

Final Year Project Report
School of Computer Science
The University of Manchester

A Statistics-Based Algorithm for Positional Analysis of Networks

William Acton

Degree Programme: B.Sc. Artificial Intelligence (w/ Industrial Experience)
Supervisor: Jonathan Shapiro
Date: May 2010

Abstract

...

Project Title: A Statistics-Based Algorithm for Positional Analysis of Networks
Author: William Acton
Date: 5th May 2010
Supervisor: Jonathan Shapiro

Acknowledgements

I would like to thank my supervisor, Jon Shapiro, for his assistance, ideas, and insight.

I would also like to thank Victoria, Katie, Natalie, Karl, Mark, and my family for their patience and support.

Contents

Abstract.....	ii
Acknowledgements.....	iii
Contents.....	iv
List of Figures	vii
List of Tables	vii
1. Introduction	1
1.1. Problem Background.....	1
1.2. Project Proposal	2
1.3. Existing Work	2
1.4. Objectives.....	3
1.5. Report Overview	3
2. Background	4
2.1. Notions of Position.....	4
2.1.1. Actor Similarity vs. Actor Cohesion	4
2.1.2. Positional Analysis.....	4
2.2. Technical Definitions and Notation.....	5
2.3. Graph Theory	6
2.3.1. Mapping: Isomorphism	6
2.3.2. Mapping: Automorphism	7
2.3.3. Centrality: Degree	8
2.3.4. Centrality: Betweenness	9
2.4. Equivalence Definitions.....	11
2.4.1. Position as Structural Equivalence	11
2.4.2. Position as Automorphic Equivalence.....	12
2.4.3. Position as Regular Equivalence.....	13
2.4.4. Selected Equivalence & Justification.....	14
2.5. Existing Algorithm	14
2.5.1. Additional Mathematical Definitions	15
2.5.2. Finding Orbits.....	15
2.5.3. Measures of the Extent of Equivalence	16
2.5.4. Pseudocode.....	17

2.5.5.	Complexity	18
3.	New Algorithm	19
3.1.	k -Means Clustering Description	19
3.2.	Cluster Means	19
3.3.	Pseudocode.....	20
3.4.	Complexity	21
3.5.	Testing.....	21
3.5.1.	k -Means Clustering Stabilisation	21
3.5.2.	Clustering Tests	22
3.5.3.	Measures of Similarity: Orbit-Cluster Equivalence	22
3.5.4.	Measures of Similarity: Point-Pair Equivalence	23
4.	Implementation	25
4.1.	Code Language	25
4.2.	File Format	25
4.3.	Important & Difficult Aspects	26
4.3.1.	Neighbourhood Generation.....	26
4.3.2.	Floyd-Warshall Algorithm	27
4.3.3.	Calculating Cluster Means (Secondary Clustering)	29
5.	Results.....	32
5.1.	Required Interaction	32
5.2.	Interpretation of Output.....	33
5.3.	Performance	35
6.	Conclusions	36
6.1.	Achievement of Objectives	36
6.2.	Changes to Original Plan	36
6.3.	Further Activity	36
	References	38
	Appendix A: Data Sets.....	41
	A – 1: Les Misérables (Knuth, 1993).....	41
	A – 2: Zachary’s Karate Club (Zachary, 1977).....	42
	A – 3: Dolphin Social Network (Lusseau, Schneider, Boisseau, Haase, Slooten, & Dawson, 2003)..	43
	Appendix B: Stabilisation Charts	44
	B – 1: “Les Misérables” Degree Stabilisation	44
	B – 2: “Les Misérables” Neighbour Type Stabilisation	45

B – 3: “Karate Club” Degree Stabilisation	46
B – 4: “Karate Club” Neighbour Type Stabilisation	47
B – 5: “Dolphins” Degree Stabilisation.....	48
B – 6: “Dolphins” Neighbour Type Stabilisation	49
Appendix C: Clustering Test Data.....	50
C – 1: Test Graph 1 (single node, isolated)	50
C – 2: Test Graph 2 (single node, reflexive loop)	50
C – 3: Test Graph 3 (two nodes, connecting edge)	50
C – 4: Test Graph 4 (two nodes, no edge).....	51
C – 5: Test Graph 5 (two nodes, two identical edges)	51
C – 6: Test Graph 6 (three nodes, single orbit)	51
C – 7: Test Graph 7 (three nodes, two orbits).....	52
C – 8: Test Graph 8 (three nodes, one isolated)	52
C – 9: Test Graph 9 (three nodes, reflexive loop)	53
C – 10: Test Graph 10 (four nodes, single orbit)	53
C – 11: Test Graph 11 (four nodes, two orbits, high connectivity)	53
C – 12: Test Graph 12 (four nodes, two orbits, low connectivity)	54
C – 13: Test Graph 13 (four nodes, one isolated, three orbits)	54
C – 14: Test Graph 14 (four nodes, two communities, single orbit)	55
C – 15: Test Graph 15 (four nodes, two isolated, two orbits).....	55
C – 16: Test Graph 16 (nine nodes, one isolated, eight orbits).....	56
C – 17: Test Graph 17 (ten nodes, ten singleton orbits).....	57
C – 18: Test Graph 18 (ten nodes, same degree, three orbits).....	57
C – 19: Test Graph 19 (ten nodes, five orbits; assignment change in secondary clustering)	58
C – 20: Test Graph 20 (empty graph).....	58
Appendix D: Input File Formats	59
Appendix E: Project Plans	60
E – 1: Original Plan	60
E – 2: Revised Plan	61

List of Figures

Figure 1: Graph with 3 vertices and 3 edges.....	5
Figure 2: Graphs G and H are isomorphic.	6
Figure 3: Graph with four automorphisms.	7
Figure 4: Graph with 6 nodes, 5 edges, and 2 orbits.	8
Figure 5: Subgraph of graph in Figure 4.	9
Figure 6: The six geodesics of the graph in Figure 5.	9
Figure 7: Graph where some node-pairs have two geodesics.	10
Figure 8: Graph in Figure 4 coloured according to structural equivalence.	12
Figure 9: Graph in Figure 4 coloured according to automorphic equivalence.	13
Figure 10: Extension of graph in Figure 4 coloured according to regular equivalence.	13
Figure 11: Point-deleted neighbourhoods of graph in Figure 4.	15
Figure 12: Graph with 5 nodes and 4 edges, coloured to highlight its orbits.	15
Figure 13: JGraphT Floyd-Warshall algorithm documentation.....	28
Figure 14: JGraphT k -shortest paths algorithm documentation.	28
Figure 15: Run configurations of the project.	32
Figure 16: Console display of a completed program run.	33
Figure 17: Files created by a run of the program.....	33
Figure 18: Example program results file.	34

List of Tables

Table 1: The graph isomorphism $\pi_1: G \rightarrow H$ relating the graphs in Figure 2.	7
Table 2: All automorphisms of the graph in Figure 3.....	8
Table 3: Matrix of shortest paths of graph in Figure 7.	10
Table 4: All automorphisms of the graph in Figure 4.....	12
Table 5: Degree vectors of neighbourhoods of the graph in Figure 12.....	16
Table 6: Everett and Borgatti (1988) algorithm applied to graph in Figure 12.	17
Table 7: Results of clustering based on neighbour type of graph in Figure 9.	20
Table 8: Example of point-pair tabulation.	24

1. Introduction

1.1. Problem Background

Locating elements of a graph that appear to behave similarly – or *actors* that occupy the same *position* or *role* – is a problem that has been of interest to many for a long time now, especially – but not limited to – those in the field of social sciences. Graph theory is often used as an intuitive model, where the vertices of a graph represent individuals and the edges between the vertices represent particular relationships between the individuals. With just this simple model, a complex web of ties can develop very quickly. Analysing the structure of this resulting network can give a great deal of insight into the properties and the behaviour of the graph elements – much more than mere visual inspection can.

Structural analysis can assign metrics to the vertices of the graph, measures that can be used to help compare and contrast the elements and to form judgements about them. Such a judgement is whether or not two elements are similar, or even equivalent. The knowledge of the presence or absence, possibly even the extent, of structural similarity between individuals is an important asset in social studies.

One significant difficulty associated with identifying elements that are equivalent is that there are a few different perceptions of equivalence (detailed later in section 2.4), varying with strictness. For example, the strictest notion of equivalence states that two elements are equivalent if they are connected by the same relations to exactly the same elements (Burt, *Positions in Networks*, 1976). In contrast, a more general equivalence states that two elements are equivalent if they share the same *types* of neighbours (White & Reitz, 1983).

Knowing which level of equivalence to use is clearly dependent upon the task, but is not always obvious. As highlighted by Borgatti and Everett (1992), researchers have been known to misidentify the most rigid notion of structural equivalence – identical connections – with the more abstract idea of social roles; this is exemplified by the following claim:

“Even more important, if a researcher uses structural equivalence as a criterion, he can identify positions, or sets of individuals, that correspond to social roles in the network of communication.” (Caldeira, 1988, p. 46)

This misidentification of equivalences leads researchers to use a different notion of equivalence than the one they specified. For example, Galaskiewicz and Krohn wrote,

“A key concept in our study is the social role. . . . Two actors are in the same relative position in social space, i.e., have the same role, to the extent that they have similar relationships to others in the social arena or fields. . . . To phrase this in the context of resource dependency theory, two actors occupy the same structural position, i.e., role in the network, to the extent that they are dependent upon the same organizations for the procurement of needed input resources and the same actors are dependent upon them for their output resources.” (pp. 528-529)

Galaskiewicz and Krohn proceeded to use a more rigid, absolute equivalence to identify organisational roles (p. 532) instead of the relative equality they described.

A further difficulty is the choice of measures used in the analysis of the network. There are a number of graph-theoretic properties that may be considered, such as degree, betweenness, closeness, reach, etc. (The most commonly used metrics are described in section 2.3) Certainly a combination of these is required to produce good estimates of similarity, and all of them must be taken into account to achieve the greatest accuracy. Considering many properties becomes unwieldy and infeasible, so in reality many of these measures are omitted in favour of the most influential ones. But in what circumstances should, for example, betweenness measures be used in favour of closeness measures, and vice versa? This is a hard question for which there is no clear-cut answer.

1.2. Project Proposal

The proposal was to devise a statistics-based algorithm that would cluster together the similar elements of a graph. In doing so, it was hoped that the strict mathematical boundaries of graph-theoretical equivalence would not have to be adhered to; the algorithm would be capable of grouping together elements that, although they will ideally be highly similar, are not necessarily equivalent by any of the formal definitions.

This flexibility is hoped to provide an additional viewpoint when clustering elements of graphs. As an analogy – one that will be used throughout this report – consider the slightly contrived case of mothers. Imagine there are three mothers, the first with a single child, the second with eight children, and the third with ten. One definition of equivalence says these mothers are all different as they have a different number of children. The definition of equivalence one generalisation above this says that all the mothers are the same as they all have children. There is a thought that it would be useful if we could distinguish the first mother from the other two – those with *many* children – as this is a more realistic and intuitive grouping. This is what the project aims to fulfil.

1.3. Existing Work

There exists extensive work into defining the different concepts of equivalence – and thus the different notions of roles, or positions, of nodes – in relation to the elements of graphs. Following from this, algorithms on how to cluster together elements of graphs based on a chosen equivalence definition have been published.

There are three major, well-known definitions of equivalence that have been well-researched and documented: *structural equivalence*, by Lorrain and White (1971), Burt (1976), and Breiger, Boorman, and Arabie (1975); *automorphic equivalence* or *structural isomorphism*, by Everett (1985) and Winship (1988); and *regular equivalence*, by White and Reitz (1983) and Borgatti and Everett (1989). These are explained fully in section 2.4. There are also other general or abstract equivalences looked at by Winship and Mandel (1983), Breiger and Pattison (1986), and Hummell and Sodeur (1987), but these are out of the scope of what is needed for the project.

Everett and Borgatti have presented notable algorithms to help cluster elements of a graph based on automorphic equivalence (Calculating Role Similarities: An Algorithm that Helps Determine the Orbits of a Graph, 1988) and on regular equivalence (Two Algorithms for Computing Regular Equivalence, 1993). The essential points of the original algorithm for computing the extent of

regular equivalence, REGE, were presented by White in three unpublished papers (1980) (1982) (1984) (Borgatti & Everett, 1993).

To the author's knowledge, there has been no prior work on statistical clustering of graph elements based upon the aforementioned perceptions of equivalence.

1.4. Objectives

The primary objective was to develop an algorithm that clusters the elements of a graph based on statistical measures. The resulting clustering was to resemble a middle-way between two generalisation levels of equivalence, providing a third, alternative clustering, as described above. This can be broken down into three milestones:

Milestone 1: Implement an existing algorithm to be used as a reference point, which the new algorithm will be compared against. This algorithm will apply a relatively strict equivalence (automorphic equivalence) to constrain the size of the clusters. Therefore a more flexible approach to clustering the elements of the same graph will be expected to have larger-sized clusters.

Milestone 2: Develop and implement a new algorithm, primarily based on statistical measures. The clustering that is produced should resemble that produced by the existing algorithm; elements that are grouped together via the rigid graph-theoretic approach should still be grouped together in this new approach.

Milestone 3: Evaluate the clustering produced by the new algorithm against that produced by the existing algorithm. There should be a numerical measure that can easily communicate the similarity between the two clusterings. Ideally this will show that the clusterings are neither identical nor dissimilar, and that clustered-together elements in the existing algorithm are clustered together in the new one.

1.5. Report Overview

Chapter 2 will elucidate, in detail, the different equivalence concepts before explaining the background mathematics and graph theory required to understand what Everett and Borgatti's existing algorithm does and why. The algorithm itself will then be explained.

Chapter 3 will present the proposed new statistics-based algorithm and explain the techniques used, before describing the testing that took place and the similarity measures developed in order to evaluate the output from the two algorithms.

Chapter 4 will detail the reasoning behind the implementation and technical choices made, and will demonstrate some of the more interesting and difficult aspects of the development.

Chapter 5 will consider the result of the new algorithm and will show how to use the algorithm in the environment it was developed.

Chapter 6 concludes the report, with reflections on goal achievement and on the further activity that could improve the algorithm, and the overall tool, to enhance its usefulness.

2. Background

2.1. Notions of Position

Position refers to how an actor¹ interacts with the rest of the network, and therefore has connotations of structural correspondence or similarity. Actors who are connected in the same way to the rest of the network are said to occupy the same position. This in turn leads to the notion of equivalence; those actors that occupy the same position – have the same role – are equivalent (Borgatti & Everett, 1992).

2.1.1. Actor Similarity vs. Actor Cohesion

The phrase “*connected in the same way to the rest of the network*” used above can, however, be interpreted in at least two fundamental ways.

The first is in a literal sense, and is the most basic interpretation. The underlying clustering principle in this case is cohesion, or proximity; two actors are connected in the same way to the rest of the network if they are tied to exactly the same nodes. The second interpretation is a metaphorical one. Here the underlying principle is similarity; two actors are connected in the same way to the rest of the network if the actors they are linked to also exhibit similarity.

Borgatti and Everett (1992) adapt a problem presented by Hofstadter (1985) to illustrate this distinction and ask the reader to consider the following two abstract structures:

$$A = \langle 1\ 2\ 3\ 4\ 5\ 5\ 4\ 3\ 2\ 1 \rangle \text{ and } B = \langle 0\ 1\ 2\ 3\ 4\ 4\ 3\ 2\ 1\ 0 \rangle$$

“Hofstadter asks, “What is to *B* as 4 is to *A*? Or, to use the language of roles: What plays the role in *B* that 4 plays in *A*?” (p. 549). According to Hofstadter, an overly literal, concrete answer is 4, whereas a more natural, more analogical response is 3.” (p. 3)

This distinction is also exemplified in mathematics. In the case of geometry, two rectangles are the same if corresponding sides are of equal length – this is a literal interpretation of “the same”. Under the metaphorical interpretation, two rectangles would be declared the same if they are similar; if they have the same aspect ratio (and therefore the same proportions).

2.1.2. Positional Analysis

Positional analysis, therefore, aims to partition actors into mutually exclusive sets of equivalent actors who share *similar* interactivity with the network. In other words, the analysis groups together actors that play the same *role* in the network. The contrasting cohesive approach (Burt, 1978)

¹ Note that the following terms are used interchangeably throughout this report. Their use is dependent on the context of the material and the level of abstraction employed.

- Graph, network;
- Node, vertex, actor, point, graph element;
- Edge, relationship, link, tie, connection;
- Position, role, type.
- Position, role, type.

(Friedkin, 1984) aims to identify closely related actors – those that have a high proximity and share identical neighbours.

2.2. Technical Definitions and Notation

In this section, the basic technical definitions and notation used throughout this report are given. These definitions will be used in subsequent sections to build upon and define more complex ideas, so it is important that they are clarified early on. The definitions and notations used here correspond with those given by Borgatti and Everett (1992).

Graphs or networks are represented as graphs denoted $G(V, E)$, where V refers to a set of vertices (or nodes, actors, points, graph elements) and where E refers to a set of edges (or relationships, links, ties, connections). The vertex sets of graph G and H are represented as $V(G)$ and $V(H)$ respectively. Similarly, the edge sets of the two graphs are represented as $E(G)$ and $E(H)$.

Furthermore, the position of node a is denoted $P(a)$. The position of a node is an attribute of that node that can be, and is, used to classify the node – that is, the node is identified as being of a specified type, occupying a particular position, or playing a certain role. Such categorisation allows positions to be instinctively visualised as colours (Everett & Borgatti, 1990). Extending positional notation, let $P(\{a, b, \dots\}) = \{P(a) \cup P(b) \cup \dots\}$, i.e., if S is a set of nodes then $P(S)$ is the set of distinct positions occupied by the nodes in set S .

In an undirected graph, the notation $N(a)$ is defined as the set of nodes directly connected to node a , so that $N(a) = \{c: (a, c) \in E\} = \{c: (c, a) \in E\}$. $N(a)$ is referred to as the neighbourhood of a . If S is a subset of V then $N(S) = S \cup \{v \in V - S: (v, s) \in E, s \in S\}$; nodes that are linked directly to the subset S are part of their neighbourhood.

This project does not take into account directed graphs, but for the purposes of completeness their neighbourhoods are defined as follows. The set of nodes that a receives ties from is known as the in-neighbourhood, $N^{in}(a)$, is defined as $N^{in}(a) = \{c: (c, a) \in E\}$. Likewise, the set of nodes that a sends ties to is known as the out-neighbourhood, $N^{out}(a)$, is defined as $N^{out}(a) = \{c: (a, c) \in E\}$. The neighbourhood $N(a)$ of node a then is defined as the ordered pair $N(a) = (N^{in}(a), N^{out}(a))$.

