DSNAT, Distributed Social Network Analysis Tool

Simon Burns Matthew Cranham James Kettle Isaac Lewis
James Michael

April 25, 2012

Abstract

Serious crimes such as violent protests and drug-trafficking are typically committed by networks of criminals. Individual members of these networks use social media such as Twitter and Facebook to communicate. Law enforcement agencies can use publicly available data from these services to target criminal networks in new ways. The West Midlands police are interested in harnessing this data to find the best ways to disrupt such networks. We plan to develop a tool that takes a sample of graph data representing a social network and uses the data to model influence and information propagation, and to provide strategic disruption advice.

Existing tools have performance issues, a problem we plan to evade by distributing the computation amongst a cluster of machines. This will provide a scalable solution to computationally expensive algorithms. We will present the tool as web service using a SOAP interface.

Contents

Chapter 1

Last Year Review

As our project is a continuation of work previously developed by another 4th year group, we must review the previous groups work and highlight any potential issues that could prove troublesome or helpful/useful in the completion of our system. The Social network analysis tool developed by the group from the previous year was a very well presented and constructed program. The main goals of this software was to allow a user to study and implement various algorithms across varying forms of social network. The interface was intended to handle extremely large amounts of local or remote data seamlessly. Focussing on the collapse of the Enron company, they modelled the communication interaction between colleagues at the company and conducted extensive research into the network model developed from the acquired data.

The Design of the social network analysis tool was split into 7 major sections which are the:

- Database
- Database Proxy
- Network model
- Algorithms
- Graphical User interface
- Batch processing component

These components connected into the system as shown in the image 1.1.

As we can see by the design model above, the software was developed using a modular approach with claims of a high degree of extensibility to the existing tool. As C# .net was used as the base programming language the resulting GUI is of an extremely high standard, with a professional presentation and feel when using it. A criticism however is although the resulting software developed has an extremely well presentable front end, the disadvantage of using a Windows based development environment and programming language is that the software is not transferable between different operating systems. The

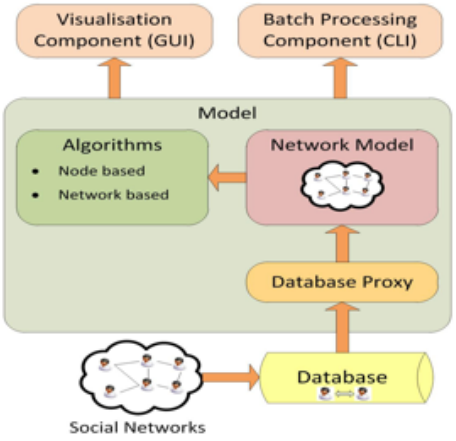


Figure 1.1: Figure

unfortunate issue with this is that it means the software cannot be run on a Linux or mac based environment.

The original design conceived had the algorithms module as a single module with no separation of the type of algorithm to be run or implemented. This was changed in the final design as shown above, so as to provide a greater degree of flexibility for potential researchers using the tool.

We intend to use the graphical user interface (GUI) in such a way that the interface is essentially a standalone client program. It should be able to access and transmit data and instructions from the client machine to a cluster of machines at the department with the software becoming a lightweight visualisation tool for different social networks or so that it could serve as a dual purpose tool, able to connect and control a cluster of machines or run independently on a single system (e.g. retain all current functionality but incorporate the functionality of a distributed system). The reason for the further development of the previous group's interface is that because the overall look and feel of the GUI is of such a high standard it would be logical to adapt it to the requirements of this project. The database schema is sound and proven to be extremely versatile with the use of Boyce-Codd normal form to reduce data redundancy. This database schema, should the current interface be user-able, is again, another highly desirable feature of the previous group's work as the ability to switch between a continuous or discrete dataset is implemented extremely well and therefore would allow us to perform research on different types of data such as real time twitter analysis or a batch of data.

There are a few issues to note with this project that have been identified after careful consideration of the requirements of the current project. The main issue that has been identified is the lack of testing in terms of scalability. While the report produced for the social network analysis tool clearly and accurately identifies problems and errors that could arise from use of the tool such as lack of tape (.i.e. the tape that is played when a user plays the message interaction is empty and therefore would throw an error), node properties incorrect or other errors, there were no reports of scalability testing. This could be perceived to be a serious flaw with the tool as it can clearly been seen in the results of research experiments conducted that the tool may have only been tested up to

a total of 7000 nodes. While this is quite a large number, we intend to extend the tools ability to incorporate a large amount of twitter data. This form of data consists mostly of a large number of small clusters with a few edges linking clusters together. With nearly 500 million users on twitter and approximately 250 million active users, the ability to visualise the network along with its associated temporal data is extremely desirable and with many claims throughout the report as to the ability of the tool to handle large amounts of data, it is a concern that the interface we intend to extend with simple object access protocol (SOAP) functionality could not be suitable for the size of graphs we wish to analyse. However, as the tool can be run in CLI mode (batch processing mode without the GUI), the claims of large data processing could in fact be true, however, as we intend to push the computational side of the tool onto a cluster for greater processing power and to achieve a distributed system our main concern with the previous tool is that the interface may not be scalable in terms of graph visualisation.

Another concern is the dependencies structure associated with the design and implementation of the previous tool. As we are only interested in the GUI, it appears substantial changes would need to be made to the tool in order to separate the GUI from the rest of the tool as it is dependant on nearly every other module within the tool.

While it is clear that the SNA tool was developed to an extremely high standard, the concerns with the previous work in relation to our own is the potential issues with scalability and modular dependency.

There are however, extremely useful sections of the final report produced by the previous group that will in fact prove very useful. For example, they highlight in great detail the operations and possibilities associated with the tool and its database proxy API's developed. This extremely large quantity of high quality specification regarding modules such as the database proxy and the database schema, should prove useful in attempts to dissect the tools structure and code. Overall the previous groups work was of a very high standard and the only criticisms that can be put forward apply to issues that they would naturally, given the specification of the SNAT project, not have tested for.

(1) : http://www.mediabistro.com/alltwitter/twitter-active-total-users_b17655 Accessed 23/4/2012

Chapter 2

Motivation

2.1 Why Is This Important?

Crimes are frequently committed by a network of individuals, be they loose-knit, tight-knit, grassroots clusters or hierarchical supply chains. Aspects of the relationships and power structures that compose these networks are betrayed in the resulting address books and digital social networks, which the police can frequently gain access to. However, this information has been distorted, almost flattened; a deep relationship may appear as a series of unrelated entries in an addressbook. While direct detective-work is invaluable for making these connections, the ever-increasing quantity of data to be processed is rendering this approach less effective. By modelling these networks as graphs, and through the application of various algorithms, we can attempt to recover some of this 'lost' data. This will provide information about both individual users and the overall network; cluster identification may detect distinct groups within a single network and the influence and centrality of individual people can likewise be calculated. At its simplest this information could help inform decisions on which people the police should focus their resources on apprehending, and where in the network to introduce undercover agents to maximise their influence.

2.1.1 Scale

As a consequence of the computational complexity of these algorithms, existing tools have serious scalability issues. The advent of social networking sites has led to sprawling networks with vast quantities of low-grade information that needs to be heavily processed to extract anything of value. We have tackled this by distributing the workload across clusters. Clusters are networks of computers typically built with commodity-grade components, rendering them cheap and scalable relative to traditional supercomputers.

2.1.2 Relevance

We chose to use publicly available data from the 'microblogging' site Twitter, as we lacked access to any police datasets. Twitter is used by two groups the police are interested in; the 'English Defence League' (EDL) and 'Unite Against Facism' (UAF). Previous projects used Enron's emails. Times have changed since then; emails only make up a fraction of a user's interactions and social networks have a far higher degree. We demonstrate these techniques on modern, relevant networks, exploiting the poor operational security exhibited our targets.

2.1.3 Alternative Applications: Marketing

Modelling influence manipulation and the like is also a core requirement of 'viral' marketing; marketers want to choose the ideal starting point in a network to achieve maximum saturation. At its purest, the website http://sponsoredtweets.com allows advertises to pay a set amount for a tweet from a particular user and influence propagation can be used to inform decisions about whether a well-connected root is worth the money.

2.1.4 Alternative applications: Persona Management

This won't just aid manual infiltration by undercover agents; once it is automated to a suitable degree it may be combined with 'persona management systems'; software that enables a single operator to control large numbers of fake online identities, massively increasing their voice and influence over discussions. Manual identification of influence relationships simply doesn't scale to this level.

http://www.faqs.org/patents/app/20090313274

Chapter 3

Research

This chapter is composed of three sections. The first concerns itself with existing research within the area of the analysis of social networks. The second discusses how influence within a network can be manipulated, which is related to the third section which discusses how behaviour within networks changes. The final section is concerned with distributed computing, and how it can be used for analysing graphs of social networks.

3.1 Social Network Analysis

Analysis of social networks is becoming ever-increasing in importance, with the huge growth social network sites such as Facebook[1] and Twitter[2] have experienced in recent years. This research undertaken within this section relates to various metrics which can be applied to a social network to identify important people within the network, and also how the structure of the network can be identified.

3.1.1 Community Detection

An important aspect of social network analysis is the detection of communities within a graph. The identifying of communities within a social network can help to identify group dynamics of the network.

A famous example of this is the friendship network within Zachary's karate club [42], which subsequently split into the two groups shown in Figure 3.1. The identification of two clusters, which cleanly divided the network into the two groups the karate club actually split into, shows the usefulness of being able to identify connections between groups of highly interlinked individuals.

[1] `https://www.facebook.com/`
[2] `http://twitter.com/`

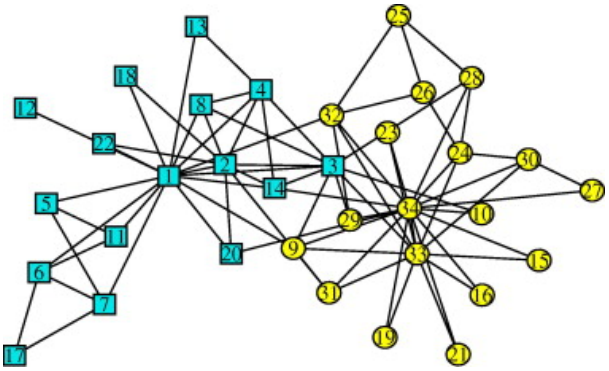


Figure 3.1: Clusters found within Zachary's karate club [42, 11]

Clustering

Within social networks, there are distinct regions of vertices with many connections passing between them, and a much smaller number connected to other vertices outside of this region. These regions are termed as *clusters*.

