-Verlag Berlin.

Sen, T. Z., Kloczkowski, A., & Jernigan, R. J. (2006). Functional clustering of yeast proteins from the protein-protein interaction network. *BMC bioinformatics*.

Smola, A. J., & Kondor, R. (2003). Kernels and Regularization on Graphs. *16th Annual Conference on Learning Theory and 7th Kernel Workshop, COLT/Kernel 2003*, (s. 144-158). Washington DC.

Traud, A. L., Kelsic, E. D., Mucha, P. J., & Porter, M. A. (2008). Community structure in online collegiate social networks. *Organization* .

Tyler, J. R., Wilkinson, D. M., & Huberman, B. A. (2003). Automated discovery of community. *Communities an Technologies*, 81-96.

Wilkinson, D. M., & Huberman, B. A. (2004). A method for finding communities of related genes. *Proceedings of the National Academy of Sciences of the United States of America*, (s. 5241-5248).

Wilson, R. J. (1972). *Introduction to Graph Theory* (1st Edition udg.). Longman.

Yuta , K. N., Ono, N., & Fujiwara, Y. (2007). A gap in the community-size distribution of a large-scale social networking site. *arXiv preprint physics/0701168*.

Zachary, W. W. (1977). An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research, 33*, 452-473.

Zha, H., Ding, C., Gu, M., He, X., & Simon, H. D. (2011). Spectral Relaxation for K-Means Clustering. *Neuural Information Processing Systems*, (s. 1057-1064). Vancouver.

Zhang, W., Zhao, D., & Wang, X. (24. April 2013). *Agglomerative clustering via maximum incremental path integral.* Retrieved from www.sciencedirect.com: http://www.sciencedirect.com/science/article/pii/S0031320313001830

Appendix A: Working with the Euclidean Commuting Time Distance and the Measures Presented by The Moore-Penrose Pseudoinverse being a Gram Matrix.

We have observed that with the Commuting Times distance metric, there is no way to calculate a centroid (See Section "3.5.1 K-Means") which prohibits it being used with the K-Means clustering algorithm and certain Hierarchical Clustering link types (See "Section 3.5.3 Hierarchical Clustering"). (Fouss, Pirotte, Renders, & Saerens, 2007) explain how to overcome this obstacle by defining distance metrics in terms of the eigenvectors of matrices derived from and/or associated with the Commuting Times distance metric. We find these extensions of the distance metric and their compatibility with a wider range of algorithms interesting so we give a description of them here.

First, we present a brief but necessary extension to our linear algebra summary in "Section 3.3 Linear Algebra". As a symmetric, square, real matrix, a Laplacian matrix L satisfies $L = L^T$. The **nullspace** or **kernel** N(A) of a matrix A is the set of all vectors x for which Ax = 0. **EP matrices** are matrices for which $N(A) = N(A^T)$ and are a generalization of normal matrices described above. Laplacian matrices are trivially EP matrices. For EP matrices, the eigenvectors of the Moore-Penrose pseudoinverse are identical to the eigenvectors of the original matrix. In addition, the eigenvalues corresponding to a particular common eigenvector are closely related, either zero for both matrices or reciprocals if non-zero. It follows that the Moore-Penrose pseudoinverse of a Laplacian matrix is also **positive-semidefinite** (All of the eigenvalues are non-negative).

With the extra definitions in place, we reproduce here some of the mathematics in Appendix D of (Fouss, Pirotte, Renders, & Saerens, 2007).

Every positive-semidefinite matrix can be transformed to a diagonal matrix, so we can write $D = U^T L^+ U$, where U is an orthonormal matrix comprising the eigenvectors of L^+.

For $1 <= i <= n$, let e_i be the unit vector corresponding to the i^{th} column of the identity matrix I_n, i.e. e_i consists of zeros apart from in the i^{th} position where it contains a "1". The formula above can thus be written as a matrix product as

$$C(v_i,v_j) = 2m(e_i - e_j)^T L^+ (e_i - e_j)$$

By using the transformation

$$e_i = Ux_i, \quad y_i = D_i^{1/2} x_i$$

we obtain

$$\begin{aligned} C(v_i,v_j) &= 2m(e_i - e_j)^T L^+ (e_i - e_j) \\ &= 2m(x_i - x_j)^T U^T L^+ U(x_i - x_j) \\ &= 2m(x_i - x_j)^T D(x_i - x_j) \\ &= 2m(x_i - x_j)^T (D^{1/2})^T D^{1/2}(x_i - x_j) \end{aligned}$$

$$= \quad 2m(y_i - y_j)^T(y_i - y_j)$$

$$= \quad 2m\,||y_i - y_j)||^2$$

In this way, the n-dimensional "node" vectors $y_1, y_2,\ldots,y_n$ form an n-dimensional Euclidean space in which the distance between them is proportional to $(C(v_i,v_j))^{½}$, the square root of the commuting time between the corresponding nodes.