From these simple definitions it can be seen that more interesting expressions can be built. For example, $P(N(a))$ – the set of positions occupied by the nodes with ties to a – and $N(P(a))$ – the set of nodes that are linked to a node by the position that is occupied by a .

To take a simple example and to demonstrate the above concepts, consider the basic graph shown in Figure 1 below:

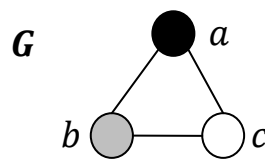


Figure 1: Graph with 3 vertices and 3 edges.

V or, more precisely, $V(G)$, is the vertex set $\{a, b, c\}$, and E or $E(G)$ is the edge set $\{ab, ac, bc\}$. Here the nodes have been coloured to enable uncomplicated representation of positions. Therefore, $P(a)$ is black $P(b)$, is grey, and $P(c)$ is white. Additionally $P(G)$, is the set $\{black, grey, white\}$. This graph is undirected so $N(c)$ is simply the set of nodes $\{a, b\}$. To move to the slightly more complicated examples, $P(N(b))$ is the set of positions $\{black, white\}$ and $N(P(b))$ is the set of nodes that are neighbours to nodes of a *grey* position, i.e., $\{a, c\}$.

Nodes have attributes; if an attribute makes no reference to the names or labels of the nodes in a graph then these are classified as *structural* or *graph-theoretic attributes*. For example, the property of being directly connected to every other node in the graph is a structural attribute. The property of being directly connected to node b is not a structural attribute. In the example in Figure 1 above, a exhibits both of these properties but only the former is a structural attribute.

2.3. Graph Theory

This section builds upon the mathematics of the previous section and introduces some graph-theoretic concepts. These will be required to understand the foremost equivalence definitions as well as the algorithms currently used to implement the notion of position as equivalence.

2.3.1. Mapping: Isomorphism

Two graphs, G and H , are isomorphic if there is a one-to-one mapping from the nodes of G to the nodes of H that preserves the adjacency of nodes (Wasserman & Faust, 1994). Thus an isomorphic graph preserves the relationships among the objects of the original graph once a one-to-one mapping has been undertaken. Therefore, if two nodes are connected in the original graph, their corresponding nodes in the second, isomorphic graph will also be connected. A formal definition for this is that a graph isomorphism between two graphs G and H is a mapping $\pi: G \rightarrow H$ such that for all $a, b \in V(G)$, $(a, b) \in E(G) \leftrightarrow (\pi(a), \pi(b)) \in E(H)$ (Borgatti & Everett, 1992).

An example of two isomorphic graphs and the mapping between them is given in Figure 2 and Table 1 below.

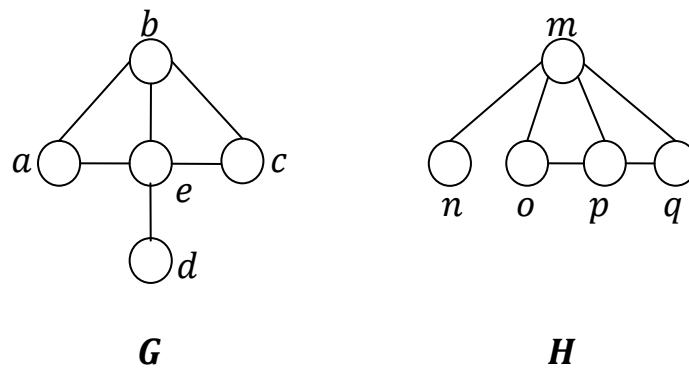


Figure 2: Graphs G and H are isomorphic.

Table 1: The graph isomorphism $\pi_1: G \rightarrow H$ relating the graphs in Figure 2.

g	$\pi_1(g)$
a	o
b	p
c	q
d	r
e	s

Isomorphic graphs are, unsurprisingly, identical with respect to all graph-theoretic properties. A useful way of thinking about whether or not two graphs are isomorphic is to imagine morphing one on top of the other. If, once the two graphs were stripped of their vertex and edge labels, one of the graphs could be picked up off the paper and then rearranged and scaled so that it can be placed perfectly on top of the other graph, they are isomorphic. At this point, without labels, the two graphs would be identical. From this it is clear that the graphs would have identical graph-theoretic properties. The only possible differences between isomorphic graphs are non-structural attributes – any labels of the nodes and edges.

2.3.2. Mapping: Automorphism

All graphs are isomorphic with themselves. More formally, for all graphs there exists a mapping $\pi: G \rightarrow G$ such that π is an isomorphism. This isomorphic mapping of a structure to itself is known as an *automorphism*. π can always be the identity mapping (for all $v \in V, \pi(v) = v$) – which is why all graphs have at least one automorphic mapping – but graphs often have automorphic mappings that are not the identity.

As an example taken from Borgatti and Everett (1992), consider the graph in Figure 3. By observing the symmetries of the graph, one can easily perceive its automorphisms. If all labels were removed, a rotation of 180° along the horizontal axis would prove indistinguishable from the original graph. Likewise, a rotation of 180° along the vertical axis would have a similar effect, as would a composition of the two rotations. Table 2 shows all the automorphism of the graph.

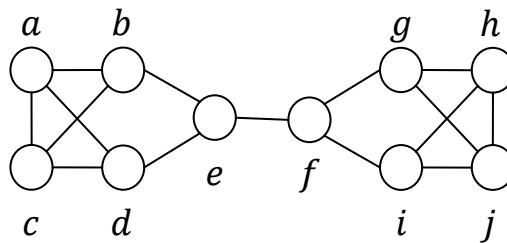


Figure 3: Graph with four automorphisms.

Table 2: All automorphisms of the graph in Figure 3.

v	$\pi_1(g)$	$\pi_2(g)$	$\pi_3(g)$	$\pi_4(g)$
a	h	c	j	a
b	g	d	i	b
c	j	a	h	c
d	i	b	g	d
e	f	e	f	e
f	e	f	e	f
g	b	i	d	g
h	a	j	c	h
i	d	g	b	i
j	c	h	a	j

Upon examination of Table 2 one notices that groups of nodes become interchangeable. For example, when any node from the set $\{a, c, h, j\}$ is mapped automorphically, the mapping can only ever be to a node from that same set. These sets formed by isomorphic actors are known as *orbits*. The orbits of the graph in Figure 3 are $\{a, c, h, j\}$, $\{b, d, g, i\}$, and $\{e, f\}$.

2.3.3. Centrality: Degree

The three significant centralities were first comprehensively defined by Freeman (1978). The first of these is *degree centrality*. Degree centrality is a straightforward index of the activity of a node, a count of the number of adjacencies it has:

$$C_D(v_j) = \sum_{i=1}^n f(v_i, v_j) \text{ where } f(v_i, v_j) = \begin{cases} 1 & \text{if and only if } v_i \text{ and } v_j \text{ are linked} \\ 0 & \text{otherwise} \end{cases}$$

$C_D(v_j)$ will be large if v_j has many direct neighbours and will be small if v_j has little direct contact. Consequently, $C_D(v_j) = 0$ if v_j is in total isolation and is not connected to any other node in the network.

To further utilise the notation used by Borgatti and Everett, the *degree vector* of a graph G is defined by $C_D(G) = (C_D(v_1), C_D(v_2), \dots, C_D(v_n))$ for all $v_i \in V(G)$ where $n = |V|$, $C_D(v_i) \leq C_D(v_{i+1})$, for $i = 1, \dots, n - 1$.

Take graph G in Figure 4 below as an example.

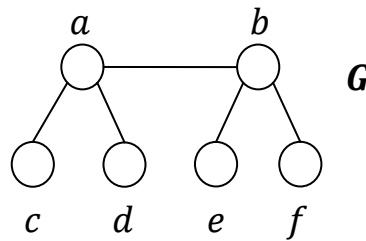


Figure 4: Graph with 6 nodes, 5 edges, and 2 orbits.

$C_D(c) = 1$, as does $C_D(d)$, whereas $C_D(a) = 3$. The degree vector $C_D(G) = (1, 1, 1, 1, 3, 3)$. Moreover, if S is a subset of V then $\langle S \rangle$ is the induced subgraph, i.e., the graph that would remain if only the vertices in set S existed. $C_D(S)$ will be written in place of $C_D(\langle S \rangle)$ for the purposes of readability. Hence, continuing with the above example, if the vertex subset $S = \{a, b, c, d\}$ then $C_D(S) = (1, 1, 1, 3)$. The subgraph $\langle S \rangle$ is depicted in Figure 5 below.

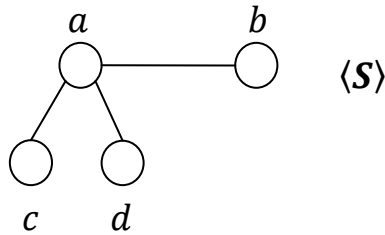


Figure 5: Subgraph of graph in Figure 4.

In order to compare the centralities of nodes from different graphs, this measure of degree can be normalised over $n - 1$, as a given node can be adjacent to at most $n - 1$ other nodes in a graph. Doing this removes the effect of network size, allowing relative comparison.

2.3.4. Centrality: Betweenness

The second view of centrality, as described again by Freeman (1978), is based upon the frequency with which a node lies on the shortest (geodesic) paths between pairs of other points. This is known as *betweenness centrality*. Consider the graph in Figure 5. There are six geodesics, as shown in Figure 6 below:

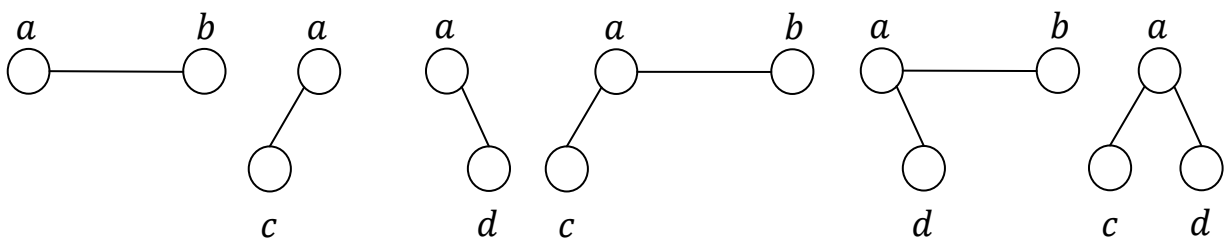


Figure 6: The six geodesics of the graph in Figure 5.

Here it can be seen that a lies on all of the shortest paths between every node in the graph; the first three geodesics are of length one and represent the shortest path from a to every other node. The final three geodesics show the shortest paths between the remaining pairs of nodes bc , bd , and cd . This means that a must be navigated in order for any two nodes to communicate, that a exhibits a potential for control of their communication. It is this potential for control that defines betweenness. Thus, the greater number of geodesics a node lies upon, the greater its betweenness.

Note that there may be more than a single geodesic between a pair of nodes. This must be taken into consideration when calculating betweenness. Let g_{ij} be the number of geodesics linking n_i and n_j , and $g_{ij}(v_k)$ be the number of those geodesics that contain v_k . $\frac{g_{ij}(v_k)}{g_{ij}}$ then gives the probability that node v_k falls on a randomly selected geodesic connecting v_i with v_j .

To calculate the overall betweenness centrality of node v_k , the probabilities of v_k lying on a randomly selected shortest path between two nodes must be summed for all pairs of nodes. The exceptions are node-pairs that are directly connected (and therefore need not traverse through any additional nodes) – they are ignored.

$$C_B(v_k) = \sum_i^n \sum_j^n \frac{g_{ij}(v_k)}{g_{ij}} \text{ where } i < j, i \neq j \neq k$$

In a similar manner to the degree vector, the *betweenness vector* of a graph G is defined by $C_B(G) = (C_B(v_1), C_B(v_2), \dots, C_B(v_n))$ for all $v_i \in V(G)$ where $n = |V|$, $C_D(v_i) \leq C_D(v_{i+1})$, for $i = 1, \dots, n - 1$.

Again consider the subgraph in Figure 5. As a lies on three geodesics that do not include a as either the start or end node, $C_B(a) = 3$. By the same token, the remaining nodes all have a betweenness of 0 as they are never on the shortest path between two other nodes. To demonstrate a slightly more complex example, take graph G shown in Figure 7 below and the matrix of shortest paths in Table 3:

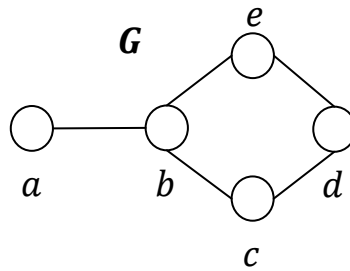


Figure 7: Graph where some node-pairs have two geodesics.

Table 3: Matrix of shortest paths of graph in Figure 7.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>		-	<i>b</i>	<i>b, c</i> <i>b, e</i>	<i>b</i>
<i>b</i>			-	<i>c</i> <i>e</i>	-
<i>c</i>				-	<i>b</i> <i>d</i>
<i>d</i>					-
<i>e</i>					

As a is not seen within the matrix, $C_B(a) = 0$. That is, none of the shortest paths found in the graph contain a . b , however, can be seen numerous times. Its betweenness is calculated by summing the probabilities of b occurring in the shortest path, i.e.:

Node b lies on the single geodesic between a and c , therefore its probability of occurrence is 1. Likewise, b lies on both of geodesics between a and d and on the single path between a and e . Of the two shortest paths between c and e , b can only be found on one of them. Therefore, the probability in this case is 0.5. Consequently, $C_B(b) = 1 + 1 + 1 + 0.5 = 3.5$.

The rest of the nodes have the following betweenness values: $C_B(c) = 1$, $C_B(d) = 0.5$, $C_B(e) = 1$. The betweenness vector of graph G is $C_B(G) = (0, 0.5, 1, 1, 3.5)$.

As with degree centrality, betweenness centrality is dependent on the size of the network. For some applications it again may be useful to have a relative measure to compare betweenness of nodes from separate graphs. Freeman (1978) proved that relative betweenness can be expressed as the ratio $\frac{2C_B(v_k)}{n^2 - 3n + 2}$.

2.4. Equivalence Definitions

This section explores the three foremost varying notions of position, as explained by Borgatti and Everett (1991) (1992). Subsequently, the equivalence selected as the template to aim towards in the new algorithm is justified.

2.4.1. Position as Structural Equivalence

Structural equivalence is the most strict, or least general, of the three equivalences discussed at here. Under this notion an actor's position is determined solely by its local neighbourhood: if G is graph and $a, b \in V$, then $P(a) = P(b)$ if and only if $N(a) = N(b)$ ². In other words, an actor's position is defined by those actors it has direct connections to; a and b are structurally equivalent if a and b have identical relationships.

To draw on the "mothers" analogy used previously in the report, this is akin to saying that two mothers are equivalent if they are both mothers of the same child. Obviously this is impossible because of the nature of the "is-a-mother-of" relation, but serves as a comprehensible parallel.

Looking at graph G in Figure 4 it can be seen that nodes c and d are structurally equivalent because they both have a single relationship with node a . Similarly, e and f are structurally equivalent because of their connection with node b . Nodes a and b are *not* structurally equivalent because a

² This definition, although sufficient for the purposes of this report, has an important shortcoming: in graphs without reflexive loops, nodes with a direct connection to each other cannot occupy the same position. A slight alteration to the definition that resolves this problem is $P(a) = P(b)$ if and only if $N(a) - \{a, b\} = N(b) - \{a, b\}$. Still, this version too is not without problem. There is fault when a single directed arc links a and b , and when trying to distinguish nodes with reflexive loops from ones without (Borgatti & Everett, 1992).

The following definition provided by Everett, Boyd, and Borgatti (1990) works in all cases: $P(a) = P(b)$ if and only if $\exists \pi \in \text{Aut}(G)$ such that $\pi = (a \ b)$ and $\pi(a) = b$, where $\text{Aut}(G)$ denotes the automorphism group of a graph G .

has ties to c and d while b has ties to e and f . This can be visualised as a *role colouring*, as demonstrated in Figure 8 below; actors of the same colour play the same role shortcoming

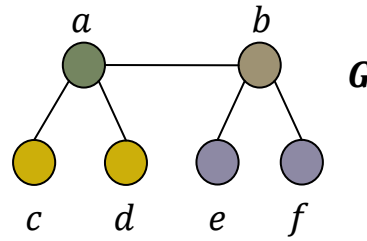


Figure 8: Graph in Figure 4 coloured according to structural equivalence.

Locating structurally equivalent actors produces cohesive subsets, as only direct adjacencies are taken into account. This reveals as much about the proximity of actors as it does their similarity. Of course, two actors with identical neighbourhoods will have a high degree of similarity. On the other hand, structural equivalence is incapable of detecting similarity *in spite of* proximity, a trait that has clear advantages.

2.4.2. Position as Automorphic Equivalence

Automorphic equivalence – also often used interchangeably with the term *structural isomorphism*, but in the interests of clarity only referred to as automorphic equivalence in the rest of this report – is an equivalence that observes similarities between two nodes' interactivity within a network regardless of the identity of the nodes in their neighbourhoods.

This is accomplished by viewing two actors as occupying the same position, not if they are tied to the same actors, but if the actors they are tied to correspond. Whereas the neighbours of structurally equivalent actors contain the same actors, the neighbours of automorphically equivalent actors contain the same *positions*. Formally, if actors a and b are isomorphic, then for all $c \in V$, $(a, c) \in E$ implies there exists an actor d isomorphic to c such that $(b, d) \in E$ (Borgatti & Everett, 1992).

Under this approach, $P(a) = P(b)$ if and only if $P(N(a)) = P(N(b))$; if the positions occupied by the neighbourhood of a is equal to the positions occupied by the neighbourhood of b , a and b are automorphically equivalent. Actors that occupy the same positions in this automorphic sense can be interchanged in a mapping, and therefore belong to the same orbit (as described earlier in section 2.3.2). Thus, finding automorphically equivalent actors becomes a case of calculating the orbits of a graph.

Table 4: All automorphisms of the graph in Figure 4.

v	$\pi_1(g)$	$\pi_2(g)$	$\pi_3(g)$	$\pi_4(g)$
a	a	a	b	b
b	b	b	a	a
c	d	c	e	f
d	c	d	f	e
e	e	f	c	d
f	f	e	d	c

Table 4 above shows all the automorphisms of the graph in Figure 4. Using this, one can identify the orbits of the graph as $\{a, b\}$ and $\{c, d, e, f\}$. Two orbits correspond to two different types of nodes (positions) in the graph. Once again this can be visualised as a role colouring as depicted in Figure 9.