Social networks are a prime example of this. A person tends to have groups of friends distinct from each other, and friends within these groups are also friends of each other. Figure 3.2 is a social network constructed from data regarding mutual friends of a person. Within this Figure, it can clearly be seen that there exists five distinct regions of mutual friends, with very few links between these regions. These regions are the clusters described above.

With data presented in this manner, it is clear to an observer where these clusters exist, and how many distinct clusters there are. [18] presents an approach to identifying communities within a graph.

Partitioning

Partitioning of a graph differs from clustering slightly. To partition a graph, we first must know the number of partitions we wish to make, and also the size of each of these partitions. Clustering does not need to know this information as the clusters are produced by the algorithms performing the clustering. Partitioning is related to clustering as it splits the input graph into distinct smaller subgraphs of highly connected vertices, with the subgraphs representing communities within the original graph.

The process of graph partitioning is to divide a graph into smaller parts, such that the number of connections between these smaller parts is as few as possible. The partitions produced are non-overlapping, of a given size, and of a given number.

Graph partitioning is known to be a difficult and complex problem to solve. For simplicity, assume that we are to partition a graph into two partitions. To naively *brute force* all permutations to find the "best" partition, which minimises the number of connections between the two partitions, soon becomes unfeasible for anything but the smallest graphs,

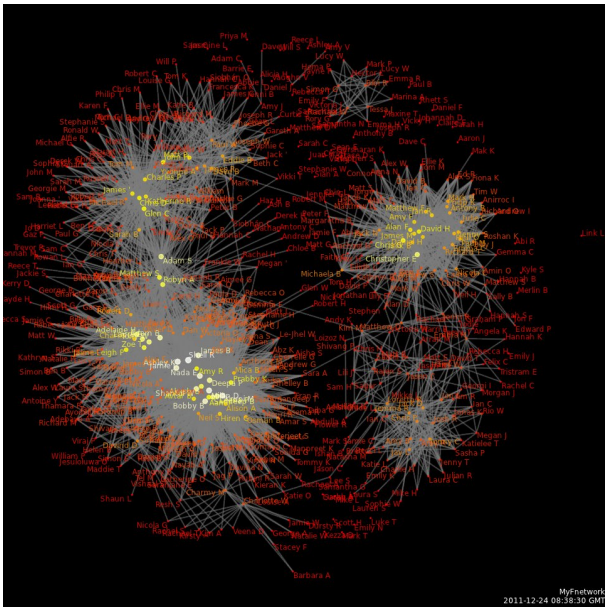


Figure 3.2: Social Network constructed from Facebook friends using the myFnetwork app

due to the number of ways of partitioning n vertices into two groups n_1 and n_2. This is $\frac{n!}{n_1!n_2!}$ permutations, and with application of Stirling's formula, can we rewritten as [31]:

$$\frac{n!}{n_1!n_2!} \simeq \frac{\sqrt{2\pi n}(n/e)^n}{\sqrt{2\pi n_1}(n_1/e)^{n_1}\sqrt{2\pi n_2}(n_2/e)^{n_2}} = \frac{n^{n+1/2}}{n_1^{n_1+1/2}n_2^{n_2+1/2}} \tag{3.1}$$

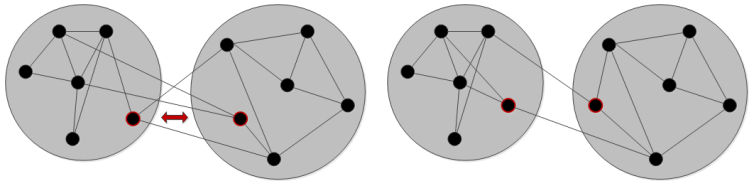
Which, if our two partitions n_1 and n_2 are equal in size, then the number of permutations of the partition is [31]:

$$\frac{n^{n+1/2}}{(n.2)^{n+1}} = \frac{2^{n+1}}{\sqrt{n}} \tag{3.2}$$

This shows that running time of the *brute force* algorithm to find all permutations grows exponentially with the size of the input graph.

Kernighan and Lin proposed an algorithm to solve the partitioning of a graph into two sections [24]. Initially, the graph is split into two partitions, A and B. For each pair of vertices $(i,\ j)$ such that $i \in A$ and $j \in B$, the algorithm finds the pair which reduces the number of connections between A and B the most (see Figure 3.3a), and swap them (Figure 3.3b). If there is no pair which reduces the number of connections between the two partitions, then the pair which increases the number of connections by the smallest amount are swapped. This is then repeated until vertices within one partition have been swapped, with the condition that once a vertex has been swapped, it cannot be swapped again. Following this, each state the graph was in after a swap is revisited, with the state with the minimum number of connections between A and B is minimal. These steps are then repeated until A and B do not change between rounds. The algorithm has been shown to run in $O(n^2 \log n)$ time [24], which is marked improvement over the *brute*

force $O(2^n)$, but still remains slow and unsuitable for graphs with a over a few thousand vertices [31].



(a) Original network

(b) The same network after interchange of two vertices

Figure 3.3: The Kernighan-Lin algorithm [31]

Figure 3.3 shows the effect of the Kernigham-Lin algorithm on the original network (Figure 3.3a). The two vertices highlighted with red reduce the number of connections between the two groups the most, so are swapped. This swap is also the best swap which happens during this iteration of the algorithm, so the network in Figure 3.3b is selected as the initial network for the next iteration. As no vertex swapping can improve on this, and as such this network is the best network for the next iteration, this network is the solution from the algorithm and the algorithm terminates.

To split a graph into more than two partitions, repeated application of partitioning the subgraphs produced from previous partitioning is applied.

There exist other algorithms to partition graphs, notably spectral algorithms [35, 14], which make use of properties of the matrix which can be used to define a graph. These spectral algorithms are a lot more complex in the operations needed to execute them, however they operate in $O(n^2)$ time, which is an order n faster than the Kernighan-Lin algorithm [31]. It is also noted that spectral algorithms do not necessarily give the best partition, with the partitions produced being of the rough general shape but not quite as good as other algorithms [31].

k-Cliques

The term *clique* is introduced by Luce and Perry in [27] to describe a subgraph which consists of at least three vertices, each of which are fully connected with each other. From a social network perspective, this translates to saying that for each person in the clique, all of their friends within the clique are friends of each other as well.

A k-clique is defined as a clique which has size k. A *maximal clique* is a clique to which no more vertices can be added without violating the conditions of a clique, and the maximum clique is a clique within a graph which contains the largest number of vertices. A 3-clique is also known as a triangle, due to the shape of the graph they are normally represented as (Figure 3.5a).

Figure 3.4 shows an example graph highlighting a 3-clique. It has one maximum clique {2,3,4} which is highlighted in red, and six other maximal cliques {1,2}, {2,8}, {4,6}, {5,6}, {6,7}, {7,8}.

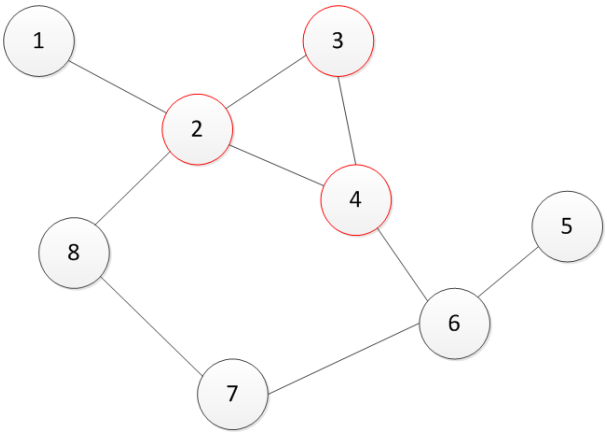
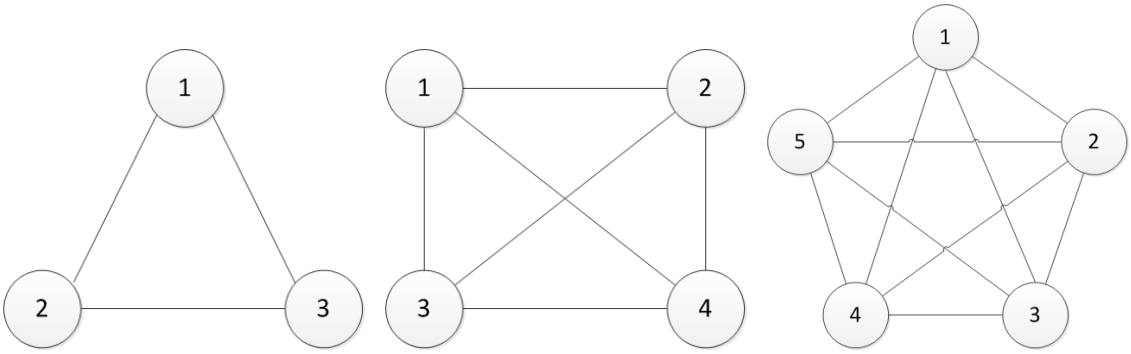


Figure 3.4: Graph highlighting a 3-clique



(a) 3-clique (b) 4-clique (c) 5-clique

Figure 3.5: k-cliques

Identifying cliques, and solving problems involving cliques as been shown to be computationally hard [4, 10]. Cliques are either too common, with cliques of only a few members are frequently too numerous to be helpful, or too rare, with large cliques too limiting to be of use [10]. Also, computation of identifying cliques scales worse than any polynomial of the problem size, making the process unfeasible for large graphs [10, 6].

There have been numerous generalisations of the clique construct to improve usefulness of the clique through relaxation of some of the properties of the clique. These include the n-clique [26], n-clan [1] and n-club [30]. However these new constructs do not solve the issues of identifying cliques, and remain hard to compute and produce too many results.

k-Trusses

Following from relaxing properties of the clique, Cohen in [10] introduces a new construct called a truss. A k-truss is a subgraph where an edge between two vertices, A and B, exists if at least k-*2* other vertices are connected to both of A and B. The k-truss has similar definitions to the clique. A maximal k-truss is a k-truss that is not a proper subgraph of another k-truss.