It follows that not only is C(u,v) a distance metric on a connected graph but that $(C(u,v))^{½}$ is a second such distance metric. To distinguish between the two, $(C(u,v))^{½}$ will be referred to as the **Euclidean Commuting Time Distance**.

We now have two commuting distance metrics defined on a connected graph but we can also show that the Moore-Penrose pseudoinverse L^+ of the Laplacian matrix of the graph is a matrix of inner products of the "node" vectors $y_1, y_2,\ldots,y_n$, so this matrix can itself be regarded as a matrix of similarity measures between the nodes of the graph:-

$$y_i^T y_j = (D_i^{½}x_i)^T D_j^{½}x_j$$

$$= \quad e_i^T U D U^T e_j$$

$$= \quad e_i^T L^+ e_j$$

$$= \quad l_{ij}^+$$

This property can be expressed by saying that L^+ is the Gram (kernel) matrix of the set of vectors $y_1, y_2,\ldots,y_n$ or that it may be regarded as a kernel on the graph itself (or more specifically, the vertex set of the graph). Having a kernel opens up the possibility of using kernel-based algorithms to attempt to capture the structure of the graph. In (Smola & Kondor, 2003), the authors introduce a family of kernels on graphs in order to extend techniques normally applied to real-valued functions to graphs.

A kernel-based algorithm which thus becomes available is **Principal Component Analysis**. This is a mathematical procedure which "uses an orthogonal transformation to a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables" called **principal components**. The number of principal components is less than or equal to the number of original variables. This transformation is defined in such a way that the first principal component has the largest possible variance (accounts for as much of the variability in the data as possible) and each component in turn has the highest variance possible under the constraint that it be orthogonal to the preceding components[27]. One of the techniques used to find the sequence of principal components is to arrange the eigenvalues of a Gram matrix in order and to select components (eigenvectors) associated with the smallest or largest eigenvalues.

In (Fouss, Pirotte, Renders, & Saerens, 2007) , the authors approximate the Moore-Penrose pseudoinverse L^+ of the Laplacian matrix of the graph by a matrix M^+ in which the smallest n - m eigenvalues of L^+ together with the corresponding eigenvalues are essentially discarded. This leaves an m-dimensional space which approximately preserves the Euclidean Commuting Distance Times from the original n-dimensional space.

[27] Source: https://en.wikipedia.org/wiki/Principal_component_analysis

The quality of the approximation can be assessed by a bound equal to the sum of the discarded eigenvalues multiplied by twice the number of edges in G. The expectation is that the first few principal components contain most of the information about the basic structure of the graph and that the remaining components related to the smaller eigenvalues represent "noise".

The **algebraic connectivity** of a graph G is the second smallest eigenvalue of the Laplacian matrix of G. A consequence of the previous discussion of Laplacian matrices is that this eigenvalue is positive if and only if G is connected. The eigenvector corresponding to this eigenvalue is called the **Fiedler vector** of the graph in honour of Miroslav Fiedler who first developed the ideas behind algebraic connectivity. The Fiedler vector has been used to partition graphs. The simplest use is to partition the graph according to negative and positive entries in the vector, the negative entries typically corresponding to poorly connected vertices. A slightly more sophisticated portioning would also separate vertices corresponding to small positive entries from those with positive entries above an appropriate threshold.

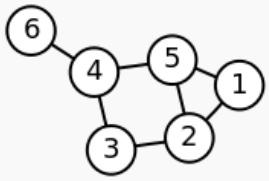
Labeled graph	Degree matrix	Adjacency matrix	Laplacian matrix
	$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$	$\begin{pmatrix} 2 & -1 & 0 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ 0 & 0 & -1 & 3 & -1 & -1 \\ -1 & -1 & 0 & -1 & 3 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix}$

Figure 42: The Fiedler vector of the graph above is (0.415, 0.309, 0.069, -0.221, 0.221, -0.794) [28]

The Fiedler vector also plays a part in more complex graph partitioning strategies, e.g. in the strategy described in (Chan, Ciarlet, & Szeto, 1997).