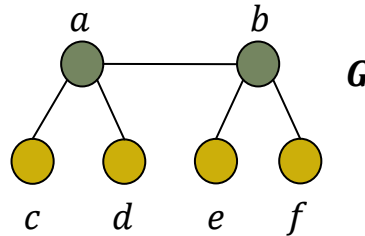


Figure 9: Graph in Figure 4 coloured according to automorphic equivalence.

Continuing with the “mothers” analogy, automorphic equivalence would say that two mothers are the same if they have the same number of children.

2.4.3. Position as Regular Equivalence

Regular equivalence is the least strict, or most general, of the three equivalences discussed, and an abstraction above automorphic equivalence. Returning to the “mothers” analogy once again, here two mothers are the same if they have children, regardless of how many.

Whereas automorphic equivalence enumerates how many of each type one particular type is connected to, regular equivalence is satisfied as long as there exists at least one connection to all the necessary types. As an example, if graph G in Figure 4 was to be extended by connecting a new node, g , to b and coloured in accordance to regular equivalence, one possible colouring can be seen in Figure 10 below:

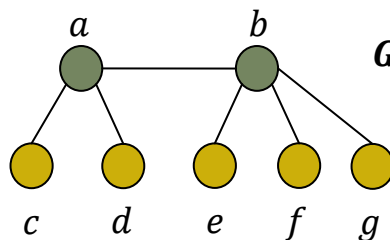


Figure 10: Extension of graph in Figure 4 coloured according to regular equivalence.

Here each green node is linked to at least one other green node and one yellow node. Each yellow node is linked to one green node. Automorphic equivalence is now broken because if one green node has ties to two yellow nodes, a green node elsewhere cannot have ties to three yellow nodes.

Herein lies the significant drawback of regular equivalence: its leniency. Although there are undoubtedly applications where disregarding the quantity of a type of neighbour is useful, it seems counter-intuitive. For most practical purposes it inherently feels astute to distinguish between

actors that have a single connection to a given type of actor from actors that have hundreds of such connections.

2.4.4. Selected Equivalence & Justification

From the three equivalences discussed in this chapter, automorphic equivalence seems to be the most obvious target for the new algorithm. Not that the new algorithm will simulate automorphic equivalence, but it will aspire to resemble a somewhat more relaxed version of it. Clearly too much leniency will result in regular equivalence, so there must also be a level of restraint.

As mentioned earlier, the weakness of regular equivalence is its ability to judge two actors as equivalent independent of how many type x nodes may be connected to it. On the other end of the spectrum is structural equivalence, which is far too strict for most realistic intentions. It would be unusual, for example, to declare that two surgeons play the same role only if they operate on the same patients. A more reasonable assertion is that two surgeons play the same role if they operate on the same types of patients; surgeons that operate on patients with brain tumours are neurosurgeons. This is an illustration of regular equivalence, and works well in this instance.

Automorphic equivalence is almost a good halfway point between the two extremes. In the “mothers” analogy, two mothers are regularly equivalent if they have children, even if one mother has one child and the other has ten. The two mothers are automorphically equivalent if they have the same number of children. On the face of it this appears to be a sensible to claim, but there is a pragmatism behind grouping mothers of nine children together with mothers of ten because in reality there is not going to be much discernable difference. In other words, there is benefit in assigning mothers to intermediary groups: those with few children, those several children, and those with a lot of children.

This is something that automorphic equivalence is too strict to allow, whereas regular equivalence is too lenient. What’s more, it is perceivable that taking a statistics-based approach to this problem could result in a clustering of actors that is indicative of this intermediate view.

The new approach needed something to compare its clustering to. It was conceived that the results were going to be resemble the clustering produced under the view of automorphic equivalence more so than the extremely general view of regular equivalence. There would also be many more clearly defined sets of actors that would be useful in the analysis and evaluation of the new approach. This explains why the notion of position as automorphic equivalence was chosen to be the chief objective to aim towards and to be compared against.

2.5. Existing Algorithm

Everett and Borgatti (1988) outline an algorithm that helps determine the orbits of a graph. This in turn partitions the vertices of a graph into distinct automorphically equivalence classes. Their algorithm is presented in this section.

2.5.1. Additional Mathematical Definitions

The basic technical definitions in section 2.2 are expanded upon here, as they are required to understand the algorithm proposed by Everett and Borgatti.

Higher-order neighbourhoods are defined inductively: $N^{i+1}(S) = N(N^i(S))$ where S is a subset of a vertex set V . To reinforce this, consider the graph in Figure 4. If $S = \{c\}$ then the immediate neighbourhood $N(S) = \{a, c\}$, the second-order neighbourhood (neighbours of neighbours) $N^2(S) = N(N(S)) = N(\{a, c\}) = \{a, b, c, d\}$, and the third-order neighbourhood $N^3(S) = \{a, b, c, d, e, f\}$ as c is no more than three links from every other node in the graph.

If $x \in V$ then the *point-deleted neighbourhood* $\tilde{N}^i(x)$ is defined as $N^i(x) - \{x\}$. In the current example where $S = \{c\}$ then $N(S) = \{a, c\}$ and $N^2(S) = \{a, b, c, d\}$ whereas $\tilde{N}(S) = \{a\}$ and $\tilde{N}^2(S) = \{a, b, d\}$. The subgraphs produced by these two point-deleted neighbourhoods are shown in Figure 11 below. From looking at these we can easily observe their degree vectors. $C_D(\tilde{N}(S)) = (0)$ and $C_D(\tilde{N}^2(S)) = (1, 1, 2)$.

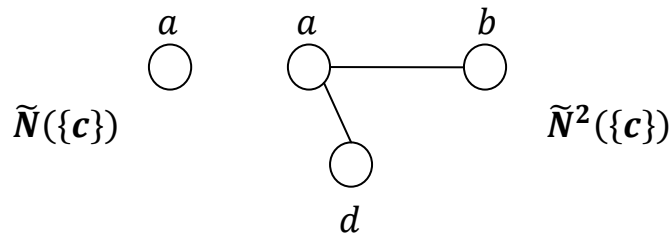


Figure 11: Point-deleted neighbourhoods of graph in Figure 4.

2.5.2. Finding Orbits

The algorithm proposed by Everett and Borgatti is based on the fact that any property not shared by two vertices guarantees that they are in separate orbits. To illustrate this, consider Figure 9, in which each orbit has been assigned a distinct colour. Partitioning the vertex set of G by degree presents two sets, $\{a, b, c, d\}$ that have a degree of 1 and $\{e, f\}$ that have a degree of 3. This has found the orbits of the graph. Usually, unfortunately, the problem will not be so simple. For more complex graphs there is a need to examine the degrees of the neighbours of a node, as opposed to the node itself. Everett and Borgatti (1988) provide the following demonstration of this using the graph in Figure 12 below:

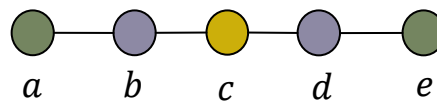


Figure 12: Graph with 5 nodes and 4 edges, coloured to highlight its orbits.

The orbits of this graph are $\{a, e\}$, $\{b, d\}$, and $\{c\}$. Partitioning the vertex set with respect to degree only, the obtained sets are $\{a, e\}$ and $\{b, c, d\}$. If the neighbourhoods, and higher-order

neighbourhoods, of the nodes are inspected instead, the following Table 5 is obtained:

Table 5: Degree vectors of neighbourhoods of the graph in Figure 12.

x	$\tilde{N}(x)$	$C_D(\tilde{N}(x))$	$\tilde{N}^2(S)$	$C_D(\tilde{N}^2(x))$
a	$\{b\}$	(0)	$\{b, c\}$	$(1, 1)$
b	$\{a, c\}$	$(0, 0)$	$\{a, c, d\}$	$(0, 1, 1)$
c	$\{b, d\}$	$(0, 0)$	$\{a, b, d, e\}$	$(1, 1, 1, 1)$
d	$\{c, e\}$	$(0, 0)$	$\{b, c, e\}$	$(0, 1, 1)$
e	$\{d\}$	(0)	$\{c, d\}$	$(1, 1)$

Examining the degree vectors of immediate (point-deleted) neighbourhoods distinguishes a and e from b , c , and d . Further examination of the degree vectors of second-order neighbourhoods then distinguishes c from b and d . Continuing to look at yet higher-order neighbourhoods produces more accurate results. In this instance, a second-order neighbourhood was all that was required to identify the orbits. For larger graphs this may not be the case.

It is also important to note that the actual orbits of the graph can only be finer partitions than those found at the highest-order neighbourhood considered. Therefore, a result that contained partitions $\{a\}$, $\{b, c\}$, and $\{d\}$, for example, has correctly distinguished those three sets as being different. The actual orbits, however, may be $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$. The partition of b and c will occur if the neighbourhood order is high enough, but at the cost of computation time.

Formally, x and y belong to the same set if and only if $C_D(\tilde{N}^i(x)) = C_D(\tilde{N}^i(y))$ for $i \geq 0$.

Although difficult, it is possible to find examples on which this method fails. A refinement of the above technique can be made to combat this; the inclusion of betweenness vectors as well as degree vectors. The combination of these two concepts, then, supplies the partitioning criteria: two vertices, $x, y \in V$ belong to the same set if the neighbourhoods are similar in respect to both degree and betweenness. That is, provided:

- (i) $C_D(\tilde{N}^i(x)) = C_D(\tilde{N}^i(y)), i = 1 \dots n - 1$
- (ii) $C_B(\tilde{N}^i(x)) = C_B(\tilde{N}^i(y)), i = 1 \dots n - 1.$

2.5.3. Measures of the Extent of Equivalence

The above method only identifies equivalence. Everett and Borgatti (1988) proceed to devise a simple measure of the extent of equivalence. First they define a function $SIM(i, x, y)$ that compares the degree and betweenness of actors x and y at neighbourhood level i , returning 1 if they are identical. Technically,

$$SIM(i, x, y) = \begin{cases} 1 & \text{if } C_D(\tilde{N}^i(x)) = C_D(\tilde{N}^i(y)) \\ & \text{and } C_B(\tilde{N}^i(x)) = C_B(\tilde{N}^i(y)) \\ 0 & \text{otherwise.} \end{cases}$$

Further refinements can be made to return the proportion of identical elements in the vectors if they are not identical.

Everett and Borgatti then define the extent of equivalence between two actors x and y as $A(x, y)$, the proportion of identical neighbourhoods:

$$A(x, y) = \frac{\sum_{i=1}^{n-1} SIM(i, x, y)/i}{\sum_{i=1}^{n-1} 1/i}$$

Therefore, actors that differ in their immediate neighbourhoods will be assigned a value close to 0, if not, 0. Whereas actors that differ in higher-order neighbourhoods closer to $n - 1$ are not penalised as heavily, and will score a value close to 1. Those that are identical in all neighbour levels will be assigned a perfect similarity of 1.0.

2.5.4. Pseudocode

The pseudocode for this algorithm, adapted from Everett and Borgatti, can be written as follows:

- Initialise equivalence matrix; $A(x, y) = 0$ for all x, y
- For each neighbourhood level $i = 1 \dots n - 1$ do:
 - For each actor $x = 1 \dots n$ do:
 - Generate neighbourhood $\tilde{N}^i(x)$
 - Compute degree vector $C_D(\tilde{N}^i(x))$
 - Compute betweenness vector $C_B(\tilde{N}^i(x))$
 - For each pair of actors $x = 2 \dots n$ and $y = 1 \dots x - 1$ do:
 - $A(x, y) = A(x, y) + SIM(i, x, y)/i$
 - $A(x, y) = A(x, y)/\sum \left(\frac{1}{i}\right)$

Applying this algorithm (considering up to third-order neighbourhoods) to the graph in Figure 12 produces the following equivalence matrix $A(x, y)$:

Table 6: Everett and Borgatti (1988) algorithm applied to graph in Figure 12.

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
<i>a</i>	1.000	0.591	0.455	0.591	1.000
<i>b</i>	0.591	1.000	0.761	1.000	0.591
<i>c</i>	0.455	0.761	1.000	0.761	0.455
<i>d</i>	0.591	1.000	0.761	1.000	0.591
<i>e</i>	1.000	0.591	0.455	0.591	1.000

Table 6 above shows which actors are equivalent – those with a value in the matrix of 1.000 – along with the similarity of actors that are not equivalent. For example, no actors are equivalent to c , but the orbit $\{b, d\}$ is more similar to c than the orbit $\{a, e\}$. This was also suggested by the difference in the vectors and their magnitudes in Table 5. Here a value is given to that idea.

2.5.5. Complexity

The worst case complexity of this algorithm occurs when the maximum order of the neighbourhood encompasses every other node, for every node in the graph. Therefore, approximately n degrees and betweennesses have to be calculated per node, or n times. This means the algorithm has a quadratic worst case time complexity, n^2 , or, in big O notation, $O(n^2)$.

3. New Algorithm

The proposed statistics-based algorithm utilises *k-means clustering*. Given k clusters, this analysis will partition the elements into the clusters with the closest mean.

3.1. *k*-Means Clustering Description

The following definition of *k-means clustering* was adapted from lecture notes written by Shapiro (2009) although first published by Lloyd in 1982 (Least Squares Quantization in PCM).

Clusters in the sense have three key factors: means, subsets, and the number of clusters. The mean of the data in cluster i is a vector of dimension D , and is denoted $m(i)$. S_i represents the set of data in cluster i . The subset S_i consists of all data points closest to the i th mean than any other mean. k refers to the number of clusters, and is selected in advance.

The objective of *k-means clustering* is to find the clustering that minimises the *quantisation error*,

$$\underset{S}{\operatorname{argmin}} \sum_{i=1}^k \sum_{x_n \in S_i} \|x_n - m(i)\|^2.$$

The standard algorithm to perform this clustering is implemented as an iterative refinement technique. It consists of an initialisation step proceeded by an assignment step and an update step. The final two steps are repeated until convergence, at which point there will be no change in the cluster assignments.

Initialising the means of the clusters is often performed based on a chosen heuristic, though it may be random. The following two steps are then repeated until the algorithm stabilises:

1. **Associate** the data with the cluster that has the closest mean:

$$S_i = \{x \mid x \in D, \|x - m(i)\|^2 \leq \|x - m(j)\|^2 \text{ for all } j\} \text{ for all } i$$

2. **Update** the means of the clusters based on the data now associated with them:

$$m(i) = \frac{1}{|S_i|} \sum_{x \in S_i} x; \text{ for all } i$$

3.2. Cluster Means

The proposed algorithm performs two stages of *k-means clustering*. The first stage clusters the elements of the graph according to degree. This preliminary clustering forms the foundation of the overall clustering. After all, two nodes of the same degree are more similar (from the viewpoint of automorphic equivalence) than two nodes of differing degree.

Consequently, if the range of degrees is greater than k , some nodes of differing degrees will be grouped together. The clusters are initialised with an even distribution to ensure that only nodes of similar degrees are grouped together, i.e., that a cluster that contains nodes with a degree of three

and four do not contain nodes of degree six when there exists a cluster containing nodes with a degree of five.

With this grouping of nodes with a similar activity, a secondary clustering then takes place. This secondary clustering focuses on neighbour *types*. An assumption is made that the number of sets produced by the initial clustering is synonymous with how many positions there are in the network. Each cluster now represents a certain position.

The secondary clustering calculates the mean number of connections each cluster has to every other cluster. In other words, how many ties to a neighbouring type, on average, each node in a group should have. This is how the second stage is initialised. The clustering then begins again. This time an assignment is made by looking at the amount of typed neighbours each node *actually* has.

This is where verbal explanations begin to get complicated, so consider the graph in Figure 9 for a graphical example. Let us assume the initial clustering on degree has produced the two sets depicted by the colouring. That is, two sets have been identified: cluster 1 contains $\{c, d, e, f\}$ with an average degree of 1, and cluster 2 $\{a, b\}$ with an average degree of 3. Repeating the assignment process according to degree will not change the clustering because the average degree is now the actual degree.

Initialisation of the secondary clustering would have the effect shown in the coloured Table 7:

Table 7: Results of clustering based on neighbour type of graph in Figure 9.

	Type 1 Neighbours	Type 2 Neighbours
Cluster 1	0	1
Cluster 2	2	1

On average, elements of the first cluster have no ties to elements of that same cluster, whereas they have a single tie with elements of the second cluster. In a similar fashion, elements of the first cluster have two type 1 neighbours and a single link to a type 2 node.

When assigning the data points to one of these clusters, a direct comparison cannot be done because there are an arbitrary number of dimensions to consider. As such, the *mean squared error*, $\sum_{i=1}^n (\bar{x}_i - \bar{x}'_i)^2$, is used.

Ideally, this neighbour-position based clustering will bring together elements that may have been initially narrowly separated by their degree but were closely related by the types of nodes they are connected to; analogous to grouping together two mothers, one with nine children and one with ten, after having originally been distinguished as dissimilar due to the differing number of children.

3.3. Pseudocode

Bearing in mind the assignment and update equations given in section 3.1, the pseudocode for this particular k -means clustering algorithm is, where $\{x_1, \dots, x_n\}$ are the nodes in the network:

- Initialise degree means of k clusters, for $i = 1 \dots k$ do:
 - $m(i) = (C_{D_{max}}/k) \times ((2i) + 1)$
- While degree means of the clusters have not stabilised:
 - For $i = 1 \dots k$ do (assign elements to clusters):
 - $S_i = \{\}$
 - For $v = 1 \dots n$ do:
 - If $\|x_v - m(i)\|^2 \leq \|x_v - m(j)\|^2$ for all $j = 1 \dots k, i \neq j$
 - $S_i = S_i \cup \{x_v\}$
 - For $i = 1 \dots k$ do (update cluster means):
 - $m(i) = \frac{1}{|S_i|} \sum_{x \in S_i} x$
- Initialise neighbour type means of resulting clustering for $i = 1 \dots k, S_i \neq \emptyset$ do:
 - For $j = 1 \dots k$ do:
 - $m(i)_j = \sum \sum_{v=1}^n \tilde{N}(x_v) \in S_j$ (count the neighbours of type S_j)
 - $m(i)_j = m(i)_j / |S_i|$
- While neighbour type means of the clusters has not stabilised:
 - Assign elements to clusters as above
 - Update cluster means as computed in the initialisation step

3.4. Complexity

When considering the time complexity of this suggested algorithm, there are a number of factors to take into account. To assign nodes to clusters, k number of n distances have to be calculated for each k . This is a quadratic time complexity of $O(k^2n)$. During the updating of cluster means, a maximum of n nodes are accounted for per cluster, giving a linear complexity of $O(kn)$. Therefore the overall time complexity is $O(k^2n)$.