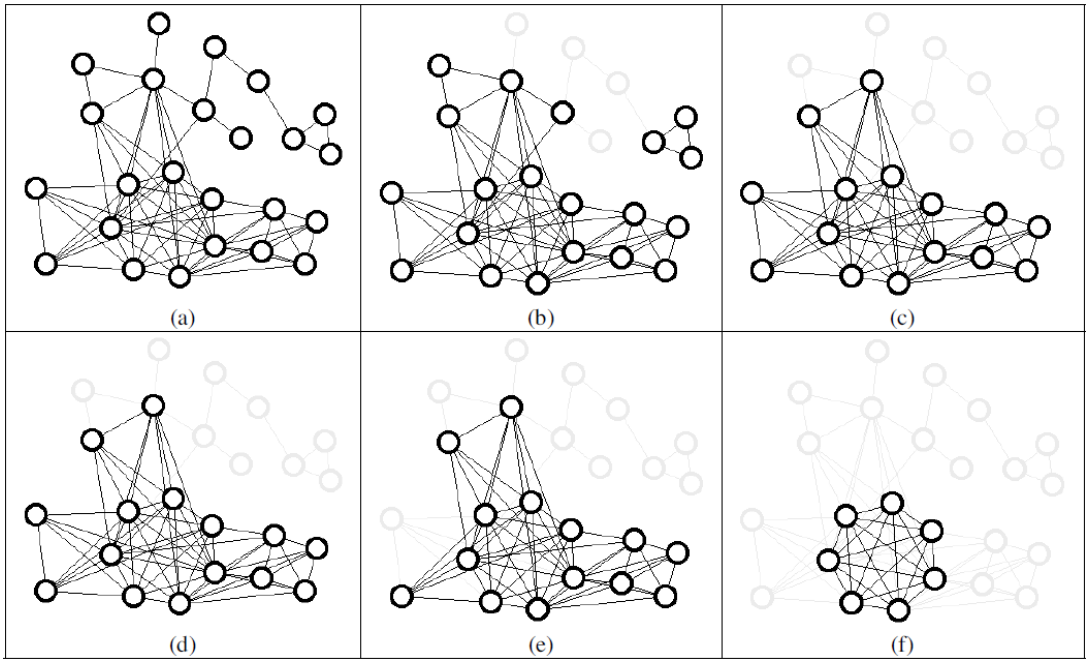


Figure 3.6: An Example of Maximal Trusses. (a) original graph, (b) 3-trusses, (c) 4-truss, (d) 5-truss, (e) 6-truss. Note that (c) and (d) are the same. [10]

Figure 3.6 shows an example application of k-trusses. Figure 3.6(a) shows the original graph. Figures 3.6(b-f) show the k-truss for increasing values of k. If the same original graph were to be analysed for k-cliques for the same values of k, more vertices in the graph would have been removed in earlier stages, which would have lost the interesting stage where Figures 3.6(c) and (d) are identical.

The k-truss is also computable in polynomial time to the input graph, which offers a marked improvement on the computation costs of identifying cliques and similar constructs. [10] shows that the naive algorithm for computing maximal k-trusses is bounded by $O(n|E|^2 + n)$, where n is the number of vertices in the graph, and $|E|$ is the number of edges in the graph. A more efficient method is shown to take $O(\Sigma d^2(v))$, where $d(v)$ is the degree of a vertex, v.

3.1.2 Clustering Coefficients

Clustering coefficients provide a way to measure the amount to which vertices within a graph cluster together. Calculating clustering coefficients requires knowing the number of triangles within the graph.

Global Clustering Coefficient

The global clustering coefficient is also known as the transitivity of a network. This is because it is similar to the mathematical property of transitivity. A relation "$\circ$" is said to be transitive if $a \circ b$ and $b \circ c$ imply together imply $a \circ c$. An example would be the equality relation, as if $a = b$, and $b = c$ it follows that $a = c$ [31]. Within a social network, the transitive relation could be described as *"the friend of my friend is also my friend"*.

A global clustering coefficient of 1 implies a social network where all components are cliques. A global clustering coefficient of 0 implies a social network where there are no mutual friends between any two people.

The global clustering coefficient is defined as:

$$C_G = \frac{3 * number\ of\ triangles}{number\ of\ connected\ triples} \tag{3.3}$$

A connected triple is simply three vertices uvw connected with edges (u, v) and (v, w); edge (u, w) does not need to be present [31]. The factor 3 is present because each triangle is composed of three connected triples.

Local Clustering Coefficient

A clustering coefficient can also be applied to each vertex within a graph. The local clustering coefficient is the probability that vertices connected to a vertex are themselves also connected to each other.

The local clustering coefficient is defined as:

$$C_L = \frac{number\ of\ triangles\ connected\ to\ vertex\ v}{number\ of\ triples\ centered\ on\ vertex\ v} \tag{3.4}$$

Within social network analysis, the local clustering coefficient

Network Local Clustering Coefficient

The network local clustering coefficient is simply the mean of all local clustering coefficients within a network. It can be used to show the average level of connectedness between vertices in a network.

$$C_N = \frac{1}{n} \sum_{i=1}^{n} C_i \tag{3.5}$$

C_i represents the local clustering coefficient of the vertex i. The result of this calculation is a number between 0 and 1, with 1 meaning all vertices are connected to each other [41].

3.1.3 Centrality

Following from identifying communities within a social network, it is also apparent that establishing the most *central* or *important* person is within a community. This person is most likely going to have a large influence over the members of the community.

Degree Centrality

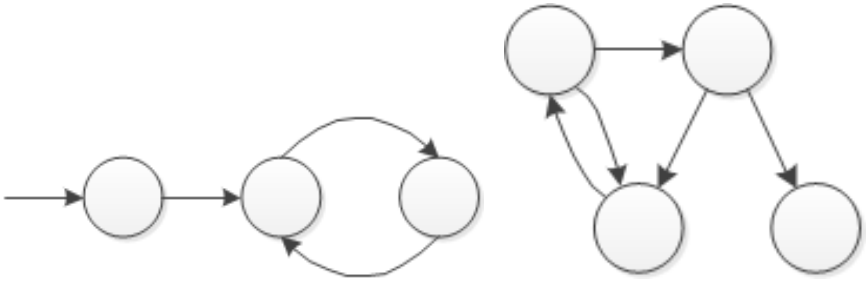
The degree centrality is a simple measure to produce for a social network. For an undirected graph (relationships are symmetric), the degree centrality for a vertex, v, is simply the number of edges connected to it. For a directed graph, say the network of Twitter[3] users following, and being followed by others, two measures are defined, the in-degree and out-degree. These are simply the number of incoming and outgoing edges respectively.

Whilst a fairly simple measure to compute, degree centrality can be useful. A person with a high degree centrality is likely to hold a larger influence amongst the people they share a connection with, as they are more popular. [31, p. 169] suggests that the number of citations an academic paper receives can be used as its in-degree centrality, and is a crude measure of whether the paper was influential and as such had an impact on scientific research.

Eigenvector Centrality

The eigenvector centrality measure identifies important vertices within a graph by awarding each vertex a score proportional to the to the sum of the centralities of adjacent vertices. This means that a vertex has a high centrality because it either has many neighbours, or because its neighbours have a high centrality themselves [31].

[3] `http://twitter.com/`



(a) A simple rank sink (b) A dangling link

Figure 3.7: Problems with simplified PageRank

PageRank

The PageRank algorithm [34] developed by Google is a further extension of the eigenvector centrality measure. The centrality score awarded to each vertex is proportional to the centrality of its neighbours divided by the number of outgoing connections. The PageRank algorithm was devised to rank webpages based on the quality and quantity of the incoming links to that webpage.

A simplified version of PageRank can be defined as, where u is the vertex to be ranked, B_u is the set of vertices with edges incident on u, N_v is the number of outgoing edges from v, and c is a factor used for normalisation [34]:

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} \tag{3.6}$$

Equation 3.6 does not take into account of two properties which can exist within a directed graph, rank sinks and dangling links. A rank sink is a closed loop within the graph, which has incoming edges, but no edges leaving the loop. As such, these loops 'keep' the PageRank which enters them and can artificially cause these vertices to be ranked higher. A dangling link is similar, but is just a vertex with no outgoing edges. The approach taken by Page et al. is to remove any of these dangling links from the graph, and if this produces new dangling links, remove these new dangling links and repeat until no dangling links remain.

This simplified PageRank makes use of the *random surfer model*, which simulates that for each outgoing link from a webpage, a user randomly selects on of these links, and continues browsing. A real user would not continue in this fashion if the outgoing links from successive webpages form a loop, they would navigate to a new webpage not linked by these pages, based on the distribution factor E. This is modelled within PageRank as a rank source:

$$R'(u) = c \sum_{v \in B_u} \frac{R'(v)}{N_v} + cE(u) \tag{3.7}$$

The rank source is the amount of rank each vertex in the graph contributes to the system. This models the behaviour of a real user and ensures that rank sinks do not accumulate rank.

Closeness Centrality

Closeness centrality is a measure of the average distance from a vertex to other vertices. A geodesic path between two vertices is the minimum number of edges required to connect them within a graph. The closeness centrality defined in equation 3.8 is the sum of the length of all geodesic paths, d_{ij}, from a vertex, i, to all other vertices, j:

$$l_i = \frac{1}{n} \sum_j d_{ij} \tag{3.8}$$

A slight alternative is to exclude the geodesic path d_{ij} when $i = j$ because this is trivially 0, and a vertex's influence on itself is not relevant to the calculation of centrality:

$$l_i = \frac{1}{n-1} \sum_{j(\neq i)} d_{ij} \tag{3.9}$$

Closeness centrality defined in equations 3.8 and 3.9 produce results differently to other measures of centrality, whereby a lower value for l_i indicates that vertex i is more central within the network. Because of this, closeness centrality can also be defined as:

$$C_i = \frac{1}{l_i} = \frac{n}{\sum_{j(\neq i)} d_{ij}} \tag{3.10}$$

Equation 3.10 produces results more consistent with other measures of centrality as a higher value of C_i indicates vertex i is more central. This equation is simply the inverse of equation 3.8.

Within a social network, a person with a high closeness centrality, defined by equation 3.10, could find that their opinions reach others in the community faster than someone with a lower closeness centrality [31].

Betweeness Centrality

Betweeness centrality is different measure of calculating centrality. Instead of being concerned with the quantity, or quality, of links, betweeness centrality measures the extent to which a vertex lies on paths between other vertices.