There is a connection between the Fiedler vector and the Principal Component Analysis technique adopted in (Fouss, Pirotte, Renders, & Saerens, 2007). The Fiedler vector contains the projections of the node vectors on the first principal component of the graph generated by the authors' PCA technique.

Before leaving this somewhat extended discussion of Commuting Time Distance, we would like to note the connection between the Principal Component Analysis algorithms which the kernel property of the Moore-Penrose pseudoinverse of the Laplacian matrix makes available and the K-means clustering method. In (Zha, Ding, Gu, He, & Simon, 2011), the authors show that the minimization of the sum-of-squares cost function which is a part of the K-Means clustering algorithm can be reformulated as a trace maximization problem associated with the Gram matrix of the data vectors. Two of the authors were subsequently able to prove that the principal components generated by Principal Component Analysis are the continuous solutions to the discrete cluster membership indicators for K-Means clustering, (Ding & He, 2004).

In this section, we have introduced two distance measures, the Commuting Distance and the Euclidean Commuting Distance and have described how the Moore-Penrose pseudoinverse L^+ of the Laplacian matrix provides directly a similarity measure. A second similarity measure of interest is the cosine product of the node vectors defined by

$\cos^+(i,j) = l_{ij}^+/(l_{ii}^+ l_{jj}^+)^{1/2}$

[28] Source: http://en.wikipedia.org/wiki/Algebraic_connectivity

So we now have two distance measures and two similarity measures which can be applied to graph data. It is well known that inner-product based measures tend to outperform Euclidean distances when computing proximities between documents using the vector-space model of information retrieval. The relative performance of the four measures above in various fields of application of graph data, e.g. collaborative recommendation, is as yet largely unknown.

In addition to the four methods above, it seems that the kernel properties of L^+ give access to a wider range of measures and techniques which may prove useful and effective in graph clustering.

As a final comment to this discussion we refer the interested reader to (Saito, 2012). This covers much of the material above and a little besides concisely and clearly.

Appendix B: User manual.

In order to run the graph clustering visualization application double click on the GraphClustering jar that can be found in the dist library, alternatively create a short cut to the jar file and run the application from the shortcut.

Figure 43: Open Pajek .net file dialog box

Once the application starts a small 'Open File' dialog box appears. This dialog box enables a graph stored in the Pajek .net format to be opened and loaded into the application. (the graphs used for this project can be found in the source code that accompanies this project or on the Google code project site[29]).

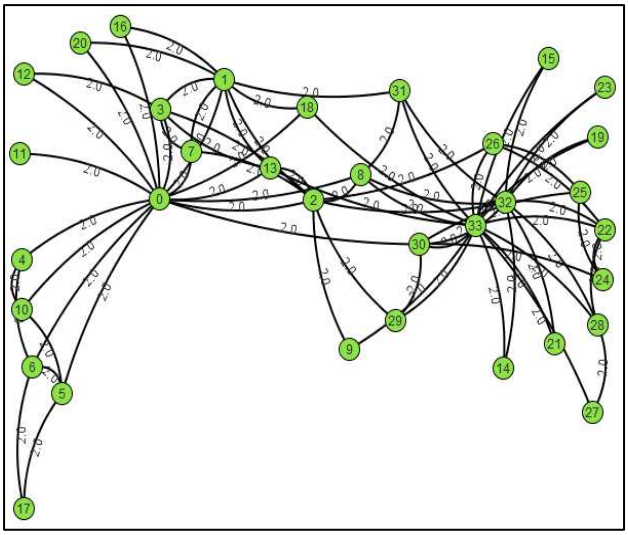


Figure 44: Initial rendering of a loaded graph

Once the chosen graph has been loaded the graph is rendered within the Gui. Each vertex is represented by a bubble, assigned a random colour, and the vertex index is shown with in the bubble.

Each edge is depicted as a line connecting two vertices and if the graph is a weighted graph the weights are displayed adjacent to the vertex (Figure 44).