This quadratic complexity is similar to the algorithm described in section 2.5, with one significant difference. When the size of a graph grows but the number of clusters remains the same, the automorphic role similarity algorithm has a quadratic growth whereas this algorithm has linear growth. It should also be noted that it is unlikely for the number of clusters to ever be greater than or equal to the number of nodes in the graph. That is, every actor plays a completely independent role. Realistically, the number of clusters in a large network is likely to be a small proportion of the number of nodes. Thus, for most intents and purposes, the k -means approach will be the faster of two algorithms.

3.5. Testing

A series of tests have been undertaken to ensure the algorithm was implemented correctly and to determine whether the k -means approach produces a clustering comparable to the automorphism-based orbit locator. There were three distinct areas of testing:

3.5.1. k -Means Clustering Stabilisation

First was evidence of convergence. If data points converge into clusters, the algorithm will stabilise. To show that this was taking place, the change in the mean of the clusters during each iteration was

recorded. This was done for both the initial clustering on degree and the secondary clustering on neighbour types. With the introduction of larger graphs it became apparent that absolutely zero change in the mean would either not occur or would take an infeasible length of time. As such, the stopping criterion was set using a threshold. Instead of waiting for the clusters to stabilise completely, stabilisation was asserted when the change in the mean was considered to be negligible. In these circumstances, the clustering is thought to be “good enough”.

The threshold was selected through trial and error. This method was employed due to the time constraints of the project despite the low likelihood of a reliable value. 0.1 was the threshold selected. That is, the clustering was “good enough” if the *squared* mean change of *every* cluster was less than or equal to 0.1. This threshold appeared to be fairly effective.

For the three networks in Appendix A: Data Sets, the change in the mean was plotted against the number of iterations. This was done once for the initial clustering, and again for the secondary clustering. These charts can be found in Appendix B: Stabilisation Charts. They all show a decreasing change in mean for most of the clusters as more iterations passed, and convergence in a short amount of time (nine iterations was the maximum).

3.5.2. Clustering Tests

In-depth testing of the two clustering methods also had to be undertaken. This involved first taking graphs with known orbits and verifying they match those returned by the orbit locating algorithm. Second, expected clusters had to be calculated based on the k -means algorithm, for given values of k . These quickly become difficult aspects to calculate by hand when networks grow. As such, mainly small, simple networks were devised in order to test the fundamentals of the two clusterings. These include empty graphs, graphs with isolated nodes, and graphs with reflexive loops. Also of interest are graphs with interesting patterns of connectivity, such as graphs where all nodes have the same degree, graphs where all nodes are contained within a single orbit, and graphs where all nodes inhabit an orbit of their own (that is, all orbits are singleton sets). Appendix C: Clustering Test Data details the test graphs used, along with their orbits and the expected clusters for certain k -values.

JUnit testing showed that the k -means algorithm worked as expected. It also showed that the orbit-finding algorithm produced expected results when allowed to consider third-order neighbourhoods. Originally the tests were applied with the orbit locator using a maximum neighbourhood level of two. This worked fine for all networks other than the one in C - 18, which needed to be examined at an extra level to identify the additional partition.

3.5.3. Measures of Similarity: Orbit-Cluster Equivalence

As the two algorithms appear to work as expected, it is worth developing a numerical measurement denoting the equivalence of the two clusterings. In this project, two such measures have been used as they represent, once again, different notions of equivalence

The first equivalence, as far as the author is aware, has been devised during and for the project. It has been dubbed *orbit-cluster equivalence*. Orbit-cluster equivalence attempts to compare a clustering of graph elements to the graphs orbits. The assumption was made that if the elements of a given orbit were clustered together in the k -means clustering, regardless of the other elements in

that cluster, then this was a good thing. It shows that those particular elements have been grouped together correctly, and drawn in others from different orbits that are presumably similar.

The idea is to find how *accurately* the clusters (detected using k -means clustering) *cover* the orbits (detected using the orbit finding algorithm) in terms of their elements. From this, the notion of *coverage* and *accuracy* were formed. Coverage represents the proportion of the orbit elements that can be found within the cluster, i.e., the percentage of orbit elements matched. Accuracy represents the proportion of the cluster elements that are elements of the orbit, i.e., the percentage of cluster elements that also belong to the orbit. Therefore, coverage penalises if the cluster does not contain orbit elements, whereas accuracy penalises if the cluster contains elements other than those found in the orbit. Combining these two metrics together as one measure, *coverage + accuracy – orbit-cluster equivalence* – gives,

$$\frac{|S_O \cap S_C|}{|S_O|} \lambda + \left(1 - \frac{|S_C/S_O|}{|S_C|}\right) (1 - \lambda).$$

S_O and S_C are the orbit and the cluster currently being compared, respectively. λ is a bias between 0 and 1. If λ is closer to 1, coverage will hold more weight when determining equivalence. Conversely, if λ is closer to 0 then accuracy will carry more influence. For this project, a straightforward even bias of 0.5 was used. An overall value of 1 represents identical sets; 0 represents two sets that have no elements in common.

For example, consider the orbit $\{a, b, c, d, e\}$. If a cluster is the set $\{a\}$, there is a coverage of one in five, or 0.2, and an accuracy of 1. Put into the above equation, this gives an equivalence of 0.6. If the cluster contains an element that does not belong to the orbit, $\{a, x\}$, the accuracy drops to 0.5. The overall equivalence is now 0.35. Similarly, when the cluster covers the entire orbit but contains additional elements, the equivalence is penalised due to accuracy. A cluster $\{a, b, c, d, e\}$ has a perfect score of 1; a cluster $\{a, b, c, d, e, x\}$ receives a score of 0.917.

This can easily be extended to calculate the orbit-cluster equivalence between the entire set of orbits of a graph, and a clustering of the elements of that graph. For each orbit, find the orbit-cluster equivalence of the most similar cluster. Average these equivalence measures over the number of orbits. Technically,

$$\sum_{i=1}^n \operatorname{argmax} \left(\frac{|Y_i \cap S_C|}{|Y_i|} \lambda + \left(1 - \frac{|S_C/Y_i|}{|S_C|}\right) (1 - \lambda) \right) / |Y|$$

where Y now represents the set of orbits and Y_i represents the i th orbit.

3.5.4. Measures of Similarity: Point-Pair Equivalence

The second measure of similarity used to compare the new clustering to the orbits is point-pair equivalence. This is an objective criterion for the evaluation of two clustering methods, proposed by Rand (1971). Clearly, in this case the two clustering methods are the orbit-location algorithm and the k -means clustering. Whereas the orbit-cluster equivalence above is asymmetric, specifically comparing a clustering to the irrefutable orbits of a graph, this evaluation can be used between any two clustering methods.

The similarity measure is a perceptive one, and is based on three basic considerations. One, that every data point is indisputably assigned to a specific cluster. Two, that clusters are defined just as much by points they do not contain as those they do. Three, that all points are of equal importance in the determination of clustering. Rand concludes that a basic unit of comparison between two clusterings is how pairs of points are clustered.

If elements of a particular point-pair are located together in both clusterings, or assigned to different sets in both clusterings, this represents a similarity in the clustering. Conversely, elements of a point-pair are not similar if can are grouped together in one clustering but are grouped separately in the other.

As a quick illustration, consider two clusterings of four points. Let $Y = \{\{a, b\}, \{c, d\}\}$ and $Y' = \{\{a, b\}, \{c\}, \{d\}\}$. The point-pairs are tabulated as follows:

Table 8: Example of point-pair tabulation.

Point-Pair	<i>ab</i>	<i>ac</i>	<i>ad</i>	<i>bc</i>	<i>bd</i>	<i>cd</i>
Together in both	x					
Separate in both		x	x	x	x	
Mixed						x

There are six point-pairings. Of these six, one pair are together in both clusters and four pairs are separate. This makes five similar pairs out of six. Rand defined $c(Y, Y')$ as the number of similarly assigned point-pairs. This is what is known in this report as point-pair equivalence. In this case, $c(Y, Y') = 0.833$. More precisely, given n points, x_1, x_2, \dots, x_n , and two clusterings of them $Y = \{y_1, \dots, y_{k_1}\}$ and $Y' = \{y'_1, \dots, y_{k_2}\}$,

$$c(Y, Y') = \sum_{i < j}^n \gamma_{ij} / \binom{n}{2} \text{ where}$$

$$\gamma_{ij} = \begin{cases} 1 & \text{if there exist } k \text{ and } k' \text{ such that} \\ & \text{both } x_i \text{ and } x_j \text{ are in both } Y_k \text{ and } Y'_{k'} \\ 1 & \text{if there exist } k \text{ and } k' \text{ such that} \\ & x_i \text{ is in both } Y_k \text{ and } Y'_{k'} \text{ while } x_j \text{ is in neither } Y_k \text{ or } Y'_{k'} \\ 0 & \text{otherwise} \end{cases}$$

This point-pair equivalence complements well the aforementioned orbit-cluster equivalence. In most cases there should be a high degree of correlation. After all, high orbit-cluster equivalence implies orbit elements are found grouped together in clusters. This corresponds to high point-pair equivalence as those elements are located together in both clusterings. If the similarities do not correspond, analysis of the measures may perhaps give some insight into the underlying cause. Nevertheless, it is difficult to think of a scenario where this might occur.

4. Implementation

4.1. Code Language

The project was coded entirely in Java, for a number of reasons. A large factor in this decision was the *JGraphT* library. *JGraphT* handles graph structures and implements an array of graph-related algorithms, including Dijkstra's shortest path algorithm and the Edmonds-Karp maximum flow algorithm amongst others.

Two chief alternatives to Java were C and MATLAB. MATLAB would have excelled in some parts of the implementation since networks can easily be represented as matrices. Operations over these matrices would have been faster than Java and C. It was believed that there would be a significant number of non-matrix operations sufficient to balance out the efficiency gained by using matrices. C would have given more control over memory management, which could have proved useful with larger networks. The downside here is graph data structures and libraries in C were not found to be readily available and easily accessible, so would have to be written by hand.

Time restraints on the project meant that developing data structures and algorithms for them would have left very little time for the more interesting aspects: the clustering algorithms and the similarity measures. As *JGraphT* exists as open source and is very well documented, the decision was made to use the library as a time-saving foundation for the less common concepts dealt with in the project.

The *Eclipse IDE* was the chosen development environment for the Java project. A development environment is essential to the organisation and manageability of code. They contain extremely useful search and refactoring functions, as well as debugging tools and subversion controls that greatly help the fixing and restoration of code. Eclipse was chosen above other IDEs because it has been primarily developed for Java projects and is familiar as it is the one used in the Department of Computer Science at the University of Manchester.

4.2. File Format

Networks needed to be saved as files that could be input into the algorithm. Three formats were considered: *GML* (Graph Modelling Language), *GraphML*, and matrices. Matrices were swiftly disregarded, simply because the majority of data sets found online were either GML or GraphML format.

Both GML and GraphML (Appendix D: Input File Formats) are adequately able to represent graphs, and are both readable by the yEd graph drawing tool, but they do have significant differences. GML has existed as a format longer than GraphML and so there are more networks in GML format than GraphML. GraphML is a language based on XML and as such the syntax is more descriptive, but more complex, than GML.

GML turned out to be the obvious choice here. As a simpler format it would be easier to write a basic parser that only extracted the small number of properties required for the two clusterings, which could then be fed into the *JGraphT* library to construct a graph. There was also more data available, meaning a simple parser would provide access to more material.

4.3. Important & Difficult Aspects

Presented here are three noteworthy aspects of the implementation of the algorithms. They are either key constituents of the algorithms or a problematic issue for which an inelegant workaround was required, and all were challenging parts of the project.

4.3.1. Neighbourhood Generation

To locate orbits, one of the first iterative steps of the pseudocode in 2.5.4 is to generate the (point-deleted) neighbourhood of a node, $N^i(x)$. From this subgraph degree and betweenness vectors are calculated, so it is vital that it is correct. The `generateNeighbourhoodGraph()` method is shown below. Comments are denoted `//` and clearly outline what the code does.

```
private UndirectedGraph<Vertex, DefaultEdge>
    generateNeighbourhoodGraph(int givenNLevel, int givenActorIndex)
{
    // acquire the vertex set of the original supergraph
    // and add the considered actor to the set of subgraph vertices
    Set<Vertex> subVertexSet = new HashSet<Vertex>();
    subVertexSet.add(inputVertexSet[givenActorIndex]);

    // acquire the edges connected to the considered actor ("ego edge set")
    Set<DefaultEdge> egoEdgeSet =
        inputGraph.edgesOf(inputVertexSet[givenActorIndex]);

    // create a set to allow collection of the subgraph edges
    // and initialise with the "ego edge set"
    Set<DefaultEdge> subEdgeSet = new HashSet<DefaultEdge>();
    subEdgeSet.addAll(egoEdgeSet);
    ArrayList<Vertex> addedVertices = new ArrayList<Vertex>();

    // add all the source & target vertices of the subgraph edges
    // to the set of subgraph vertices
    // .....
    // each iteration produces a higher-order neighbourhood
    // (i.e. 1st iteration = immediate neighbours,
    // 2nd iteration = 2nd-order neighbourhood)
    for (int nLevel = 0; nLevel < givenNLevel; nLevel++)
    {
        // first, add all edges of vertices for which we want the neighbours
        // NOTE: always null for the first iteration
        // - i.e. only want neighbours of self
        if (addedVertices != null)
            for (int vertexIndex = 0; vertexIndex < addedVertices.size();
                vertexIndex++)
                subEdgeSet.addAll(
                    inputGraph.edgesOf(addedVertices.get(vertexIndex)));

        // then iterate over the set of edges collected so far
        // if edge has a source/target vertex
        // that isn't in the set of subgraph vertices
        // add this vertex to the set
        Iterator<DefaultEdge> edgeIterator = subEdgeSet.iterator();
        while (edgeIterator.hasNext())
        {
            DefaultEdge currentEdge = edgeIterator.next();
            if (!subVertexSet.contains(inputGraph.getEdgeSource(currentEdge)))
```

```

    {
        subVertexSet.add(inputGraph.getEdgeSource(currentEdge));
        addedVertices.add(inputGraph.getEdgeSource(currentEdge));
    } // if
    if (!subVertexSet.contains(inputGraph.getEdgeTarget(currentEdge)))
    {
        subVertexSet.add(inputGraph.getEdgeTarget(currentEdge));
        addedVertices.add(inputGraph.getEdgeTarget(currentEdge));
    } // if
    } // while
} // for

// finally add any additional edges between the added vertices
// by looping through all added vertices and checking for edges between
// and adding them to the set of subgraph edges if not already contained
if (addedVertices != null)
    for (int v1Index = 0; v1Index < addedVertices.size() - 1; v1Index++)
        for (int v2Index = v1Index + 1; v2Index < addedVertices.size();
            v2Index++)
        {
            DefaultEdge additionalEdge
                = inputGraph.getEdge(
                    addedVertices.get(v1Index), addedVertices.get(v2Index));
            if (additionalEdge != null)
            {
                if (!subEdgeSet.contains(additionalEdge))
                    subEdgeSet.add(additionalEdge);
            } // if
        } // for

// create the subgraph of the considered node and its neighbourhood
UndirectedSubgraph<Vertex, DefaultEdge> neighbourhoodSubgraph
    = new UndirectedSubgraph<Vertex, DefaultEdge>(
        inputGraph, subVertexSet, subEdgeSet);

// remove considered node from the graph
// to be left with just the subgraph of its neighbourhood
neighbourhoodSubgraph.removeVertex(inputVertexSet[givenActorIndex]);

return neighbourhoodSubgraph;
} // neighbourhoodSubgraph

```

The complexity of this method comes from the interaction with JGraphT to select the nodes to be used in the construction of a new UndirectedSubgraph – no method for neighbourhood subgraph generation existed within the library at the time. Despite this, the underlying process is quite straightforward. The first step is always to acquire the neighbours of the node currently under inspection. Then, for a second-order neighbourhood, adjacent nodes to the set found by the first step must also be acquired, and so on for higher-order neighbourhoods. JGraphT does not provide a mechanism for retrieving neighbouring points, but it can return the edges connected to a given node and so adjacencies can be determined via edge sources and targets.

4.3.2. Floyd-Warshall Algorithm

Once a subgraph of a node's neighbourhood exists the corresponding betweenness vector needs to be calculated. This involves counting how many geodesics each subgraph element lies upon. The Floyd-Warshall algorithm (Floyd, 1962) (Warshall, 1962) finds all the shortest paths between all pairs

of nodes. With this information, calculating betweenness is, essentially, a relatively simple matter of finding how many of these contain a specific node.

Unfortunately, the JGraphT implementation of Floyd-Warshall only returns the length of the shortest path between two specified nodes:

Constructor Summary	
	<u>FloydWarshallShortestPaths</u> (<u>Graph</u> < <u>V</u> , <u>E</u> > g) Constructs the shortest path array for the given graph.
Method Summary	
double	<u>getDiameter</u> ()
double	<u>shortestDistance</u> (<u>V</u> v1, <u>V</u> v2) Retrieves the shortest distance between two vertices.

Figure 13: JGraphT Floyd-Warshall algorithm documentation.

Again due to time constraints, writing a function to perform the Floyd-Warshall algorithm on data handled by an external library seemed impractical. After studying the other algorithms supplied by JGraphT a solution was conceived.

JGraphT contains another algorithm called *k*-shortest paths. This returns a list of paths between two nodes that have a maximum distance supplied as an argument:

Constructor Summary	
	<u>KShortestPaths</u> (<u>Graph</u> < <u>V</u> , <u>E</u> > graph, <u>V</u> startVertex, int k) Creates an object to compute ranking shortest paths between the start vertex and others vertices.
	<u>KShortestPaths</u> (<u>Graph</u> < <u>V</u> , <u>E</u> > graph, <u>V</u> startVertex, int nPaths, int nMaxHops) Creates an object to calculate ranking shortest paths between the start vertex and others vertices.
Method Summary	
java.util.List< <u>GraphPath</u> < <u>V</u> , <u>E</u> >>	<u>getPaths</u> (<u>V</u> endVertex) Returns the k shortest simple paths in increasing order of weight.

Figure 14: JGraphT *k*-shortest paths algorithm documentation.

Therefore, if the maximum distance argument is the shortest length between the two nodes, *k*-shortest paths will return only the geodesics. It follows that the two algorithms can be combined; for every node-pair, Floyd-Warshall can be used to find the shortest distance, which can be fed into

k -shortest paths as the maximum distance. The result should be a matrix of the geodesics from which betweenness can be calculated.