Betweeness centrality for a vertex i is mathematically defined as the sum of the number of shortest paths, n^i_{st}, from vertex s to vertex t which pass through i, divided by the total

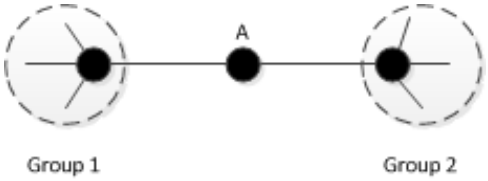


Figure 3.8: A low degree vertex with high betweeness [31]

number of shortest paths from s to t:

$$x_i = \sum_{st} \frac{n_{st}^i}{g_{st}} \tag{3.11}$$

Figure 3.8 shows a sketch of a network which contains two distinct groups connected to each other via vertex A. All paths between the two groups must pass through vertex A, so it has a high betweeness even though its degree is low [31]. With this example as well, it is quite likely that vertex A would score poorly with other measures of centrality, however due it forming the bridge between the two groups, vertex A is quite likely to be quite influential in controlling the flow of information between the two groups.

3.2 Influence Propogation

Influence propogation algorithms attempt to model the spread of ideas (Rogers 2003), viruses (Anderson and May 1991), word-of-mouth recommendations (Goldenberg, Libai and Muller 2001), viral marketing campaigns (Kempe, Kleinberg and Tardos 2003) or other transmissable entities, through a social network.

Influence propogation algorithms make the following assumptions about the social networks that they model. Social networks are modelled as directed graphs, with nodes representing individuals, and vertices representing relationships between individuals. In most influence propogation models, nodes can be in one of two states: "converted" or "unconverted". At the begin of the simulation, a subset of the nodes will be converted; over time they will convert their unconverted neighbours. There is an assumption of monotonicity, ie, converted nodes do not become unconverted.

The exact conditions that determine when nodes will change from unconverted to converted depends on the influence propogation model used. There are three main models of transmission used; the independent cascade model, the linear threshold model, and ???

Mention more complicated models, three states etc

3.2.1 Independent Cascade

In the independent cascade model, whenever a node becomes converted, it has one chance to convert each of its neighbours. Every edge (A -> B) is assigned a weight between 0

and 1, which represents the probability that node A can convert node B, or vice versa.

Assume a node n has a set of x neighbours n_0,n_1,n_2...n_x each with edge weights e_0,e_1,e_2...e+x. When n is converted, each neighbour n_m has a e_m chance to become converted.

This can be expressed in pseudocode as follows:

```
recently-converted-nodes = a queue of the initially converted nodes
while recently-converted-nodes is non-empty {
  node = recently-converted-nodes.next()
  for each neighbour in node.neighbours {
    if rand(0,1) < neighbour.edge-weight {
      neighbour.converted = true
      recently-converted-nodes.enqueue(neighbour)
    }
  }
}
```

3.2.2 Linear Threshold

In the linear threshold model, every node has a threshold value t. When the number of converted neighbours is greater than t, the node becomes converted. Again, edges can be weighted, in which case the condition for conversion is:

Maths: sum for all neighbours(if neighbour is converted(0,1) * edge weight)

That is, nodes are converted when a threshold of their neighbours are converted. In pseudocode:

```
do {
  changed-nodes = 0
  for each node in unconverted-nodes {
    neighbour-influence = 0
    for each neighbour in neighbours {
      if neighbour.converted {
        neighbour-influence += neighbour.edge-weight
      }
    }
    if neighbour-influence >= node.threshold {
      node.converted = true
      changed-nodes += 1
    }
  }
} while changed-nodes > 0
```

3.2.3 Decreasing Cascade Model

The decreasing cascade model fits situations where nodes are less likely to be influenced with each conversion attempt. One real world example was [cite:digg], where it was found that a standard independent cascade model did not fit the spread of stories through the social news site Digg. The decreasing cascade model proved to be a much better fit for the data; one possible conclusion from this is that if users do not "digg" (vote for) a story the first time they are exposed to it, they are unlikely to do so on subsequent exposures.

The decreasing cascade model works similarly to the independent cascade model, but after a failed conversion attempt, the unconverted node becomes more difficult to convert in future. In the most basic example, the probability of future conversions becomes 0 after 1 failed conversion attempt.

3.2.4 Influence Maximisation Problem

[Maximizing the Spread of Influence through a Social Network]

[http://arxiv.org/pdf/math/0612046.pdf]

One common use of an influence propogation model is to determine the best subset of nodes, that, if converted at the start of the simulation, will maximise the number of converted nodes at the end of the simulation. More formally, given a fixed budget k, and a function f(S), which takes an initial set of converted nodes and computes the expected final number of converted nodes, find a set of k nodes that maximises f(S).

One practical example would be a viral marketing company that might wish to kickstart their campaign by giving a group of influential bloggers a free trial of their service, and hope that these bloggers would then hopefuly recommend the product to their followers, who would recommend it to their followers, and so on. One strategy for choosing the optimal set of free trial users would be to rank the bloggers by a precomputed influence rating, and then select the top k most influential individuals from the list.

Even if influence of a single node could be computed reliably, this strategy would not generally find the optimum subset. For example, the most influential bloggers might share the same audience, wheras a less popular blogger might have considerable influence amongst a niche audience. In graph theory terms, there may be a short distance between two influential nodes – they may even share overlapping neighbourhoods – which means there may be a set of nodes far from the influential pair. These nodes might be more influenced by a node that is less globally influential, but nearer. Obviously, the structure of the graph determines the likelihood of this. For example, graphs with a community structure are more likely to require an influential seed node within each community to achieve maximum influence.

Solving the optimum problem is NP-hard, for both the linear threshold and independent cascade models. This can be shown by reducing the independent cascade maxmisation problem to a special case of the set cover problem, and the linear threshold maximisation problem to a special case of the vertex cover problem.

However, a greedy solution works as an approximation. !he set of nodes chosen by the greedy solution will convert at least 63% of the nodes converted by the optimum solution. This result can be proved by showing that the function f(S) is monotonic and submodular.

The monotonicity property requires that f(S) increases whenever an element is added to S; ie, f(S+v) >= f(S) for any S, v. Proving this property of f(S) is trivial. The submodularity property can be intuitively thought of as a "dimininishing returns" property; if S is a subset of T, the marginal gain from adding an element v to S is at least as high as adding the same element to T. Formally:

f(S+v) - f(S) >= f(T+v) - f(T)

Nemhauser, Wolsey and Fisher showed that a greedy-hill climbing algorithm can be used to find the optimum value of such a submodular function to within a factor of 1/(1-e), or 63%. The greedy algorithm works by adding elements to the seed set one at a time, each time choosing the element that provides the largest marginal increase to the value of the function. In the case of influence maximisation, this is the node that adds the greatest number of converted nodes at the end of the simulation.

Of course, this method relies on a fast method of working out the expected number of nodes that will be converted by a seed set. There is currently no simple means of calculating this value, so in practice a Monte Carlo method is used; ie, running several iterations of the model and taking an average of the final number of converted nodes. This is computationally expensive, though some work has been done to improve the performance. [CITE??]

3.2.5 Adversarial Social Networks

There are limitations to the two-state models described above. For example, they cannot represent any scenario where two competing ideas spread through a network. An alternative model for such situations allows nodes to be in one of 3 states; "unconverted", "converted to idea A", or "converted to idea B". In this case, A and B represent opposing beliefs.

The paper [Influence Propagation in Adversarial Social NetworkâĂŤImpact of Space and Time] looks in depth at this situation, and applies it to the spread of radical and counter-radical ideas through the Muslim community.

They observe that the key reason that the standard influence propogation model cannot capture the nature of adverserial networks is as follows. Observe the network described in fig???. Node_2 is initially converted to idea A, and node_4 is converted to idea B. All other nodes are unconverted. At some point, node_9, which is a bottleneck between the two sides of the graph, may be converted to one idea or the other. Once this happens – assume that it is converted to idea A – there will be no chance that it will ever be converted to idea B. For that reason, there is no way that idea B can "pass through" node_9 and so no way that nodes 10 to 14 will ever be converted to idea B. The standard influence models have no way to capture this situation.

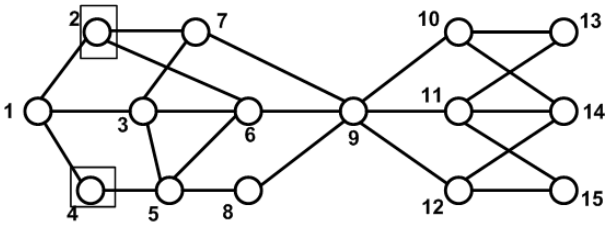


Figure 3.9: Adverserial network

Other real-life applications of this concept include marketing – promoting your product while a rival company is promoting a competing product – or preventing the spread of misinformation. A recent example was the Kony 2012 campaign, a viral video spread by a non-profit activist organisation to highlight the issue of child soldiers in Uganda. The video spread very rapidly via online social networks; however, this soon led to people questioning the aims and methods of the campaigning organisation. This led to a counter-campaign that also spread virally, via social media. (Of course, which party provided misinformation in this scenario depends on individual perceptions).

[Limiting the Spread of Misinformation in Social Networks] looks more in depth at the problem of countering mis-information. They quantify this situation as a "Multi-Campaign Independent Cascade Model", where two campains, C and L, spread through a network. Each campaign has an initial seed set of nodes, A_C/A_L. As with the independent cascade model, when a node is converted, it has one chance to convert each of its neighbours. The probability that a node n converts its neighbour o is given by p_C_n_o / p_L_n_o, depending on which campaign the node was converted to. Note that in the general case, their may be different probabilities for the two campaigns (representing the situation where one campaign is more "infectious" than the other). Unlike in the independent cascade model, when an unconverted node has multiple converted neighbours, the order of "conversion attempts" matters; eg, campaign C may succeed in converting a node before campaign L has had the chance to convert it. Their model assumes that one campaign always precedes the other in such situations, ie, that campaign always gets the "first chance".

They also used a more specific model, the "Campaign Oblivious Independent Cascade". This is a special case of the multi-campaign model, where the conversion probabilities are the same for both campaigns.

They then look at minimising the influence of the opposing campaign, as opposed to maximising their own influence; this problem is referred to as the "eventual influence limitation problem". This can also be seen as maximising a function g(S), where S represents the seed set of campaign A (the "good" campaign), and g(S) computes the number of "saved" nodes – ie, the number of nodes that were converted to campaign A, but would have been converted to campaign B if A was not present. Even with some simplifying assumptions (campaign B has a seed set consisting of only one node, campaign A has a conversion probability of 1), the problem is still NP-hard.