Once the graph has been loaded the commute time matrix is calculated and the graph is ready to be clustered by using either the Girvan-Newman algorithm or the K-Medoids algorithm.

At the bottom of the application window is the control panel. The control panel is equipped with 3 function buttons, two sliders and two dropdown boxes.

[29] Source code repository: http://code.google.com/p/graph-clustering-ra/

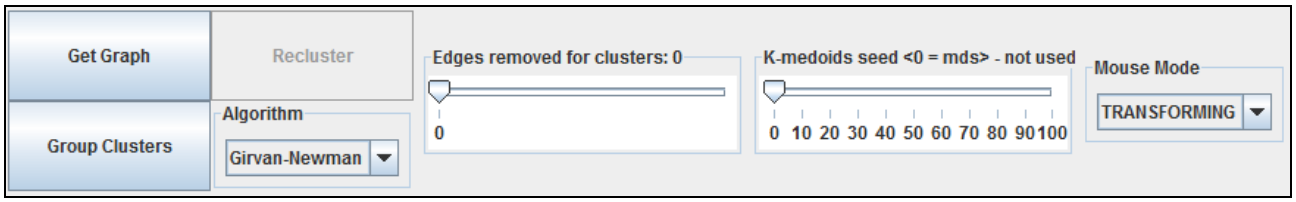


Figure 45: Graph clustering GUI control panel

- The 'Get Graph' button shows the 'Open File' dialog box, which will enable a new graph to be selected and opened.
- The 'Recluster' button will recluster a graph with a new random seed when the seed value is 100 and the K-Medoids algorithm is chosen.
- The 'Group Clusters' button toggles between a view that depicts the graph, with coloured vertices that correspond to the cluster they belong to, and a view that depicts, as groups, the clusters that have been found by the chosen algorithm, as depicted in figures .

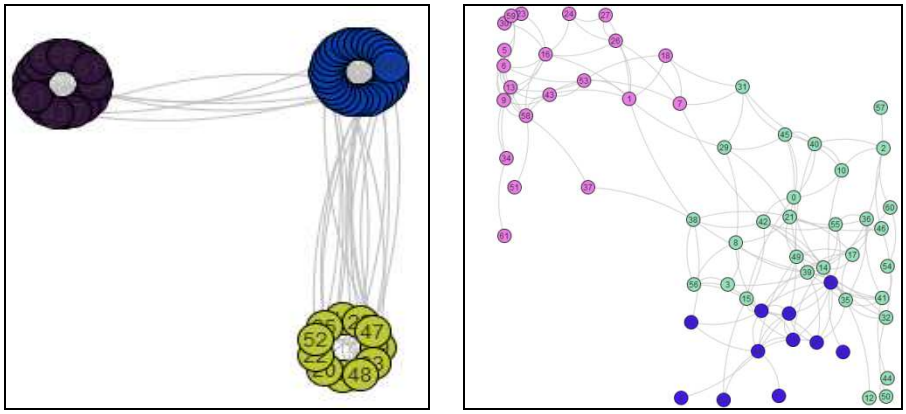


Figure 46: Grouping of clusters and how the clusters appear when not grouped

- The 'Algorithm' dropdown box shows which clustering algorithm has been chosen. The choice of algorithm also influences the functionality associated with its adjacent slider. The Girvan-Newman algorithm expects a number of edges to be removed while the K-Medoids algorithm expects to know how many clusters are expected/required.
- The 'Number of clusters required' / 'Edges removed for clusters' slider is redrawn each time a graph is loaded or the algorithm is changed. The number of possible edges that can be removed is calculated by counting the number of edges within the graph, while the number of clusters is calculated by counting the number of vertices within the graph.
- The 'K-Medoids seed' slider allows the definition of a seed between 1 and 100, a 0 value indicates that no seed value is to be used and that the minimum distance sums initialization of the K-Medoids algorithm is to be used instead. If the seed value is **100** then a random seed is computed before the algorithm is run.
  The slider has no influence on the Girvan-Newman algorithm.

- The 'Mouse mode' dropdown changes the mouse functionality. The 'transforming' option enables the whole graph to be repositioned within the Gui while the 'Picking' option allows a single or a group of vertices to be re-positioned.

The ability to reposition individual or groups of vertices is especially useful when comparing clustering results. It is advantageous to reposition the vertices after each clustering run so that the results of using differing clustering parameters for both algorithms can be compared.