The actual code to do this is rather fiddly and intricate. Instead of the entire `calculateBetweennessVector()` method, then, below are a few snippets of it demonstrating only what has been explained above.

```
...

// create new FloydWarshall algorithm
// to know shortest distance between two nodes of the given graph
FloydWarshallShortestPaths<Vertex, DefaultEdge> floydWarshall
    = new FloydWarshallShortestPaths<Vertex, DefaultEdge>(givenGraph);

// create a matrix to store all the shortest paths
// between two given nodes
GraphPathVector[][] shortestPathsMatrix
    = new GraphPathVector[givenVertexSet.length][givenVertexSet.length];

...

// loop through each pair of nodes, finding the shortest paths
for (int startIndex = 0; startIndex < givenVertexSet.length;
     startIndex++)
    for (int endIndex = 0; endIndex < givenVertexSet.length; endIndex++)
    {
        // get the shortest distance between the nodes
        int shortestDistance
            = (int) floydWarshall.shortestDistance(givenVertexSet[startIndex],
                                                    givenVertexSet[endIndex]);

        ...

        // create a new K-shortest paths and use to get all the paths
        // that are the shortest distance (as max dist.) between two nodes
        KShortestPaths<Vertex, DefaultEdge> kShortestPaths
            = new KShortestPaths<Vertex, DefaultEdge>
                (givenGraph, givenVertexSet[startIndex], nPaths, shortestDistance);

        shortestPaths = kShortestPaths.getPaths(givenVertexSet[endIndex]);

        ...

        // store the shortest paths in the matrix of shortest paths
        shortestPathsMatrix[startIndex][endIndex]
            = new GraphPathVector(shortestPaths);
    }
}
```

4.3.3. Calculating Cluster Means (Secondary Clustering)

One consistent difficulty encountered during the course of coding the project was the level of nesting required due to the nature of the data structure. An excellent example of this is the update step of the clustering according to neighbour types, which requires four nested for loops. Each cluster must have its mean calculated. The mean relies upon connections to every possible cluster, so each of these in turn must be accessed. Each of the vertices in the cluster being assessed may be tied to a node in the second cluster, so each edge of each vertex must be checked to see if a connection exists.

In these circumstances code quickly becomes obfuscated and tricky to follow. This is especially prevalent where further nesting occurs due to conditional statements. The method for the above operation, `calculateNeighbourhoodMeans()` is shown below, but for brevity only snippets are presented and conditional statements have been removed:

```
private ArrayList<ArrayList<Double>> calculateNeighbourhoodMeans()
{
    ArrayList<ArrayList<Double>> neighbourhoodMeans
        = new ArrayList<ArrayList<Double>>();

    // calculate the means
    // based on the existing clusters and their neighbours
    for (int clusterIndex = 0; clusterIndex < noOfClusters; clusterIndex++)
    {
        ArrayList<Double> adjacencyMeans = new ArrayList<Double>();

        ...

        // for each cluster, loop through each possible adjacent cluster
        for (int adjIndex = 0; adjIndex < noOfClusters; adjIndex++)
        {
            ArrayList<Vertex> currentAdjacents = clusters.get(adjIndex);
            Double adjCount = 0.0;

            // for each adjacent cluster
            // loop through all the vertices of the current cluster
            for (int vertexIndex = 0; vertexIndex < currentCluster.size();
                vertexIndex++)
            {
                ...

                // for each vertex in the current cluster
                // loop through the edges that are connected to it
                for (int edgeIndex = 0; edgeIndex < currentEdges.length;
                    edgeIndex++)
                {
                    ...

                    // if the node on the other end of edge is in adjacent cluster
                    // increment count of how many neighbours of that position

                    ...

                } // for
            } // for

            // average the count over the number of elements in the set

            ...

        } // for

        neighbourhoodMeans.add(adjacencyMeans);
    } // for

    return neighbourhoodMeans;
} // calculateNeighbourhoodMeans
```

The outline of the code is easy enough to track and understand after some thought, but when many calculations are involved, for example in the assignment step (that takes place immediately after the above method), it is easy for confusion to set in.

5. Results

The end product of this endeavour was a Java project that clusters the points of a network around the notion of automorphic equivalence in two ways – first by finding orbits, second by k -means clustering – and compares the clusterings via the use of two measures. This chapter explores how to use the program that has been created.

5.1. Required Interaction

As the project was about developing an algorithm as opposed to creating a tool for an end-user, there is little in the way of human interaction, let alone a front-end interface. Assuming the project is ran in Eclipse, as it was developed, the user must first set the main class of the project to *AnalysePositions*. This is done by clicking Run > Run Configurations... in the menu bar and entering the class name in the “Main class:” text both under the “Main” tab. *AnalysePositions* handles the arguments entered by the user and generates an instance of a ‘position analysis’.

All the user needs to do then is to enter the program arguments, of which there are two. First is file path of the graph to be analysed, in GML format. Second is the number of clusters to be used in the k -means clustering. In Eclipse these can be entered under the “Arguments” tab of the run configurations. This is illustrated using the graph of A - 1: “Les Misérables” network data. with 30 clusters is shown in Figure 15 below:

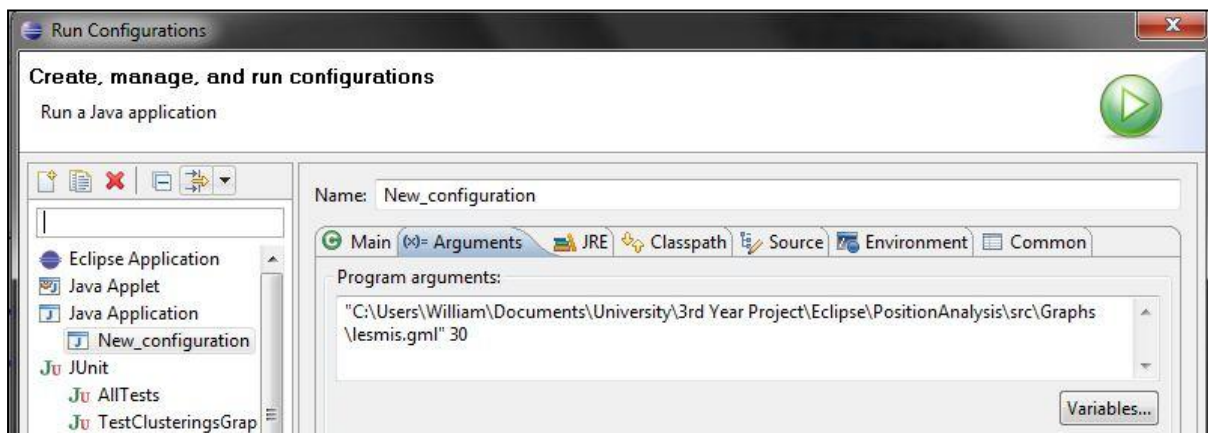


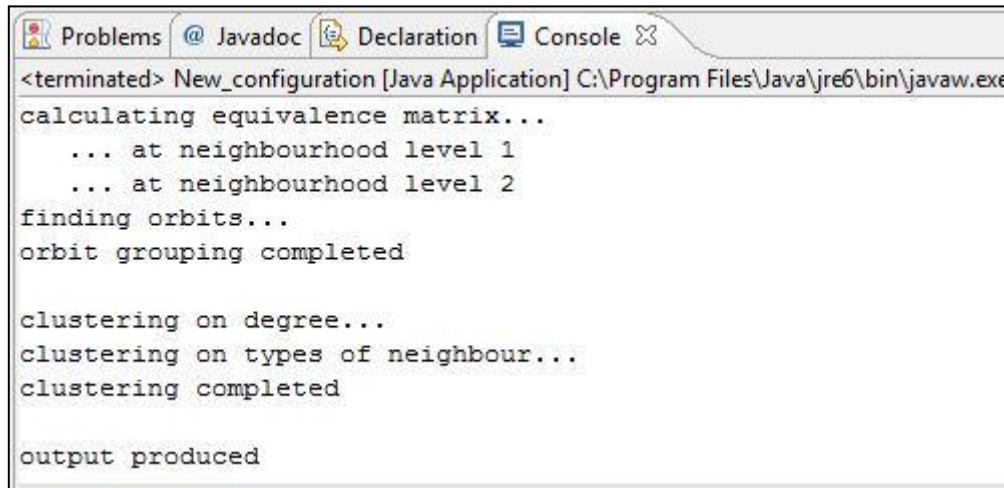
Figure 15: Run configurations of the project.

Pressing the “Run” button on the same screen starts the program.

Currently there is no argument to deal with the neighbourhood level used in the orbit-finding algorithm. For the time being it has been hard-coded as 2 into the method call of the class that uses it, because a second-order neighbourhood located the orbits of all the graphs used for testing but one (C - 16: Test network 16.) notably faster than higher-order neighbourhoods. Changing the program parameters to accommodate a specified neighbourhood level should be trouble free.

5.2. Interpretation of Output

Output of the program has to be interpreted by eye so care has been taken to ensure output is comprehensible. As the program runs it updates the console at the principal stages, allowing the user to view approximate progress. A completed run will display Figure 16 in the console.



```

<terminated> New_configuration [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
calculating equivalence matrix...
    ... at neighbourhood level 1
    ... at neighbourhood level 2
finding orbits...
orbit grouping completed

clustering on degree...
clustering on types of neighbour...
clustering completed

output produced
    
```

Figure 16: Console display of a completed program run.

The first chunk of text refers to the orbit-finding algorithm, the second to k -means clustering, and the third to the files produced by the program containing the results. These files will be created in the folder where the input graph is located, of which there will be four, as exemplified in Figure 17 below:

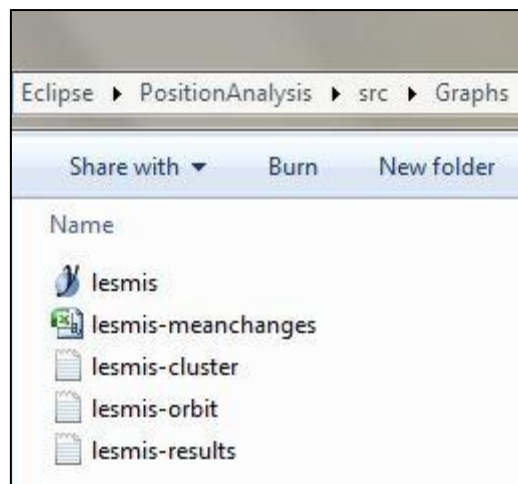


Figure 17: Files created by a run of the program.

The CSV file suffixed “-meanchanges” contains data recording the change in mean of the clusters against iterations and is used to confirm stabilisation. The TXT files suffixed “-cluster” and “-orbit” represent the node groupings found by the respective algorithms, and are essentially lists of sets. The main file of interest is that suffixed “-results”, and is split into five distinct sections. The first lists the orbits of the graph and the second lists the clusters as produced by the statistical approach. The third section is an equivalence matrix of the orbits similar to that in Table 6. Section four gives the

orbit-cluster equivalence measure along with details of how the figure was calculated; section five does the same but with point-pair equivalence. The following screenshot illustrates the appearance of this file, and uses the network in Figure 12 that produced Table 6.

```

Orbit 001: [a, e]
Orbit 002: [b, d]
Orbit 003: [c]

Cluster 001: []
Cluster 002: []
Cluster 003: [a, e]
Cluster 004: []
Cluster 005: [b, c, d]

Orbit Equiv | 001 | 002 | 003 |
-----|-----|-----|-----|
001 | 1.000 | 0.591 | 0.455 |
002 | 0.591 | 1.000 | 0.761 |
003 | 0.455 | 0.761 | 1.000 |

equivalence between the set of orbits and the set of clusters: 0.8333333333333334
[coverage/accuracy bias: 0.5]

=====
orbit: 1 | most similar cluster: 3 | equivalence: 1.0
- orbit: [a, e]
- cluster: [a, e]
# coverage = matched elements / size of orbit = 2 / 2 = 1.0
# accuracy = 1 - (cluster elements not in orbit / size of cluster) = 1 - (0 / 2) = 1.0
# orbit-cluster equivalence = (coverage * bias) + (accuracy * (1 - bias))
= (1.0 * 0.5) + (1.0 * (1 - 0.5)) = 1.0

=====
orbit: 2 | most similar cluster: 5 | equivalence: 0.8333333333333334
- orbit: [b, d]
- cluster: [b, c, d]
# coverage = matched elements / size of orbit = 2 / 2 = 1.0
# accuracy = 1 - (cluster elements not in orbit / size of cluster) = 1 - (1 / 3) = 0.6666666666666667
# orbit-cluster equivalence = (coverage * bias) + (accuracy * (1 - bias))
= (1.0 * 0.5) + (0.6666666666666667 * (1 - 0.5)) = 0.8333333333333334

=====
orbit: 3 | most similar cluster: 5 | equivalence: 0.6666666666666667
- orbit: [c]
- cluster: [b, c, d]
# coverage = matched elements / size of orbit = 1 / 1 = 1.0
# accuracy = 1 - (cluster elements not in orbit / size of cluster) = 1 - (2 / 3) = 0.3333333333333333
# orbit-cluster equivalence = (coverage * bias) + (accuracy * (1 - bias))
= (1.0 * 0.5) + (0.3333333333333333 * (1 - 0.5)) = 0.6666666666666667

average orbit-cluster equivalence:
(1.000 + 0.833 + 0.667) / 3 = 0.8333333333333334

=====
similarity between the two clusterings: 0.8

=====
similarly clustered pairs:
* (a-b)
* (a-c)
* (a-d)
* (a-e)
* (b-d)
* (b-e)
* (c-e)
* (d-e)
+ 8 pairs

=====
dissimilarly clustered pairs:
* (b-c)
* (c-d)
+ 2 pairs

=====
similarity between the two clusterings:
8 / 10 = 0.8
=====

```

Figure 18: Example program results file.

It can be seen through observation that the orbits of Figure 12 are $\{a, e\}, \{b, d\}, \{c\}$ whereas the statistical technique produced the clusters $\{a, e\}, \{b, c, d\}$. The equivalence matrix underneath shows that orbits $\{b, d\}$ and $\{c\}$ are the most similar pair of orbits with a similarity of 0.761. Then, for the comparison of the two clusterings, the orbit-cluster equivalences scores 0.834 whilst the point-pair equivalence scores 0.8. Both of these measures indicate that the clusterings are reasonably similar, but not identical.

5.3. Performance

The performance of the program is affected mostly by the orbit-finding algorithm; the main factors, then, are the size of the network and the neighbourhood level inspected. Applying a third-order neighbourhood to moderately-sized networks such as A – 1: *Les Misérables* takes over five minutes on a fast machine – noticeably longer than just first- and second-order neighbourhoods. There are a couple of conceivable reasons for this. One is that the primary operations, namely generating neighbourhoods and calculating vectors, are much costlier than first anticipated. Therefore, despite both algorithms having a quadratic worst case time complexity, it is the orbit-finder that contains the computationally expensive operations. Furthermore, it does not require a high neighbourhood level to reach the worst case scenario. Most nodes of A – 1 are connected after the third-order. Only a handful are connected through a higher order. These expensive operations on such an easily achievable worst case scenario may be accountable for undesirable performance.

Another possibility is that the implementation of the algorithm is inefficient in places. There may be unnecessary calculations where a form of memoization (????) could have been used, but this is uncertain. Due to inexperience of the subject matter during coding, this should not be ruled as a cause, or aggravator, of poor performance.

The other areas of the program show no major performance issues, only reasonable extra computation time for larger networks.

6. Conclusions

This chapter summarises the achievements of the project with regards to the original requirements, before going on to suggest further activity that could extend or enhance the work that has been done.

6.1. Achievement of Objectives

The three milestones set out in section 1.4 have been met. Milestone 1, implementation of the orbit-finding algorithm, was done early in the project. There were some difficulties, as highlighted in sections 4.3.1 and 4.3.2, but these were alleviated thanks to well-documented of the algorithm and the concepts behind it. In the end it was the coding more than fundamental understanding that caused setbacks. Milestone 2, implementation of a statistics-based approach, proved to be troublesome in both aspects. k -means clustering was unfamiliar and envisioning a single measure of distance was a struggle, despite the underlying concept being uncomplicated. This, in combination with deep nesting of for loops, took time to comprehend, but eventually the milestone was met with success. Milestone 3, a measure of similarity, was unexpectedly tricky to meet. A lot of deliberation took place over what was required from a similarity measure in relation to this project, and a range of ideas were experimented with. The difficulty was that treating the two clusterings as arbitrary clusterings of the same data didn't quite make sense. Indeed, the orbit-finding method is a very precise, mathematical one, independent of initialisation choices such as the number of clusters and mean values. Handling the orbits as something of a target prompted challenge and innovation. It is felt this milestone has been met relatively successfully, and that the devised metric is a useful one.

6.2. Changes to Original Plan

In Appendix E: Project Plans there are two Gantt charts depicting project plans. The first plan was made before the project had started, for a somewhat different proposal. Originally, the project was going to be about discovering community structures in graphs. The plan was drawn up and the first two weeks were spent researching literature. Then it was suggested that positional analysis could be an alternative, more interesting project. After another couple of weeks of researching material on positional analysis, it was chosen as the area to progress in. The project plan was revised shortly after to correspond with the new path taken. Since the project was split into very distinct chunks, everything beyond "Reading Week" advanced closely with the plan. The only notable deviation from this is that "Clustering 2" implementation – the k -means clustering – and the design of "Clustering Measures" required more time than the designated three weeks. This forced the "Clustering Measures" implementation to be carried out in a relatively short space of time.

6.3. Further Activity

The project satisfied the basic milestones required for a functional end result, but there are some marked activities that could extend or improve what has been done here. As stated earlier in section 5.1, the neighbourhood level used in the orbit-finder could be set as a program argument. This would perhaps need some mediation as, for larger networks, even low values cause the program the run slowly, and is why this has not been implemented here. The most significant

enhancement to the project would probably be an auto-selection of k , the number of clusters in the statistical clustering. This would remove the need for human estimation or trial and error, and will likely produce a better clustering the first time round. Not only would this be a noteworthy improvement, it is also plausible. To do so, one must minimise $E_q + \alpha Dk \log n$, where E_q is the quantisation error as described in section 3.1, n is the number of data points, D is the dimension of the data, and α is a constant (Shapiro, 2009). It is a straightforward formula that would have been attempted had the time been available.