However, as with the two-state influence models, the function g can be shown to be submodular, and therefore a greedy algorithm can again achieve a performance of 1-

(1/e) or 63%. This greedy algorithm works similarly to the one used for the influence maximisation problem; ie, iteratively adding to the seed set the node that will provide the best marginal improvement.

Although this is a polynomial-time algorithm, the datasets used for practical social network analysis may be extremely large, making even this algorithm unworkable in practice. Some alternative heuristics are therefore possible. These include the "degree centrality heuristic" (ie, choosing the nodes with the most incident edges), the "early infectees heuristic" (ie, choosing the nodes that are expected to be infected by the rival campaign early on), and the "largest infectees heuristic" (ie, choosing the nodes that are expected to infect the highest number of other nodes). By evaluating these heuristics via Monte Carlo simulations, it can be shown that the largest infectees heuristic is the most effective, approaching the performance of the greedy algorithm. The early infectees heuristic proved to be the least effective approach.

3.3 Influence Manipulation

todo: check KNN + biblio

3.3.1 Recommender Systems

Although relatively little publicly available research exists on the topic of manipulating social networks, there is a wealth of information on attack and defence of recommender systems. Recommender systems are a tool used to tackle the problem of information overload; where more content exists than a user has time to evaluate. As the name suggests, they do so by performing an initial evaluation on behalf of the user, whittling the quantity of presented content down to a level the user can cope with. Amazon and Twitter them to recommend products and followers respectively. They are fundamentally similar to social networks in the underlying graph structure, the attempt to model a real-world phenomena, and the clear incentives various parties have to manipulate them.

Attacker Goals

Collaborative Recommendation: A Robustness Analysis defines two simple intents an attacker may have: 'nuking' an item (decreasing its rating / the frequency with which it is recommended) and 'pushing' an item (increasing its rating). 'Vandalising'; making an entire system function poorly, is discussed in another paper. Every one of these intents corresponds to something one may wish to achieve in a real influence network; altering an entity's influence or disrupting an entire network.

k-Nearest-Neighbour

The actual attacks used to achieve a given goal vary greatly depending on the implementation of the target system, but the base components remain the same. Early recommender systems such as Tapestry used a variation of the k-Nearest-Neighbour algorithm to identify like-minded people to new users, and uses this as a basis for their recommendations:

$$p_{u,i} = \bar{r_u} + \frac{\Sigma_{v \in U_{u,i}[w_{u,v}(r_{v,i} - \bar{r_v})]}}{\Sigma_{v \in U_{u,i}|w_{u,v}|}}$$

This calculates the user u's predicted rating for item i. $\bar{r_u}$ is the user's average rating over all items, $w_{u,v}$ is a measure of the similarity between users u and v, and $U_{u,i}$ is u's neigbourhood for item i. The final result is thus the user's average rating, modified by the rating others gave it, weighted by their how similar their taste is to the user.

This is by design not dissimilar to the reputation and influence systems underlying the networks that we want to tackle. The item-item algorithm's fundamental difference is that it directly calculates the distance between items rather than between users. "People who liked X also liked Y"

Attacks

The paper 'Shilling Recommender Systems for Fun and Profit' considers attacks in terms of their intent, targets, required knowledge and cost. There is a cost per shill whether they are an undercover agent, the product of a persona management system or simply a solved captcha (the latter being the only part of account-creation on a website that can't be automated, and costing $1.39/1000 from deathbycaptcha.com). The simplest push/nuke attacks revolve around gaining as many neighbours as possible, then rating the target item. The simple but ineffective approach to the linking stage taken by the 'RandomBot' is to assign a random rating to all the items in the database. The 'AverageBot' attack instead assigns to each item the average rating it's received so far. This has a lower cost at the expense of a greater knowledge requirement. Even on a network like Twitter where most data is publicly available, there is a cost associated with acquiring and processing it. Toward Trustworthy Recommender Systems: An Analysis of Attack Models and Algorithm Robustness introduces two additional, more implementation specific attacks. The bandwagon attack achieves a high level of association for a low cost by rating a small number of frequently and consistently rated items (e.g. bestselling novels). The segment attack is simply a refinement that only attempts to promote items to those already disposed to buying them. Other attacks include the self-explanatory 'reverse bandwagon' and 'love/hate' techniques.

These attacks provide us with a number of starting techniques, as well as an insight into the relationship between knowledge and cost requirements. A hybrid persona/person approach may prove necessary.

Defences

The defences are also relevant. It's entirely plausible that as these attacks are turned towards infiltrating and manipulating social networks, parties with a vested interest in protecting the network's integrity will look at the compatriot countermeasures; attack detection and attack mitigation. Shill detection algorithms are already in use on Twitter to detect and remove the original shills; spambots. It may be well beyond the scope of this project, but that doesn't diminish the value of a design informed by an understanding of future problems.

Mitigation

One mitigation that is at least conceptually simple is to use a hybrid model where a significant portion of the model isńt taken from user data. For example, semantically enhanced collaborative modelling uses text-mining techniques to create a similarity measure entirely independent of user input. From the perspective of a persona management system, all target systems use a hybrid of malleable online input and untouchable real-world interactions. Physically deployed undercover operatives have almost the inverse restriction; excessive manipulation of online systems may raise suspicion, something distinctly more perilous for those physically present.

Detection

Attacks may be detected using simple metrics such as analysing user's rating deviation and profile length; the randomBot attack has a clear footprint in this regard. Perhaps in recognition of the inherent inaccuracy of such measures, once a profile is classified as "suspicious" its ratings are simply ignored, rather than it being booted off the system as might happen to a hacker on an attack aware application. Groups of shills may also betray themselves by acting in unison. Interactions on social networks are stored indefinitely, so even now the utmost care must be taken to avoid these signature patterns.

`http://blog.mozilla.org/webappsec/2011/02/02/attack-aware-applications/`

Original: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.2238\&rep=rep1\&type=pdf`

Fun&profit: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.579\&rep=rep1\&type=pdf`

Latest: `http://maya.cs.depaul.edu/~mobasher/papers/mbbw-acmtoit-07.pdf`

3.4 Emergent Behaviour

The purpose of this component of the project is to examine and describe how networks of agents converge and cooperate, and to look at measuring the robustness of this coopera-

tion of such networks in the presence of infiltrators. More specifically, we will be looking at networks which use a tag-based approach to cooperation based on the work done by Riolo, Cohen, Axelrod (RCA) [37]. The infiltrators will be self-interested agents, agents which do not follow the social norms of the network. In our case, these self-interested agents could be thought of as law-enforcement individuals, who may be interested in joining such a network but not cooperate. In general the presence of self-interested agents have a negative affect on the cooperation within such a network [19]. We will therefore look at methods to preserve the high rates of cooperation in these networks when self-interested agents are present.

3.4.1 Existing Research

The work will be based on a number of existing pieces of research on tag-based cooperation within a network of agents in the presence of self-interested agents.

3.4.2 RCA Tag-based Cooperation

Tag-based cooperation was introduced to observe the nature and method that cooperation in a network emerges [37]. The purpose of this research was to observe how cooperation can be achieved in a network where there is no direct reciprocity.

The cooperation approach described by RCA takes place in a population of agents. Each agent is assigned a single tag and a single tolerance value. Both the tag, τ, and tolerance, T, are selected randomly from a uniform distribution between 0 and 1. The tag is a cultural artefact—a form of a public identifier. The tolerance is used to determine the willingness of an agent to cooperate.

The approach described by RCA is generational. Each generation is divided into two phases: a donation phase and a reproduction phase. The donation phase occurs first and allows every agent to perform a number of donations. After the donation phase the reproduction phase allows agents to reproduce based on the relative success of each agent—more successful agents will produce a larger amount of offspring.

During the donation phase, each agent in the population is selected to make a number of donations. The number of donations each agent is selected to make is known as the number of interactions pairings per generation, P. When an agent has been selected for donation an agent is picked at random to which they have the opportunity to donate. In the RCA approach an agent will donate to the random agent if the difference between the two agents' tags is less than the tolerance of the donator, using the formula presented in 3.10. If an agent chooses to donate, they will incur a small cost, c, and provide a benefit, b, to the receiving agent.

$$|\tau_A - \tau_B| < T_A$$

Figure 3.10: Equation for determining if an agent will donate

The cooperation of a population of agents is measured by the donation rate in each generation.

Once every agent has been given an opportunity to donate P times, the reproduction phase takes place. During the reproduction phase, each agent reproduces in accordance with their relative success. The most successful agents produce more offspring than the less successful agents. Each agent compares itself to a randomly selected agent. If the agent is less successful than the randomly selected agent then its offspring derives from the random agent. If the agent is more successful than the random agent, then its offspring derives from itself. The offspring takes the tag and tolerance values of the more successful agent. There is a small chance that the tag and tolerance values could be mutated. If the tag is mutated, the offspring will receive a new tag taken randomly from the uniform distribution $[0, 1]$. If the tolerance is mutated, a small amount of gaussian noise with a mean of 0 is added to the tolerance value.

Once the donation and reproduction phases for a given generation have occurred, the process is repeated on the new population.

In experiments, RCA observed that the donation rate quickly stabilises after around 100 generations. Occasionally, an agent gets a tolerance value that is much smaller than the average tolerance as the result of a mutation. This mutation allows the agent to become more successful as they are less likely to donate. The impact of this reduces the overall donation rate until the agents converge on the new tag—due to the reproduction process—at which time the donation rate raises and stabilises again.

By experimentation, RCA found that with a higher number of interaction pairings in each generation, the donation rate and average tolerance increases. RCA also noted that when the cost of donating became too high relative to the benefit of receiving a donation, the donation rate dropped dramatically. From these results, we will choose the number of interaction pairings per generation, P, to be at least 3, the cost of donation to be 0.1, and the benefit of receiving a donation to be 1.

While RCA chose to use a generational approach where every agent produces offspring, an alternative approach based around a learning interpretation was mentioned but not implemented.

3.4.3 Learning-Interpretation of Reproduction

Hales and Edmonds (HE) applied the learning interpretation presented by RCA [21].

The learning interpretation differs from the *r*eproduction based interpretation of RCA in that no offspring are not produced in the reproduction phase. Instead, if during the reproduction phase an agent compares itself to a randomly selected agent which is more successful than itself, the agent will adopt the randomly selected agents' tag and tolerance values. This means that the population of agents remains the same but the tag and tolerances of the agents change in each generation. HE demonstrated that the rate of cooperation was not altered by adopting this approach.