The three clustering results depicted below show the results for (a) the Girvan-Newman algorithm, (b) The K-Medoids algorithm using minimum distance sum initialization with 4 clusters and (c) the K-Medoids algorithm using a seed of 2.

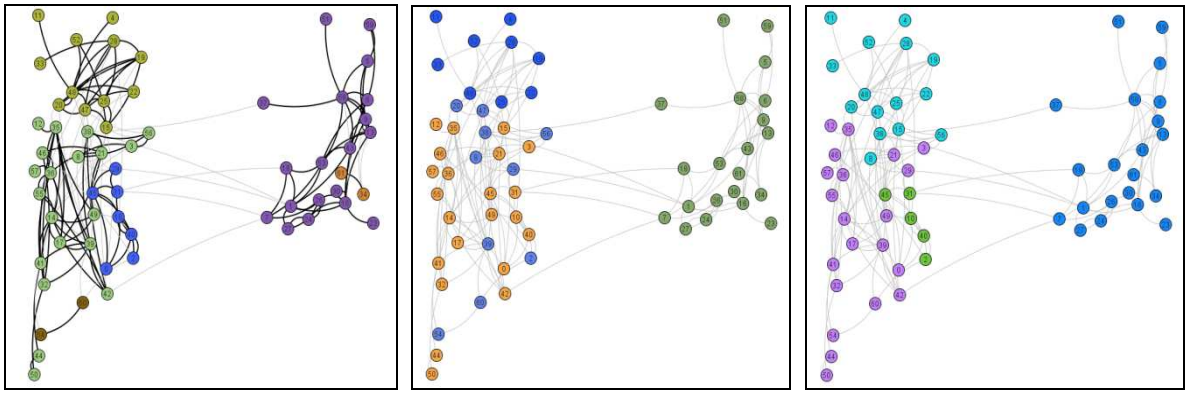


Figure 47: (a) Girvan-Newman clusters (b) K-medoids with MDS (c) K-medoids with a seed of 2

An alternative way of using the application is to run the two algorithms with varying parameters using the 'Group cluster' mode and when interesting clusters appear switch from 'Group mode' to 'Expanded' mode in order to see the where the clusters appear within the graph.

Use of the two modes is depicted below (a) a graph clustered and shown in 'Group' mode and (b) the same graph shown in 'Expanded' mode.

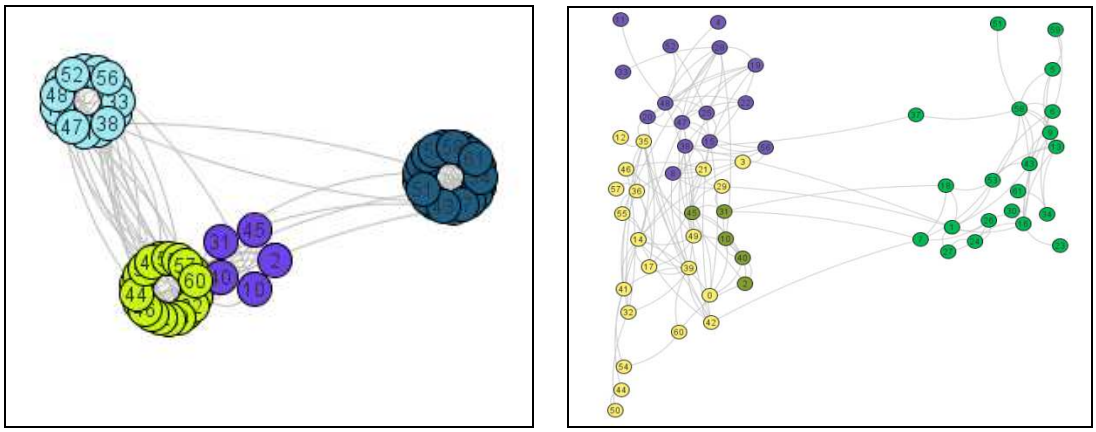


Figure 48: (a) a graph clustered and shown in 'Group' mode (b) the same graph shown in 'Expanded' mode.

Toggling the ‘Group Clusters’ button will also re-render the graph with a new selection of random colours. This functionality is especially helpful if the resulting random colouring of the individual vertices is not pleasing to the eye.