One final thing to note: there is rationale to modify the k -means clustering algorithm proposed in this report. Currently, it is a naïve approach, assuming actors are similar if they have a similar degree. This is reasonable, but raises the question: is an actor with no neighbours similar to an actor with one neighbour? Or are these isolated nodes as dissimilar to highly connected nodes as they are to singularly connected nodes? This issue is context sensitive; a mother with no child is clearly not a mother, and arguably incomparable to genuine mothers. Yet one can be a neurosurgeon before having operated on any patients³. Perhaps there are situations where isolated actors are more akin to those with hundreds of connections than those with one or two. Modifications to deal with such things would require an entire new level of human discretion and interaction, bringing with it an array of new difficulties. It would be an interesting extension, but how much overall effect this would have is unclear, though it isn't anticipated to make a great difference.

³ This may not be true, but hopefully it brings the point across.

References

- Borgatti, S. P., & Everett, M. G. (1992). Notions of Position in Social Network Analysis. *Sociological Methodology*, 22, 1-35.
- Borgatti, S. P., & Everett, M. G. (1989). The Class of All Regular Equivalences: Algebraic Structure and Computation. *Social Networks*, 11, 65-88.
- Borgatti, S. P., & Everett, M. G. (1993). Two Algorithms for Computing Regular Equivalence. *Social Networks*, 361-376.
- Breiger, R. L., & Pattison, P. E. (1986). Social Networks. *Cumulated Social Roles: The Duality of Persons and their Algebras*, 8, 215-256.
- Breiger, R. L., Boorman, S., & Arabie, P. (1975). An Algorithm for Clustering Relational Data with Applications to Social Network Analysis. *Journal of Mathematical Psychology*, 12, 329-383.
- Burt, R. S. (1978). Cohesion Versus Structural Equivalence as a Basis for Network Subgroups. *Sociological Methods and Research*, 7, 189-212.
- Burt, R. S. (1976). Positions in Networks. *Social Forces*, 55, 93-122.
- Caldeira, G. A. (1988). Legal Precedent: Structures of Communication Between State Supreme Courts. *Social Networks*, 10, 29-55.
- Eclipse Downloads. (n.d.). Retrieved from Eclipse: <http://www.eclipse.org/downloads/>
- Everett, M. G. (1985). Role Similarity and Complexity in Social Networks. *Social Networks*, 7, 353-359.
- Everett, M. G., & Borgatti, S. P. (1990). Any Colour You Like It. *Connections*, 13, 19-22.
- Everett, M. G., & Borgatti, S. P. (1988). Calculating Role Similarities: An Algorithm that Helps Determine the Orbits of a Graph. *Social Networks*, 10, 77-91.
- Everett, M. G., & Borgatti, S. P. (1991). Role Colouring a Graph. *Mathematical Social Sciences*, 21, 183-188.
- Everett, M. G., Boyd, J. P., & Borgatti, S. P. (1990). Ego-Centered and Local Roles: A Graph-Theoretic Approach. *Journal of Mathematical Sociology*, 15, 163-72.
- Freeman, L. C. (1978). Centrality in Social Networks: Conceptual Clarification. *Social Networks*, 1, 215-239.
- Friedkin, N. E. (1984). Structural Cohesion and Equivalence Explanations of Social Homogeneity. *Sociological Methods and Research*, 12, 235-261.
- Galaskiewicz, J., & Krohn, K. R. (1984). Positions, Roles, and Dependencies in a Community Integrated System. *Sociological Quarterly*, 25, 527-550.

- Hofstadter, D. R. (1985). *Metamagical Themas: Questing for the Essence of Mind and Pattern*. New York: Basic Books.
- Hummell, H., & Sodeur, W. (1987). Strukturbeschreibung von Positionen in sozialen Beziehungsnetzen. *Methoden der Netzwerkanalyse* , 1-21.
- Knuth, D. E. (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*. Reading, MA: Addison-Wesley.
- Lloyd, S. P. (1982). Least Squares Quantization in PCM. *IEEE Transactions on Information Theory* , 28, 129-137.
- Lorrain, F. P., & White, H. C. (1971). Structural Equivalence of Individuals in Networks. *Journal of Mathematical Sociology* , 1, 49-80.
- Lusseau, D., Schneider, K., Boisseau, O. J., Haase, P., Slooten, E., & Dawson, S. M. (2003). The bottlenose dolphin community of Doubtful Sound features a large proportion of long-lasting associations. Can geographic isolation explain this unique trait? *Behavioural Ecology and Sociobiology* , 54, 396-405.
- Manning, C. D., Raghavan, P., & Schütze, H. (2009). *Introduction to Information Retrieval*. Cambridge University Press.
- Naveh, B. (n.d.). *JGraphT*. Retrieved November 13, 2009, from JGraphT: <http://jgrapht.sourceforge.net/>
- Newman, M. (2009, June 29). *Network data*. Retrieved November 30, 2009, from Mark Newman: <http://www-personal.umich.edu/~mejn/netdata/>
- Rand, W. M. (1971). Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association* , 66, 846-850.
- Shapiro, J. L. (2009). Lecture 4.1 - Unsupervised Learning I: Competitive Learning. *CS6483* .
- The GML File Format*. (1997, July 20). Retrieved November 13, 2009, from Graphlet: <http://www.infosun.fim.uni-passau.de/Graphlet/GML/>
- The GraphML File Format*. (2007, April 5). Retrieved November 13, 2009, from The GraphML File Format: <http://graphml.graphdrawing.org/>
- Wasserman, S., & Faust, K. (1994). *Social Network Analysis: Methods and Applications*. New York: Cambridge University Press.
- White, D. R. (1982). Measures of global equivalence in social networks. *Unpublished manuscript* .
- White, D. R. (1984). REGGE: a REGular Graph Equivalence algorithm for computing role distances prior to blockmodeling. *Unpublished manuscript* .
- White, D. R. (1980). Structural equivalences in social networks: concepts and measurement of role structures. *Unpublished manuscript* .

White, D. R., & Reitz, K. (1983). Graph and semigroup homomorphisms on semigroups of relations. *Social Networks* , 5, 193-234.

Winship, C. (1988). Thoughts about Roles and Relations: An Old Document Revisited. *Social Networks* , 10, 209-231.

Winship, C., & Mandel, M. J. (1983). Roles and Positions: A Critique and Extension of the Blockmodeling Approach. *Sociological Methodology*, 1983-84 , 314-344.

yEd Graph Editor. (n.d.). Retrieved November 13, 2009, from yWorks:
http://www.yworks.com/en/products_yed_about.html

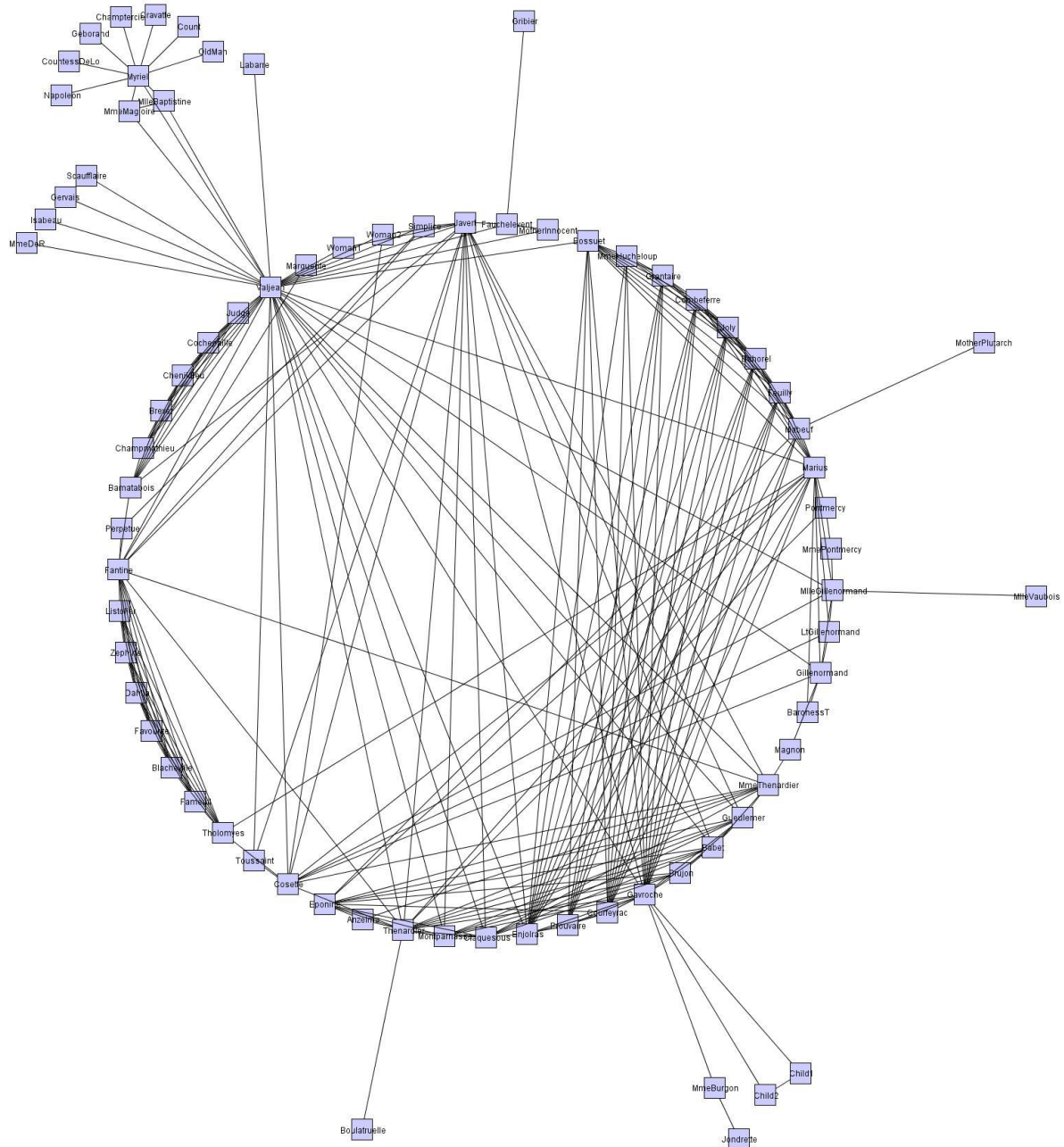
Zachary, W. W. (1977). An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research* , 33, 452-473.

Appendix A: Data Sets

All researched network data used in this project was retrieved from the website of Mark Newman. The network depictions have been constructed by the *yEd Graph Editor*.

A – 1: Les Misérables (Knuth, 1993)

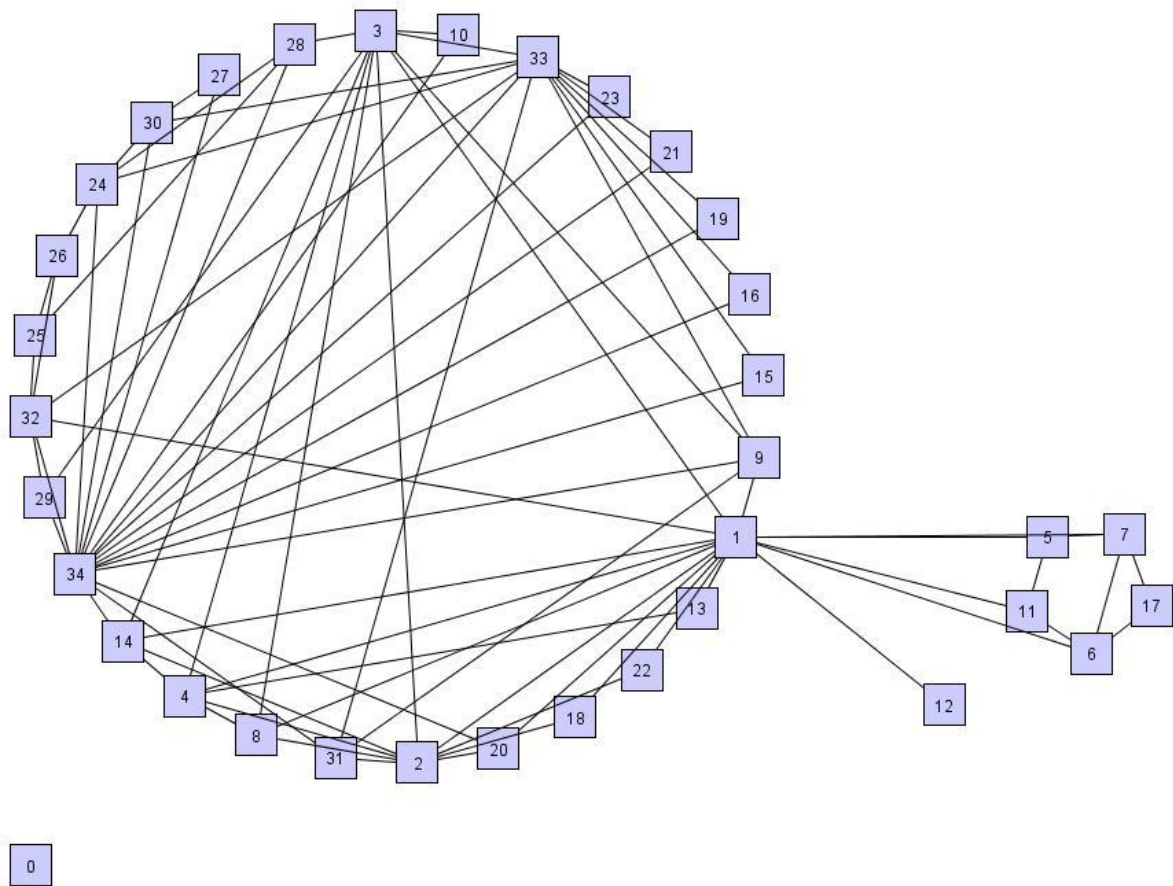
A coappearance network of characters in the novel *Les Misérables*.



A - 1: “Les Misérables” network data.

A – 2: Zachary’s Karate Club (Zachary, 1977)

A social network of friendships between 34 members of a karate club at a university in the United States in the 1970s.



A - 2: “Karate Club” network data.

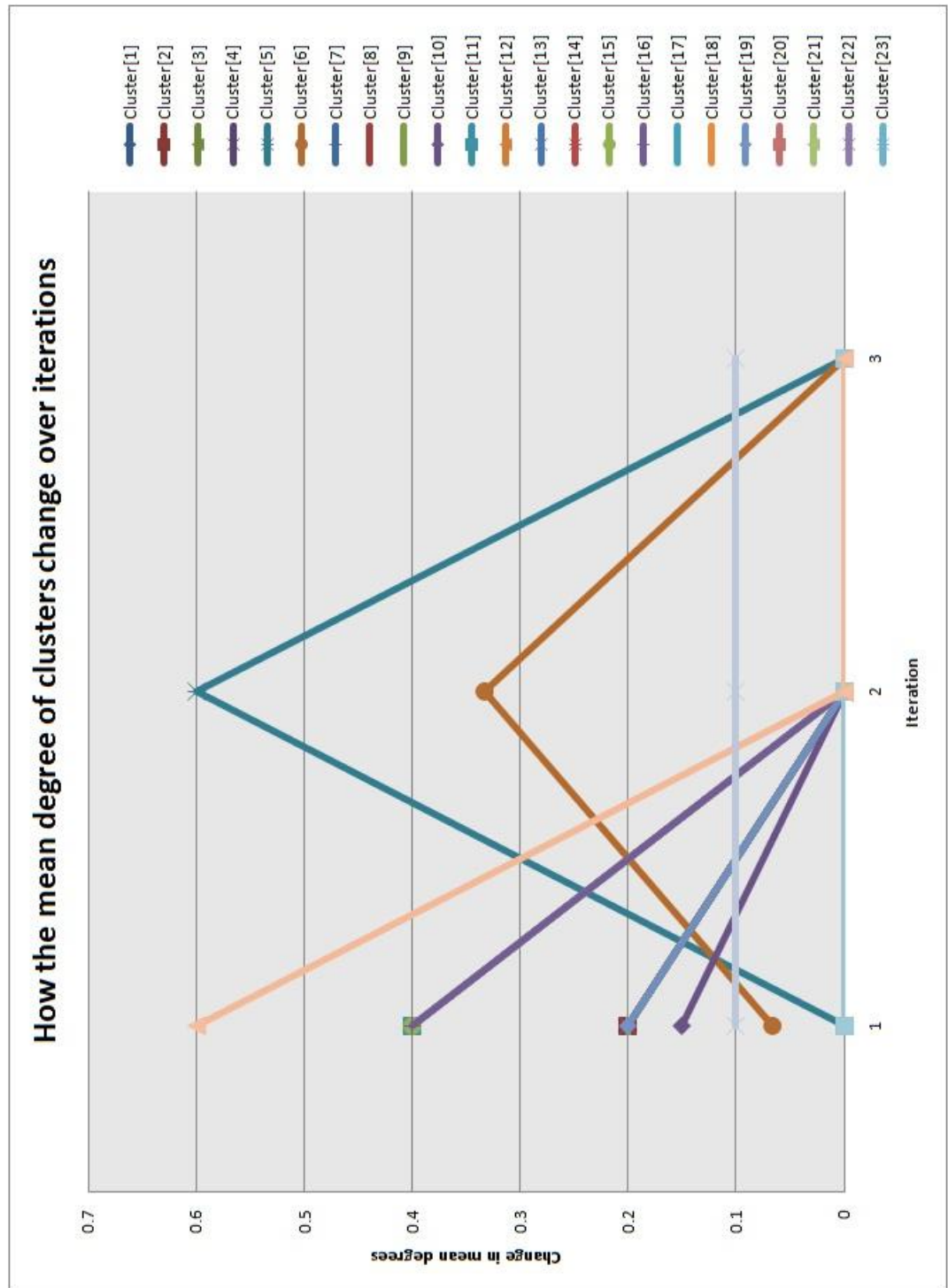
A social network of frequent associations between 62 dolphins in a community living off Doubtful Sound, New Zealand.