Another change HE made was to make a tag represent the neighbourhood of the agent. When a new tag is learnt from another agent or through mutation, this new tag effectively causes the agent to rewire itself in a network so as to remove connections to those agents with its old tag, and forge connections to agents that share the same new tag. In the HE approach, donations are only permitted between agents in the same neighbourhood. Using this interpretation, a tag could be thought of as a gang sign, and agents would only have a chance to cooperate with those agents in the same gang.

3.4.4 Image Scoring

In 2008, the notion of tag-based cooperation was altered to observe how cooperation changes in the presence of self-interested agents [19]. The basic approach of RCA was combined with the learning interpretation of HE. In addition, all the agents were connected in a graph with a random network topology. Agents were only able to donate to the agents to which they are connected.

This model was used to observe the effect of self-interested agents (cheaters) on the donation rate. In this case, a cheater is an agent that always refuses to donate to other agents, despite the tag and tolerance values. As such a cheater is likely to be very successful, as they receive the benefits of donation without incurring the costs. It was shown that the presence of just a small amount of cheaters in the network caused a large reduction in the overall rates of cooperation.

In order to address this problem, an image scoring mechanism was introduced. Every agent observes the interaction pairings of its neighbours. Each agent keeps track of the times when a neighbouring agent chooses or refuses to make a donation. The observations for each neighbour are stored in a queue data structure. When an agent is selected to make a donation, they will consider the observations of their local network when donating in addition to their tag and tolerance values.

This approach to ranking a neighbourhood is chosen due to the lack of direct reciprocity. Direct reciprocity is when, by donating to another agent, it is likely that the other agent will in future donate to you. In these such networks however, direct reciprocity does not occur, it was therefore considered important to take into account the neighbourhood to which an agent belongs when they make a donation. The choice of an agent to donate is determined by the equation 3.11.

$$T'_A = T_A + \left(T_A \times \frac{\sum_{i=1}^{n} \delta_i}{n} \right)$$

Figure 3.11: Equation to determine if an agent is ready to donate, augmented with neighbourhood assessment

Until each agent has recorded enough observations, the basic tag and tolerance approach described by RCA is used.

It was observed that using this approach, a significant increase in the donation rate, particularly with small percentage of cheaters, could be achieved.

3.4.5 Network Rewiring

In order to further increase cooperation in the presence of cheaters, the ability for agents to rewire their connections was introduced [20]. This allows an agent to rewire their connections to other agents, in effect changing the neighbourhood of an agent. The reasoning behind this was to allow an agent to drop connections to the agents in their neighbourhood which are unlikely to donate, and replace these with connections to new agents which are more likely to donate.

Each agent which learnt during the learning phase, is given an opportunity to rewire their own connections. At this point, each of these agents is allowed to drop any number of its connections to other agents and then add connections to new neighbours. The aim of this is to allow agents to remove un-cooperative agents from their neighbourhood in order to increase cooperation in their network.

There are two parameters that are used to control the rewiring phase: the rewire proportion and the rewire strategy. The rewire proportion is a percentage which is used to determine the number of neighbours removed or added during a rewiring. When the rewire proportion is zero, then no rewiring takes place. When the rewire proportion is one, then all of an agents connections will be dropped and replaced with new connections. There is a single rewire proportion value for all agents, which remains constant throughout the whole simulation.

The rewiring strategies determine which of an agents' connections are dropped and which new connections are to be acquired. In their paper, Griffiths and Luck introduced four different rewiring strategies, these are: Random, Random Replace Worst, Individual Replace Worst and Group Replace Worst. For the purposes of the following descriptions of these strategies, it is assumed that each neighbour of an agent can be ranked in order of their observed past donation behaviour. This assumption is used to allow a rewiring strategy to identify the most and least cooperative agents in an agent's network.

Random Rewiring Strategy When the random rewiring strategy is used, each agent will drop a proportion of their connections at random. Once an agent has dropped these connections, the agent will then add new connections to new agents at random to replace the connections dropped.

Random Replace Worst Rewiring Strategy In this strategy, each agent will rank their neighbouring agents in terms of their perceived willingness to donate, based on past observations. Based on these rankings, the rewiring agent will drop their neighbours which have the worst donation rate track record. Once these connections have been dropped, random connections to new neighbours will be forged—as in the random rewiring strategy.

Individual Replace Worst Rewiring Strategy The individual replace worst strategy will—like the random replace worst rewire strategy—remove connections to

their neighbours which have the worst record of donations. After these connections have been dropped, the agent will look at their most willing neighbour and replace the lost connections with their neighbour's best ranked connections. In the case where adding a new connection would duplicate an existing connection, or connect an agent to itself, a random connection is added instead. The effect of this is that the new connections are the same as the best connections of their best neighbour.

Group Replace Worst Rewiring Strategy The group replace worst rewiring strategy is very similar to the individual replace worst rewiring strategy. Like the individual replace worst rewiring strategy, the neighbouring agents with the worst donation track record are dropped. Once these connections have been dropped, the agent will connect to the best ranked neighbour of each of its best ranked neighbours. As in the individual replace worst rewiring strategy, any possible duplicate connections will instead by replace by a random new connection.

3.4.6 Effects of Rewiring Strategy on Cooperation

To observe the effect of these different rewiring strategies as well as the rewire proportion on the donation rate, Griffiths and Luck performed a number of experiments.

It was found that the random rewiring strategy performed better than the RCA approach, but worse than the RCA approach augmented with context assessment. It was found that all the other rewiring strategies resulted in a better donation rate than either of the RCA approaches. It was found that rewiring was poorest when the rewiring proportion was close to zero or one. It was found that the random replace worst rewiring strategy performed poorly when the rewiring proportion was greater than around 0.6. It was found that the individual and group replace worst rewiring strategies performed similarly and had the best results when the rewire proportion was between 0.4 and 0.8.

From the results, it was show that by implementing these simple rewiring strategies, it is possible to get a significant increase in overall cooperation— in the paper, there was an increase of around 20% over the image-scoring approach in populations with 10%, 20%, and 30% cheaters.

From these results, it appears to be sensible to choose a rewire proportion in the range 0.4 to 0.8. In the paper, it was suggested to use a value of 0.6 for the rewire proportion.

3.5 Distributed Computing

With the huge sizes social networks can reach, there needs to be efficient approaches to decrease the running time of algorithms for analysis of the networks. One approach is to use distributed computing to parallelise these algorithms, so that computation across the network occurs at the same time, where possible, and as such should reduce the running time of these algorithms.

Listing 3.1: Calculating the frequency of words in files using MapReduce [13]

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, ``1'');

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

One area of distributed computing which has been growing in recently is the use of the MapReduce framework developed by Google, and the free implementation by Apache of MapReduce, Hadoop. We will be looking to use Hadoop as an approach to parallelise algorithms for social network analysis.

This should achieve a reduced running time for algorithms, and also allow analysis of far larger networks than previously possible due to the scalability of the Hadoop framework.

3.5.1 MapReduce

MapReduce is a programming framework designed to simply processing data on large clusters [13]. MapReduce works by the user supplying two functions, Map and Reduce, which operate in parallel on the data set provided. The Map function processes data from key/value pairs into an intermediate set of key/value pairs, before the Reduce function processes this intermediate set into final key/value pairs.

Listing 3.1 is an example MapReduce program which counts the frequency of words within a selection of documents stored on a system. The Map function reads each word, w, and emits the word as key/value pair `(w, ``1'')` to signify that there is an occurrence of w at that position. The Reduce function sums together the value each of these emitted key/value pairs, where the key is the same, and emits the sum for each word.

Whilst the example given in Listing 3.1 uses Strings for the input and output for both of the Map and Reduce functions, it is not necessarily the case that all Map and Reduce functions operate in this way. [13] explains that the types used by both are linked, as shown by:

```
map     (k1, v1)         -> list(k2, v2)
reduce  (k2, list(v2))   -> list(v2)
```

This states the input keys and values are from a different domain to the output keys and values, which also means that the types used can differ.

Google File System

The Google File System, GFS, provides the distributed file system which MapReduce operates with [13], but was developed outside of MapReduce to address issues found with previous distributed file systems [17].

The GFS was designed to meet three major points identified with existing distributed file systems [17]:

1. Component failures are the norm, rather than the exception
2. Files are huge by traditional standards
3. Files are mutated by appending new data

As hardware failures are common, the design of the GFS incorporates this, and the system is monitoring itself continually to detect, tolerate, and recover promptly from component failures on a routine basis [17]. In addition to this, the system used by Google makes use of inexpensive hardware due to the frequent failures experienced, and as such is a more cost-effective solution than using more expensive tailored hardware.

A GFS cluster is split into a single *master* and multiple *chunkservers* and is accessed by multiple *clients* [17]. Files stored in the GFS are split into chunks, which are stored across the cluster on the hard disks located on each chunkserver. By default, each chunk is replicated in the file system three times for reliability of access to data within file system.

Files are stored into the GFS in chunks of size 64MB. This size was chosen to reduce the need to interact with master to find the location of chunks to read data from, and write data to. The larger chunk size also reduces the quantity of metadata stored on the master, which increases the performance of the master as the metadata can be stored in memory, reducing lookup times [17].

The master node maintains the metadata for the file system. This includes the locations of chunks across the file system, and which chunks compose the files stored. The master node communicates with each chunkserver frequently, and if it does not receive a response, the chunkserver is deemed to have failed and any chunks which are then under replicated in the file system are re-replicated to ensure that the minimum number of replications for each chunk are observed.

3.5.2 Hadoop

Hadoop[4] is framework for performing distributed computing. It is a free implementation of the MapReduce framework developed at Google, and is also a top-level project hosted

[4] `http://hadoop.apache.org/`

by Apache.

Hadoop has diversified itself from its conception, and is now composed of three subprojects, Hadoop Common, Haddop Distributed File System, and Hadoop MapReduce. Hadoop Common providse common utilities which support the other Hadoop subprojects. The Hadoop Distributed File System is described in more detail in Section 3.5.2

Hadoop MapReduce is the subproject by which Hadoop itself more known for. It provides functionality similar to the MapReduce framework developed by Google, where there exists a *Map* and a *Reduce* function which process data across a cluster.

Hadoop Distributed File System

Hadoop also provides a the Hadoop Distributed File System, HDFS. The HDFS is a free implementation of the Google File System, and is designed to be used with Hadoop itself, though can also be used a distributed file system by itself [3].