Appendix B: Stabilisation Charts

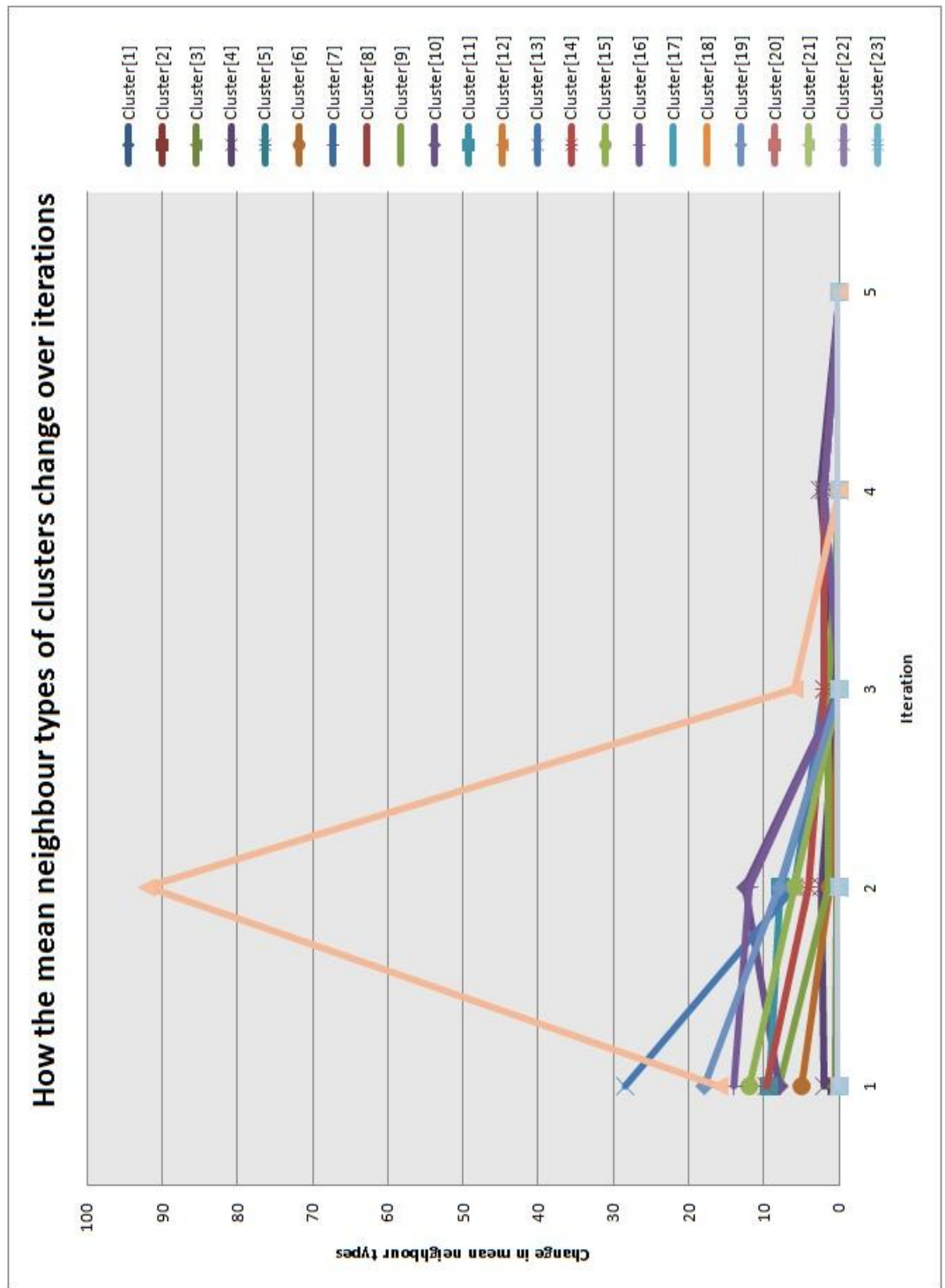
The charts illustrate how the data points of the networks converge over time during k -means clustering.

B – 1: “Les Misérables” Degree Stabilisation



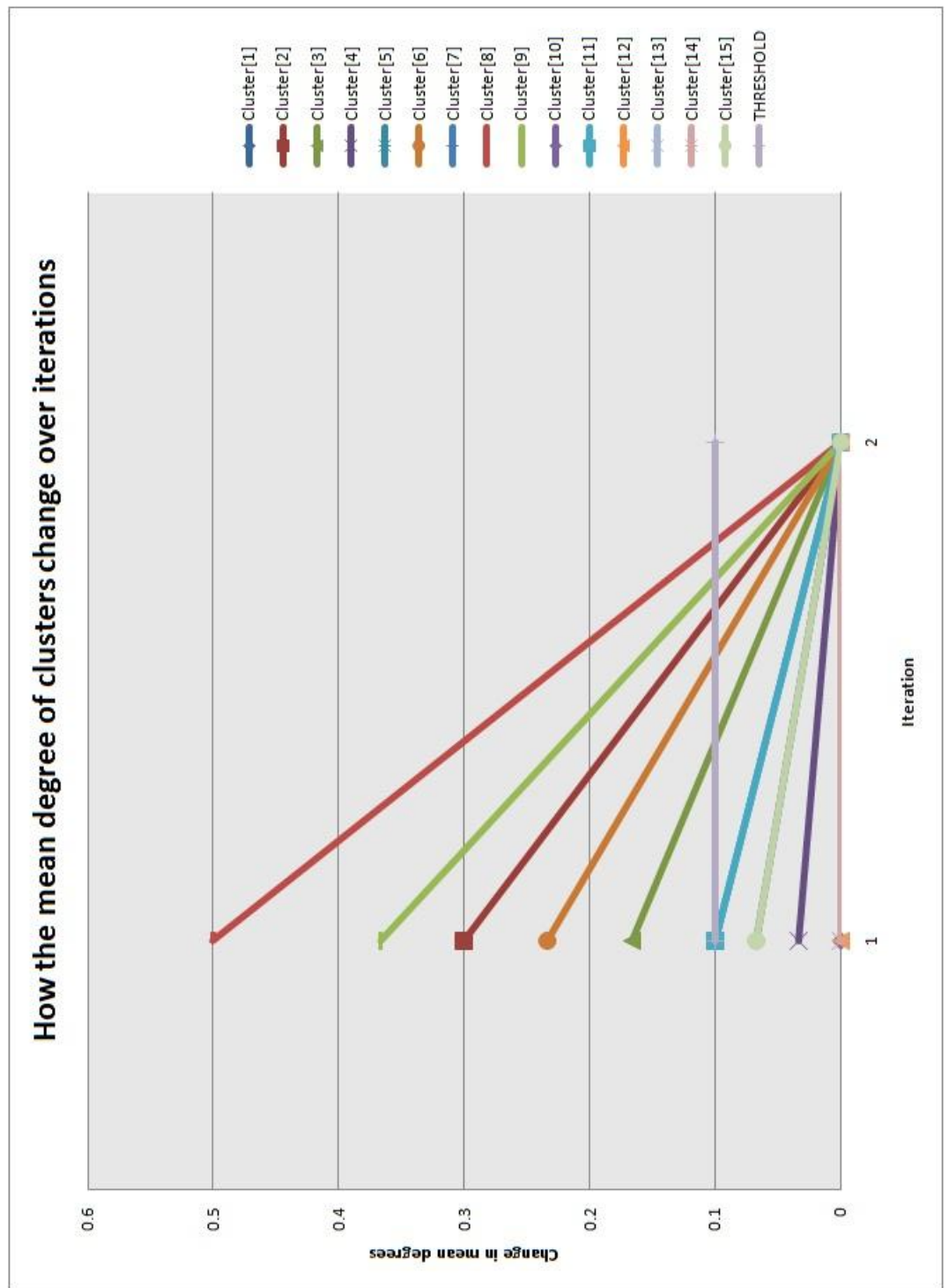
B - 1: Degree stabilisation of network in A - 1.

B – 2: “Les Misérables” Neighbour Type Stabilisation



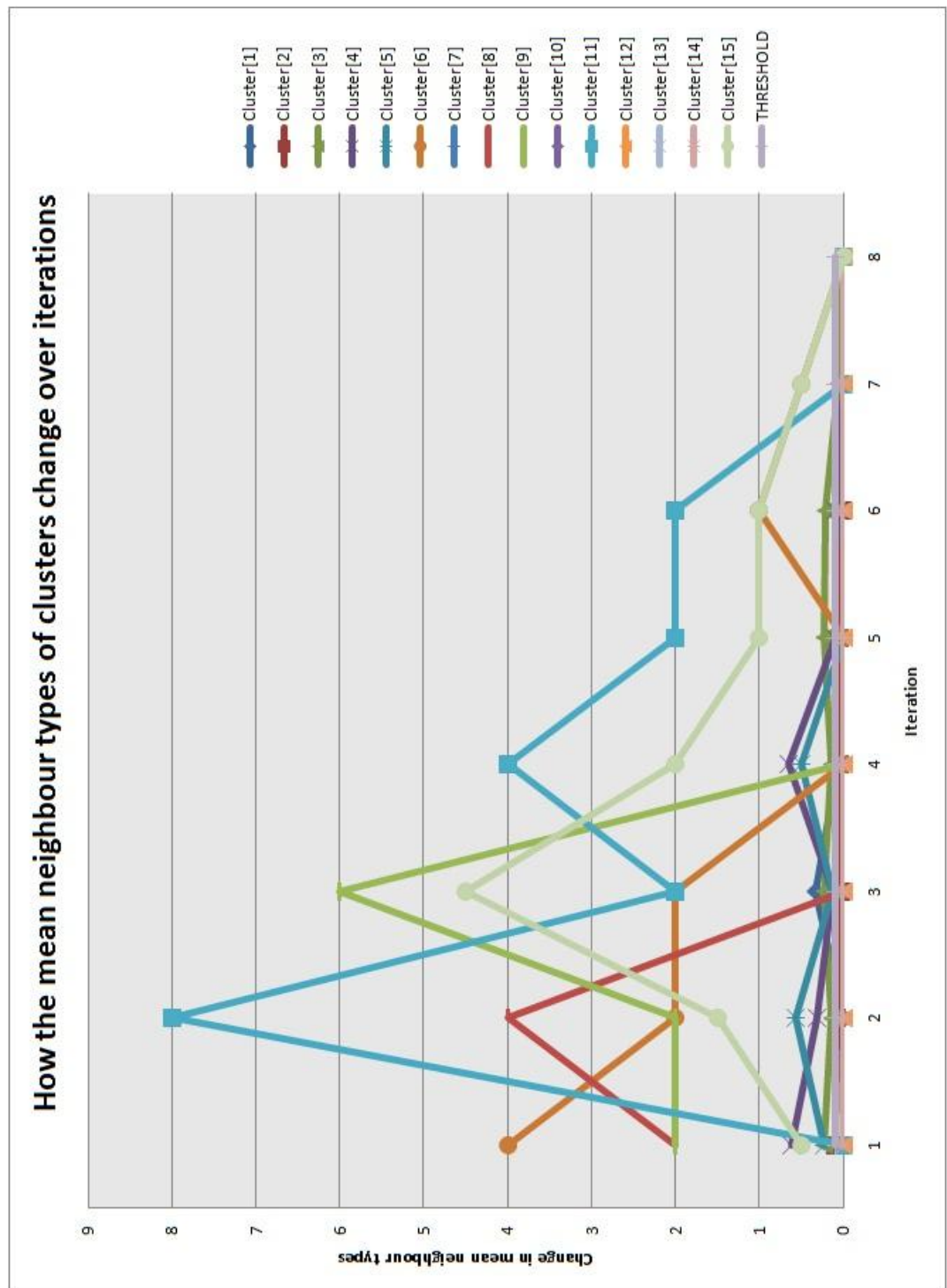
B - 2: Neighbour type stabilisation of network in A - 1.

B – 3: “Karate Club” Degree Stabilisation



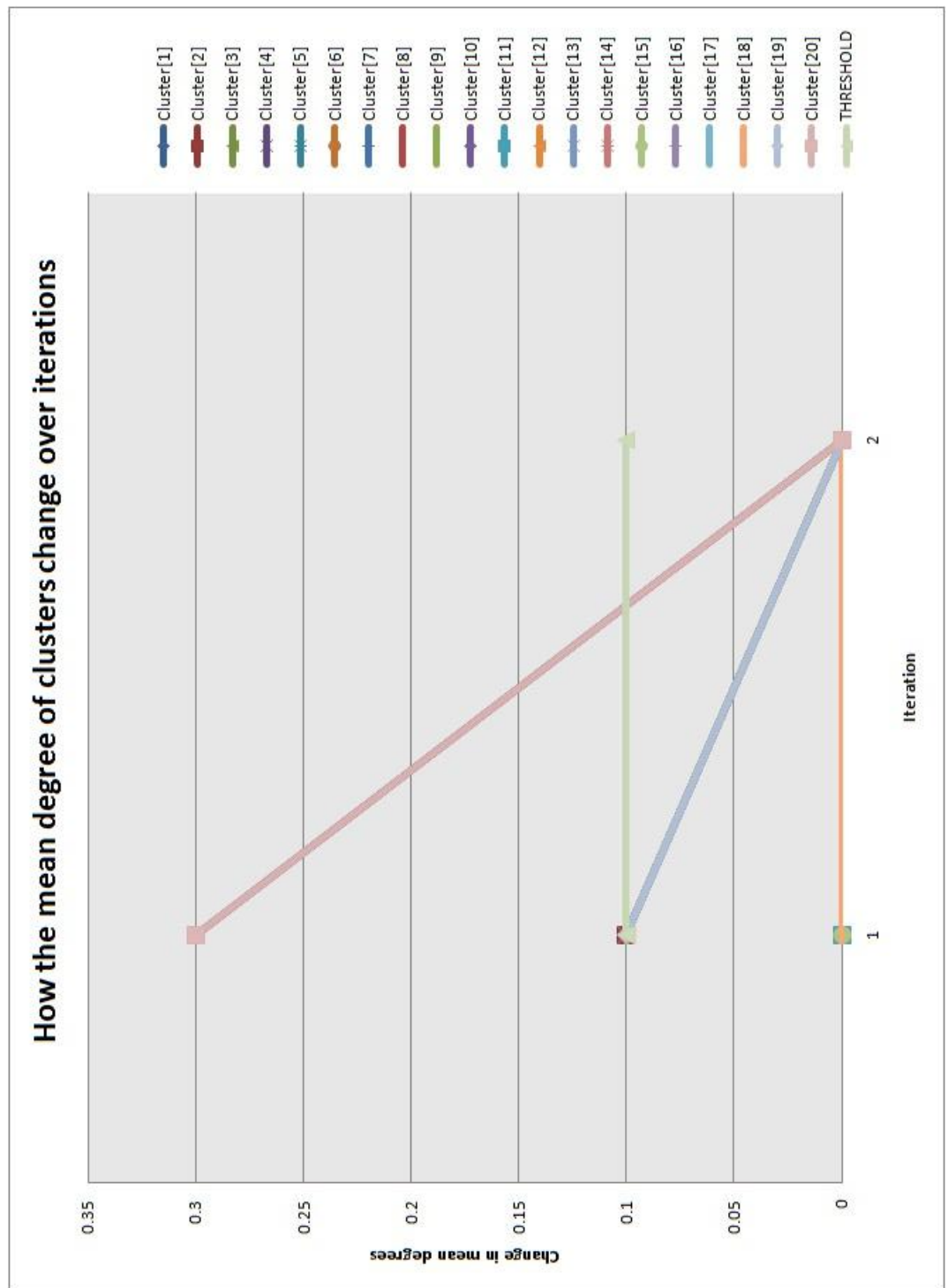
B - 3: Degree stabilisation of network in A - 2.

B – 4: “Karate Club” Neighbour Type Stabilisation



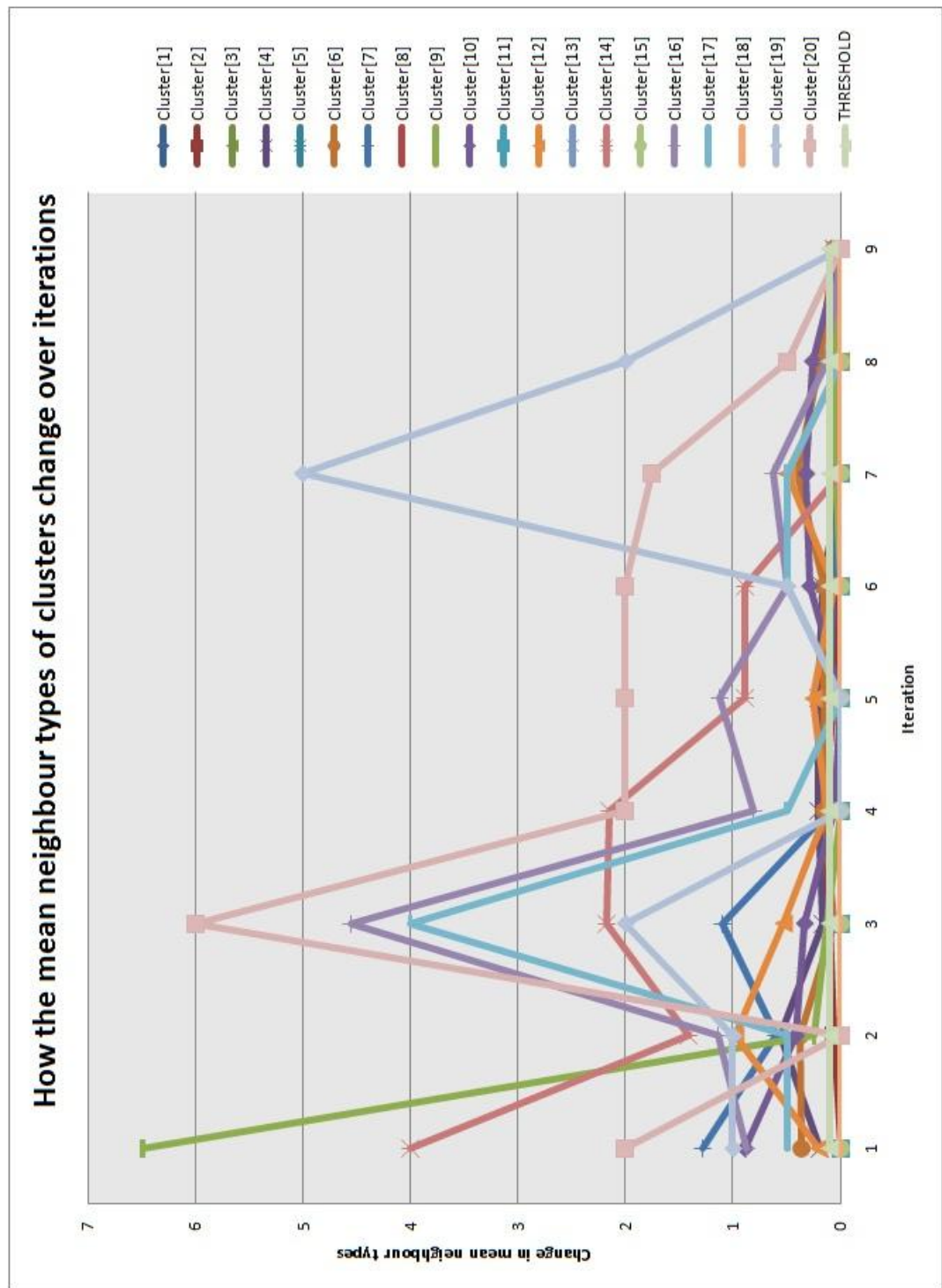
B - 4: Neighbour type stabilisation of network in A - 2.

B – 5: “Dolphins” Degree Stabilisation



B - 5: Degree stabilisation of network in A - 3.

B – 6: “Dolphins” Neighbour Type Stabilisation



B - 6: Neighbour type stabilisation of network in A - 3.

Appendix C: Clustering Test Data

Here are listed all the devised networks used to test the two clustering algorithms. Note that where k is greater than the number of clusters, the clusters not shown are empty and disregarded.

C – 1: Test Graph 1 (single node, isolated)



C - 1: Test network 1.

Orbits: $\{A\}$

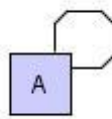
$k = 0$

Exception – there must be more than zero clusters

$k = 10$

Cluster 1: $\{A\}$

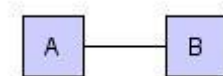
C – 2: Test Graph 2 (single node, reflexive loop)



C - 2: Test network 2.

Exception – reflexive loops are not handled by these algorithms

C – 3: Test Graph 3 (two nodes, connecting edge)



C - 3: Test network 3.

Orbits: $\{A, B\}$

$k = 2$

Cluster 2: $\{A, B\}$

$k = 10$

Cluster 10: $\{A, B\}$

C - 4: Test Graph 4 (two nodes, no edge)



C - 4: Test network 4.

Orbits: $\{A, B\}$

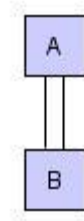
$k = 2$

Cluster 1: $\{A, B\}$

$k = 10$

Cluster 1: $\{A, B\}$

C - 5: Test Graph 5 (two nodes, two identical edges)



C - 5: Test network 5.

Orbits: $\{A, B\}$

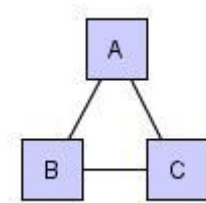
$k = 2$

Cluster 2: $\{A, B\}$

$k = 10$

Cluster 10: $\{A, B\}$

C - 6: Test Graph 6 (three nodes, single orbit)



C - 6: Test network 6.

Orbits: $\{A, B, C\}$

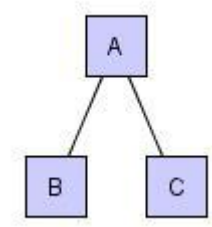
$k = 2$

Cluster 2: $\{A, B, C\}$

$k = 10$

Cluster 10: $\{A, B, C\}$

C - 7: Test Graph 7 (three nodes, two orbits)



C - 7: Test network 7.

Orbits: $\{A\}, \{B, C\}$

$k = 2$

Cluster 1: $\{B, C\}$

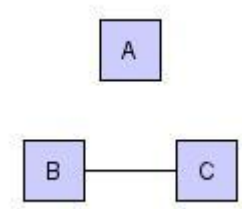
Cluster 2: $\{A\}$

$k = 10$

Cluster 5: $\{B, C\}$

Cluster 10: $\{A\}$

C - 8: Test Graph 8 (three nodes, one isolated)



C - 8: Test network 8.

Orbits: $\{A\}, \{B, C\}$

$k = 2$

Cluster 1: $\{A\}$

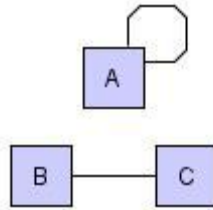
Cluster 2: $\{B, C\}$

$k = 10$

Cluster 1: $\{A\}$

Cluster 10: $\{B, C\}$

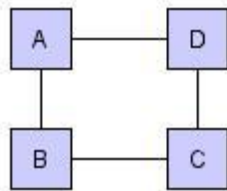
C – 9: Test Graph 9 (three nodes, reflexive loop)



C - 9: Test network 9.

Exception – reflexive loops are not handled by these algorithms

C – 10: Test Graph 10 (four nodes, single orbit)



C - 10: Test network 10.

Orbits: $\{A, B, C, D\}$

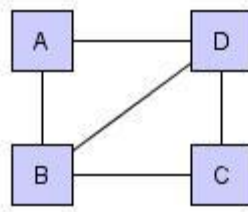
$k = 2$

Cluster 2: $\{A, B, C, D\}$

$k = 10$

Cluster 10: $\{A, B, C, D\}$

C – 11: Test Graph 11 (four nodes, two orbits, high connectivity)



C - 11: Test network 11.

Orbits: $\{A, C\}, \{B, D\}$

$k = 2$

Cluster 1: $\{A, C\}$

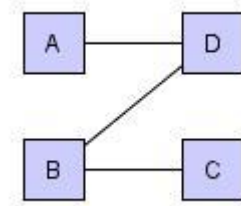
Cluster 2: $\{B, D\}$

$k = 10$

Cluster 7: $\{A, C\}$

Cluster 10: $\{B, D\}$

C - 12: Test Graph 12 (four nodes, two orbits, low connectivity)



C - 12: Test network 12.

Orbits: $\{A, C\}, \{B, D\}$

$k = 2$

Cluster 1: $\{A, C\}$

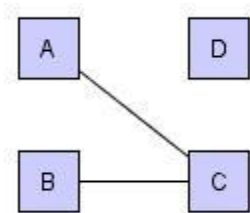
Cluster 2: $\{B, D\}$

$k = 10$

Cluster 5: $\{A, C\}$

Cluster 10: $\{B, D\}$

C - 13: Test Graph 13 (four nodes, one isolated, three orbits)



C - 13: Test network 13.

Orbits: $\{A, B\}, \{C\}, \{D\}$

$k = 2$

Cluster 1: $\{A, B, D\}$

Cluster 2: $\{C\}$

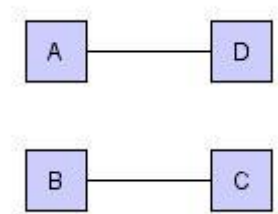
$k = 3$

Cluster 1: $\{D\}$
 Cluster 2: $\{A, B\}$
 Cluster 3: $\{C\}$

$k = 10$

Cluster 1: $\{D\}$
 Cluster 5: $\{A, B\}$
 Cluster 10: $\{C\}$

C - 14: Test Graph 14 (four nodes, two communities, single orbit)



C - 14: Test network 14.

Orbits: $\{A, B, C, D\}$

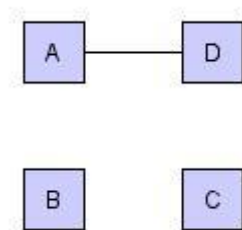
$k = 2$

Cluster 2: $\{A, B, C, D\}$

$k = 10$

Cluster 10: $\{A, B, C, D\}$

C - 15: Test Graph 15 (four nodes, two isolated, two orbits)



C - 15: Test network 15.

Orbits: $\{A, D\}, \{B, C\}$

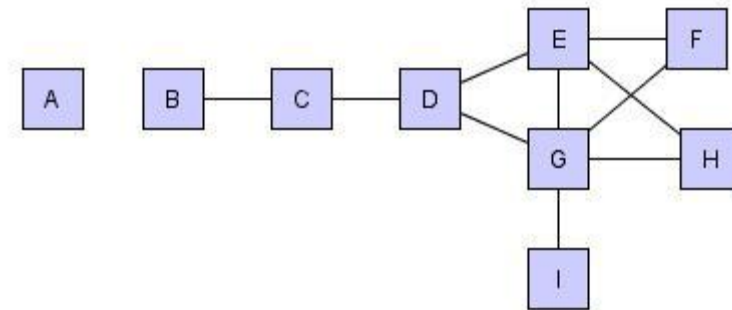
$k = 2$

Cluster 1: $\{B, C\}$
 Cluster 2: $\{A, D\}$

$k = 10$

Cluster 1: $\{B, C\}$
Cluster 10: $\{A, D\}$

C – 16: Test Graph 16 (nine nodes, one isolated, eight orbits)



C - 16: Test network 16.

Orbits: $\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F, H\}, \{G\}, \{I\}$

$k = 5$

Cluster 1: $\{A, B, I\}$
Cluster 2: $\{C, F, H\}$
Cluster 3: $\{D\}$
Cluster 4: $\{E\}$
Cluster 5: $\{G\}$

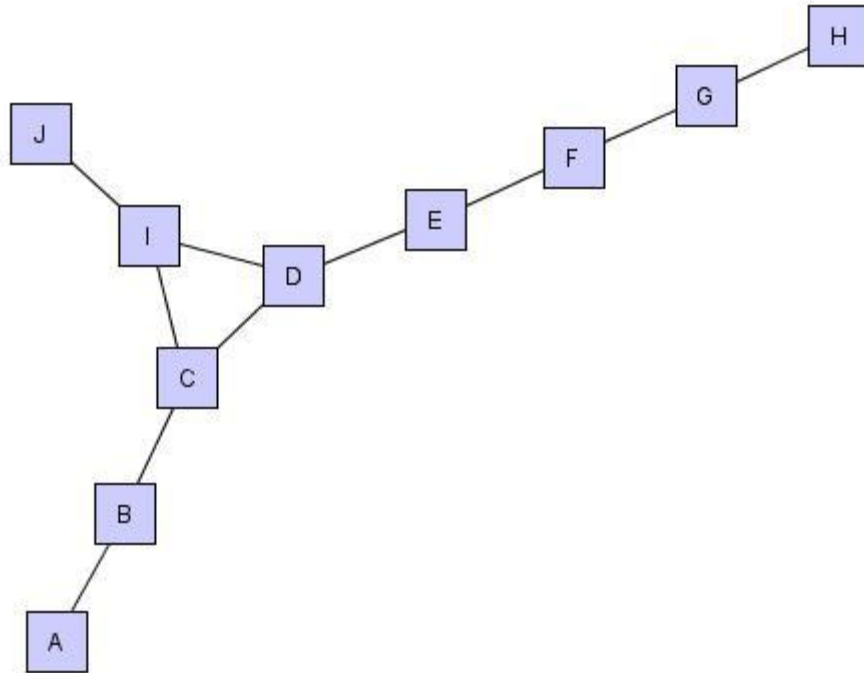
$k = 8$

Cluster 1: $\{A\}$
Cluster 2: $\{B, I\}$
Cluster 4: $\{C, F, H\}$
Cluster 5: $\{D\}$
Cluster 7: $\{E\}$
Cluster 8: $\{G\}$

$k = 10$

Cluster 1: $\{A\}$
Cluster 2: $\{B, I\}$
Cluster 4: $\{C, F, H\}$
Cluster 6: $\{D\}$
Cluster 8: $\{E\}$
Cluster 10: $\{G\}$

C - 17: Test Graph 17 (ten nodes, ten singleton orbits)



C - 17: Test network 17.

Orbits: $\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{G\}, \{H\}, \{I\}, \{J\}$

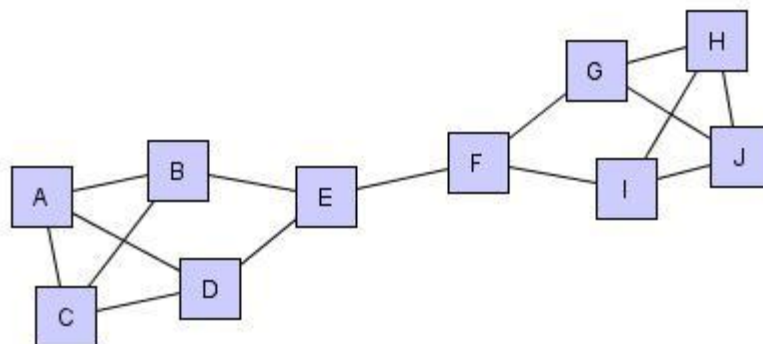
$k = 10$

Cluster 4: $\{A, H, J\}$

Cluster 7: $\{B, E, F, G\}$

Cluster 10: $\{C, I, D\}$

C - 18: Test Graph 18 (ten nodes, same degree, three orbits)



C - 18: Test network 18.

Orbits: $\{A, C, H, J\}, \{B, D, G, I\}, \{E, F\}$

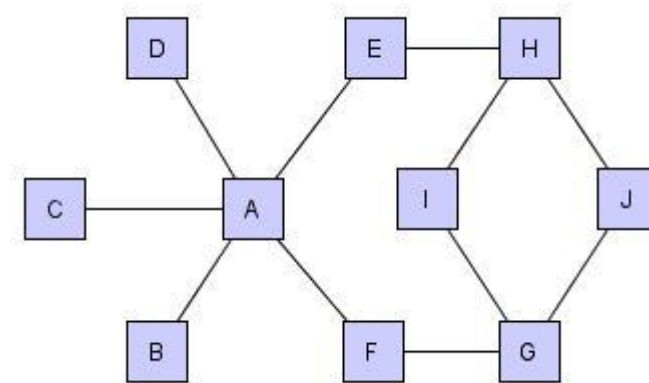
$k = 3$

Cluster 3: $\{A, B, C, D, E, F, G, H, I, J\}$

$k = 10$

Cluster 10: $\{A, B, C, D, E, F, G, H, I, J\}$

C – 19: Test Graph 19 (ten nodes, five orbits; assignment change in secondary clustering)



C - 19: Test network 19.

Orbits: $\{A\}, \{B, C, D\}, \{E, F\}, \{G, H\}, \{I, J\}$

$k = 3$: initial clustering

Cluster 1: $\{B, C, D\}$

Cluster 2: $\{E, F, G, H, I, J\}$

Cluster 3: $\{A\}$

$k = 3$: secondary clustering

Cluster 1: $\{B, C, D, E, F\}$

Cluster 2: $\{G, H, I, J\}$

Cluster 3: $\{A\}$

C – 20: Test Graph 20 (empty graph)

Exception – graphs must not be empty

Appendix D: Input File Formats

The following table shows how the graph in C - 6 is represented in different file formats.

D - 1: Comparison of two input file formats of network C – 6: Test Graph 6 (three nodes, single orbit).

GML	GraphML
<pre> graph [node [id 0 label "A"] node [id 1 label "B"] node [id 2 label "C"] edge [source 0 target 1 value 1] edge [source 1 target 2 value 1] edge [source 2 target 0 value 1]]</pre>	<pre> <?xml version="1.0" encoding="UTF-8"?> <graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd"> <graph id="G" edgedefault="undirected"> <node id="A"/> <node id="B"/> <node id="C"/> <edge source="A" target="B"/> <edge source="A" target="C"/> <edge source="B" target="C"/> </graph> </graphml></pre>

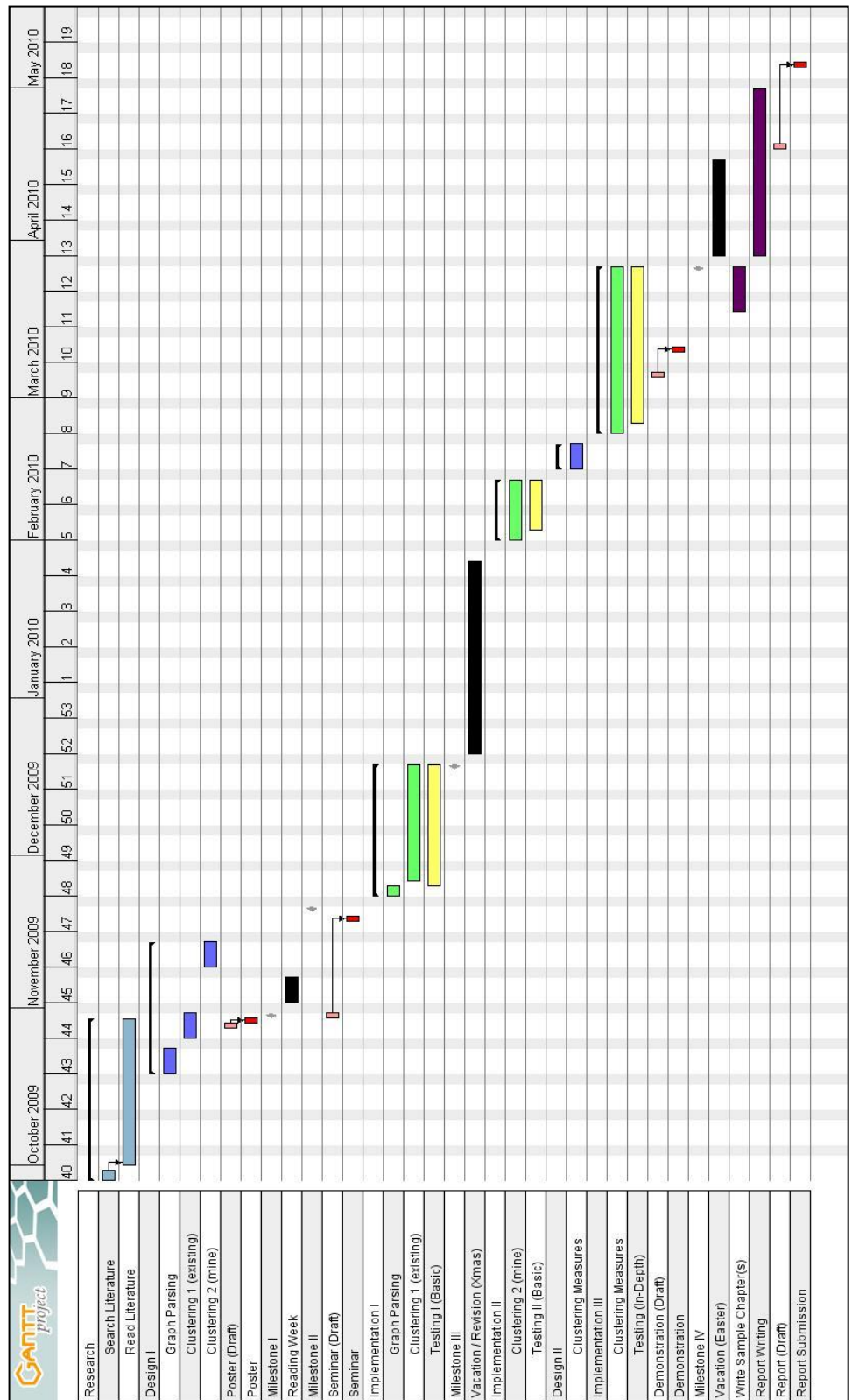
Appendix E: Project Plans

E – 1: Original Plan



E - 1: Original project plan.

E – 2: Revised Plan



E - 2: Revised Project Plan