The HDFS operates in a similar approach to the operation of Hadoop and the Google File System. There exists a master node, called the NameNode, and many slave nodes, called DataNodes. The NameNode co-ordinates the access of files stored in the HDFS, and also manages the file system namespace. There is usually a DataNode present on each physical node within the cluster. It is the DataNode which control the storage of files on the storage system present on the node, and also controls the reading and writing of files to the HDFS from a user.

The HDFS ensures data integrity through replication of data across different nodes within the cluster. A replication factor is set for the cluster, generally at least 3, which causes all data within the HDFS to be replicated at least that many times. Data within the HDFS is split into blocks, with a large file being represented by many smaller blocks, and it is these blocks which are replicated across the HDFS.

In case of a problem with the HDFS, such as a partial network failure, or hard disk failure, each DataNode is required to periodically message the NameNode which contains a report on all data blocks stored by that DataNode. If there is a failure of some kind, then the NameNode either receives an incomplete message or no message at all. This informs the NameNode that there is a problem with the HDFS and takes appropriate action, including re-replicating the lost data from other DataNodes to new DataNodes.

The HDFS also provides another service called the SecondaryNameNode. The SecondaryNameNode is not a direct failover service to the NameNode. Instead, the SecondaryNameNode takes periodic checkpoints of the state of the NameNode, so that in case of the NameNode failing, a recent copy of the state of the HDFS can be loaded when the NameNode is restarted, which should result in minimal problems with resuming the HDFS.

3.5.3 Graph Processing

Whilst MapReduce and Hadoop provide a way for most algorithms to perform in a distributed manner, not all algorithms can be expressed in the MapReduce paradigm easily, or suffer from problems of being adapted to MapReduce when a different paradigm would be more suited. Google developed a framework called Pregel [28], and an open-source development inspired by Pregel, called Giraph [8] began in 2011

Pregel

Pregel is a framework developed at Google for large-scale graph processing. The purpose of Pregel was to produce a scalable and fault-tolerant platform with an API that is sufficiently flexible to express arbitrary graph algorithms [28]. Existing solutions, other than Pregel, which achieve large-scale graph processing do suffer from issues ranging from poor performance, to a lack of fault tolerance within the system. Pregel addresses these issues within its framework.

A Pregel program executes as a series of parallel supersteps, similar in style to the Bulk Synchronous Parallel computation model described in [40]. At each superstep, each active vertex performs some computation, which could modify the state of itself, its outgoing edges or send messages to other vertices to be received in the next superstep. An active vertex can also *VoteToHalt*, which stops this vertex it from performing any extra computation whilst the algorithm is being executed, unless the inactive vertex receives a message, in which case it becomes active again to process the messages it has received. When all vertices have voted to halt, the algorithm terminates.

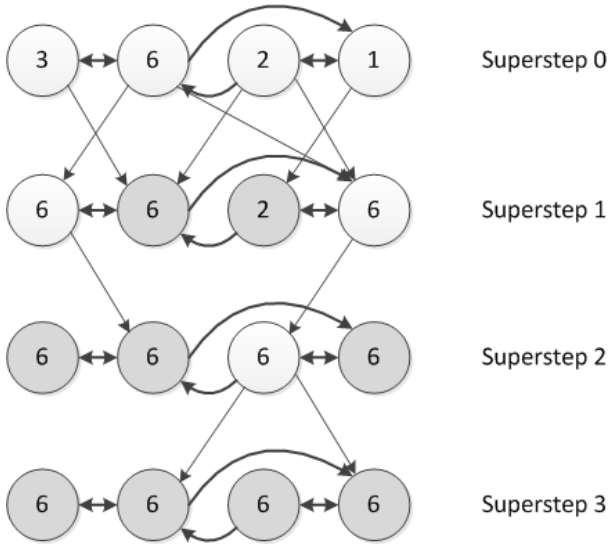


Figure 3.12: Maximum Value Example. Lighter lines are messages. Shaded vertices have voted to halt [28].

Figure 3.12 shows an example program in Pregel, which finds the maximum vertex value. Initially, all vertices are active, and send a message to each vertex they are connected

directly containing their value. In the next superstep, if a vertex receives a message with a higher value than its currently stored value, then it stores the new higher value, otherwise the vertex deactivates as it currently has a local maximum whilst the program has not terminated. The vertices which did change send their value send a message containing their updated value to their neighbours. In superstep 2, the third vertex activates and changes its value from 2 to 6 as it receives a message containing the value 6. The two active vertices deactivate because they did not receive a message. The third vertex then sends a message to the second and fourth vertices containing the value 6, and in the third superstep, no vertex receiving a message changes its value, so does not need to send a message resulting in all vertices becoming inactive and terminating the program. The maximum value of all the vertices within the graph is now stored in all vertices.

As a Pregel program begins execution, multiple copies of the program start executing on a cluster of machines. A single copy is assigned as the *master* which does not perform any work for the actual program, and instead manages the remaining *worker* copies of the program. The *master* partitions the input graph across the *workers* and instructs the *workers* to perform a superstep. This causes one iteration of the program to occur, and is repeated until all vertices vote-to-halt. The *master* can the instruct each *worker* to save its partition of the graph to memory.

The use of an underlying Bulk Synchonous Parallel model provides the Pregel framework with a simple way to model iteration of graph algorithms, and also provides a simple way to achieve a fault tolerant system. Each superstep in the execution of a Pregel program provides a natural barrier to synchronise each vertex's computation. If a *worker* fails to communicate with the *master* within a defined time period, the *master* assumes that the *worker* has failed, and can restart the program from the checkpoint made at the previous superstep completion, re-partitioning the graph if the failed *worker* is still unavailable.

The Pregel framework also provides some extra useful features called combiners and aggregators. A combiner groups together a series of messages from one vertex to another into a single message containing the result of an operation on these messages. This is to reduce the overhead incurred from sending many messages and then performing the same operation at the receiving vertex. An aggregator is a global data construct, which each vertex has access to its value at each superstep, and the value can be updated at the end of each superstep in time for the next superstep. Aggregators can be used to store information about a graph, such as the number of edges in the graph.

Pregel also supports mutations of the graph an algorithm is being executed on. There are two types of mutations, global and local. A global mutation includes the addition and removal of vertices, as these can affect more that just a single vertex. Global mutations are partially ordered, and the effects of these mutations are seen in the next superstep. A local mutation includes a single vertex adding or removing an edge to another vertex. As these only affect the vertex which is performing these mutations, they happen immediately without any conflicts.

Listing 3.2 shows how the PageRank algorithm [34] can be written using the Pregel framework. It can clearly be seen how the algorithm works. Each vertex receives a series of messages which contain the PageRank for the vertex which sent the message. The sum

Listing 3.2: PageRank algorithm in Pregel [28]

```
class PageRankVertex
      : public Vertex<double, void, double> {
  public :
    virtual void Compute(MessageIterator* msgs) {
g     if (superstep() >= 1) {
        double sum = 0;
        for (; !msgs->Done(); msgs->Next())
          sum += msgs->Value();
        *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
      }

      if (superstep() < 30) {
        const int64 n = GetOutEdgeIterator().size();
        SendMessageToAllNeighbors(GetValue() / n);
      } else {
        VoteToHalt();
      }
    }
};
```

of these PageRanks is then computed, and the damping factor applied, and the result is stored as the value of the vertex. This values is then divided by the number of outgoing edges from the vertex, and is then sent as a message to each of the vertices the vertex is connected to.

Giraph

Giraph is a graph processing framework, inspired by Pregel, initially developed at Yahoo!. It has since become an Apache Incubator project[5]. Giraph is still under development, with more features being added and improved. There are a number of existing solutions to achieve large-scale graph processing, however these also have their own problems.

Graph algorithms can be executed as a sequence of map-reduce jobs in Hadoop, but this suffers from the overheads of repeatedly launching these jobs and the map-reduce model is not a good fit for graph algorithms. Pregel itself has problems in that it requires a separate computing infrastructure, and is also unavailable to non-Google employees. The Message Passage Interface can also be used for graph processing, however it lacks any form of fault tolerance and is consider too generic [8].

Listing 3.3 shows the PageRank algorithm [34] produced using Giraph. Strong comparisons can be seen between this, and the Pregel implementation in Listing 3.2. Both Listings show that a vertex receives a series of messages, of which the values of each are

[5] `http://incubator.apache.org/giraph/`

Listing 3.3: PageRank algorithm in Giraph [8]

```
public class SimplePageRankVertex extends
    HadoopVertex<LongWritable, DoubleWritable, FloatWritable,
    DoubleWritable> {

  public void compute(Iterator<DoubleWritable> msgIterator) {
    double sum = 0;

    while (msgIterator.hasNext()) {
      sum += msgIterator.next().get();
    }

    setVertexValue(new DoubleWritable((0.15f/getNumVertices()) +
        0.85f * sum);

    if (getSuperstep() < 30) {
      long edges = getOutEdgeIterator().size();

      sendMsgToAllEdges(new
          DoubleWritable(getVertexValue().get() / edges));
    } else {
      voteToHalt();
    }
  }
}
```

summer together. A damping factor is then applied and the result stored as the value of the vertex, before this value divided by the number of outgoing edges is sent to each vertex this vertex is connected to.

Small differences can be seen, due to Giraph using Hadoop as an underlying component, such as the type system used in Giraph being composed from Java objects as opposed to primitive date types in Pregel.

There are advantages to Giraph being built on top of Hadoop. Hadoop is being used by a large number of organisations for educational and production uses[6] with other organisations providing Hadoop services for users[7]. This means that Giraph can be deployed onto these clusters with minimal effort required. It also means that Giraph only suffers from Hadoop-based problems, such as the Hadoop namenode and jobtracker being single points of failure for a Hadoop cluster.

Being inspired by Pregel, Giraph also makes use of the Bulk Synchronous Parallel computation model [40]. Again for similar reasons to Pregel, the Bulk Synchronous Parallel model provides natural barriers for checkpoints during the execution of programs, and programs can restart from a previous superstep in the case of any failure. Giraph also operates with one copy of the program being designated the *master* and the remaining copies being designated as *workers*. The *master* co-ordinates the *workers*, and also handles the distribution of the input data across the cluster. The *workers* execute the `compute()` method for every vertex on the partition of data they receive.

Giraph operates as a Map-only job on Hadoop, with there being no Reduce stage. A Giraph job has its own InputFormat and OutputFormat classes which call the user defined Vertex versions of these classes to load the data into each of the *worker* processes.

The basic construct of a Giraph program is similar in style to a Hadoop program. Table 3.1 shows this similarity. The `compute()` method of Giraph is analogous to the `map()` method of Hadoop. The I, V, E and M represent datatypes to be chosen by the user for their own implementation of Vertex, where:

- I → VertexId
- V → VertexValue
- E → EdgeValue
- M → MsgValue

Giraph only deals with directed graphs, undirected graphs are represented with edges connecting vertices in both directions.

[6] `http://wiki.apache.org/hadoop/PoweredBy`
[7] `http://wiki.apache.org/hadoop/DistributionsandCommercialSupport`

Hadoop	Giraph

public class Mapper<
KEYIN,
VALUEIN,
KEYOUT,
VALUEOUT> {
void map(KEYIN key,
VALUEIN value,
Context context)
throws IOException,
InterruptedException;
}
	public class Vertex<
	I extends WritableComparable,
	V extends Writable,
	E extends Writable,
	M extends Writable> {
	void compute(
	Iterator<M> msgIterator);
	}

Table 3.1: Comparison between basic Hadoop and Giraph program construct [8]

Chapter 4

Specification

4.1 Legal Issues

We used public data from twitter so direct issues of confidentiality and data-protection were avoided.

4.1.1 Licensing

We used the following software tools and packages during this project: Oracle Java[1], Apache Collections[2], Apache Hadoop[3], Apache Giraph[4], Apache Ant[5], Apache Maven[6], JUNG[7], Perl[8], Python[9], Cytoscape[10], and SQLite[11]. Each of these packages have permissive licences. The Apache projects are licensed under the Apache licence. JUNG is licensed under the BSD licence. In accordance with these licences, a copy of these licences is included on the attached CD. Part of the project is based on last years project which is licensed according to the LGPL which is a copy-left licence, the code produced in this project is also licensed under the LGPL. A copy of this licence is also provided on the attached CD.

[1] http://www.java.com/en/
[2] http://commons.apache.org/collections/
[3] http://hadoop.apache.org/
[4] http://incubator.apache.org/giraph/
[5] http://ant.apache.org/
[6] http://maven.apache.org/
[7] http://jung.sourceforge.net/
[8] http://www.perl.org/
[9] http://www.python.org/
[10] http://www.cytoscape.org/
[11] http://sqlite.org/

4.1.2 Terms Of Use

Two stipulations of the twitter API were relevant to us:

'You will not attempt or encourage others to: E) use or access the Twitter API to aggregate, cache (except as part of a Tweet), or store place and other geographic location information contained in Twitter Content.'

We don't violate this as we only store geographic information in tweets, but future extensions should be wary of it.

'Your service should not: impersonate or facilitate impersonation of others in a manner that can mislead, confuse, or deceive users.'

Again, we don't violate this but stray close enough to it in the intended use-case of our software.

https://dev.twitter.com/terms/api-terms https://twitter.com/tos

4.2 Ethical Issues

4.1

4.2.1 Surveillance

The primary ethical issue with this project is that it is ultimately about surveillance. As someone walking a public street having no reasonable expectation of privacy doesn't quite excuse watching their every step with CCTV cameras, a user making a post on Twitter doesn't quite justify merging it into a database in an attempt to model their relationships. This issue is exacerbated by people's tendency to view posts made on public forums as being as off-the-record as a spoken conversation with a close friend.

While it could be argued that this is merely a matter of user education, what would the result of such 'education' be if not stifled, warily self-conscious communications? Who wouldn't think twice about conversing with someone known to have Interesting views, and what would the effect of such isolation be on an already borderline extremist?

4.2.2 Intended Use

The application of this information may have equally chilling consequences. A simplistic view would be to claim that ultimately this is a tool that could be used for good or bad and as such the responsibility lies with the user. In fact, the Computer Misuse Act has something to say about 'dual use articles'

'In determining the likelihood of an article being used (or misused) to commit a criminal offence, prosecutors should consider the following:

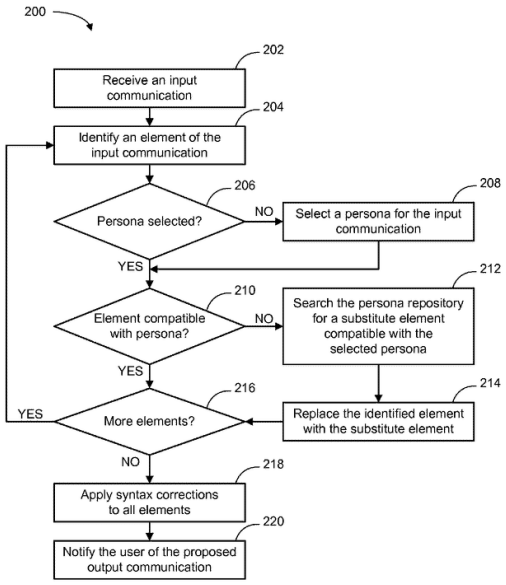


FIG. 6

Figure 4.1: Persona Management

Has the article been developed primarily, deliberately and for the sole purpose of committing a CMA offence (i.e. unauthorised access to computer material)? Is the article available on a wide scale commercial basis and sold through legitimate channels? Is the article widely used for legitimate purposes? Does it have a substantial installation base? What was the context in which the article was used to commit the offence compared with its original intended purpose?'

http://www.cps.gov.uk/legal/a_to_c/computer_misuse_act_1990/#an10 http://www.lightbluetouch
tool-guidance-finally-appears/ http://www.guardian.co.uk/technology/2011/mar/17/us-spy-operation-social-networks

4.2.3 Cracking 'protected' users

There was one boundary that we consciously decided not to cross while harvesting information from Twitter. A minority of users mark their accounts as 'protected', which means that no information on their account is public. One has to send them a follow request, and then only if it is accepted will one be able to view their data. These users naturally disrupt the graph somewhat, and there are a couple of ways we could have tackled them. A fairly ethical but ineffective approach is to collate and store the information we indirectly gained on them. The alternative was to manipulate these users into accepting follow requests from our bot. Given that all our data gathering is done through a single user account, it would have been a simple matter to make the account masquerade as a member of the target group. Simply follow the most central users, and retweet the most popular tweets from those users. This would ensure that the account passed cursory examination of its 'following' and tweets. This would just leave 'followers' which would be easily acquired by 'followbacks'; it is a common practise to follow someone who follows you, and users who follow this practise are easily identified because they are following more people than are following them. So obtaining followers would simply be a matter of identifying these users and following them. None of this would have been a tricky addition to the existing code, but the not insignificant ethical concerns of automating impersonation on this scale prevented us.

Chapter 5

Design

5.1 System Architecture

Figure 5.1 contains the intended architecture of DSNAT, and how various components relate to it.

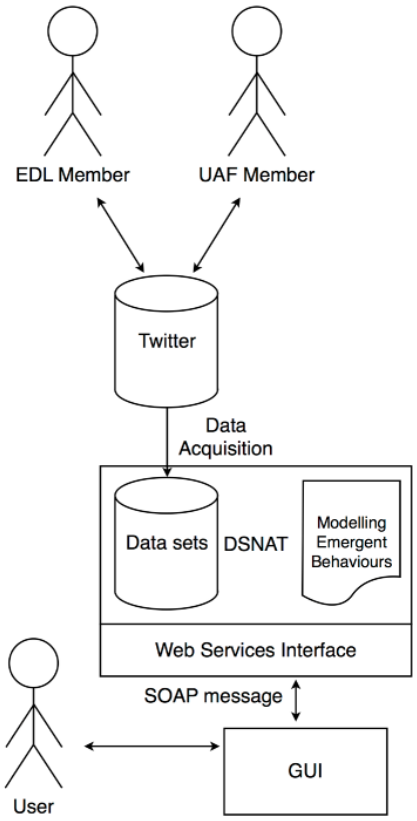


Figure 5.1: DSNAT system architecture.

The GUI component of the system is being reused from the social network analysis tool [36] created in the previous year.

5.2 Data Acquisition

5.2.1 Twitter

Our requirements were as follows:

- Sufficient data per item to construct a graph
- Temporal data for use in last year's interface
- Sufficient nodes to illustrate the clusterâĂŹs usefulness
- Modern, real-world data

The social networking site Twitter met all these requirements. As an added bonus, it had information on two groups our customer has an interest in; the English Defence League (EDL) and Unite Against Fascism (UAF). Twitter is easily mapped onto a directed graph as it uses an asymmetric following system where user X can follow user Y without user Y having to follow user X. In addition to collecting the following/followers lists for each user we store the messages they broadcast, their 'tweets'. Tweets may be addressed to particular users, and can thus be modelled as edges themselves. In addition to normal, hand-typed tweets there are 'retweets' where the user has essentially endorsed another userâĂŹs tweet by repeating it themselves. These can likewise be modelled as directed edges. Along with the text content of a tweet comes a significant amount of metadata such as its timestamp, a unique id, who itś addressed to and sometimes geolocational data. Followers have much less information available on them; there is no way to directly tell when a user followed another, or whether a user used to follow another.

Twitter API

Twitter provides two main interfaces for developers wishing to harvest data from it; a Representational State Transfer (REST) API and a streaming API. REST is a stateless protocol similar in style to the ubiquitous HTTP/1.1. As a result of prior abuse and overuse The REST API is heavily rate limited. According to the documentation non authenticated calls are limited to 150 requests per hour per IP, and OAuth-authenticated requests are limited to 350 per hour per account. Each of the exposed functions have strict limits on the amount of data they can return; the 'statuses/user_timeline' will only return the 3,200 most recent tweets of a user, and no more than 200 per request. This places a hard limit of 200x350/3200 20 user's tweets per hour.

The initial plan was to use the streaming API since it has higher throughput more generous limits and is thus more suited to the quantity of data we wanted to obtain. However, after significant delays and repeated failures to get added to the whitelist we resorted to using the REST API with OAuth for the increased limit.

As a result of the rate limits it will need to be run for time in excess of several days to obtain a substantial amount of data, making it unsuitable for use on our desktop computers. We needed a stand alone script that could be deployed anywhere with minimal

setup. SQLite was selected as a lightweight database solution that was easily setup and moved.

Target Identification Algorithm

We considered two approaches to identifying members of the target groups; manually identifying a central node and mapping outward from there, or mapping all users who use associated hashtags such as #UAF. However, preliminary research revealed that many tags have multiple meanings; UAF is commonly used to mean 'ugly as f***'. Between tag overloading and news agencies using the #EDL tag on stories about the english defence league, tags would not have been a substantive foundation for the basic mapping algorithm. Instead, we let the operator specify a central node (easily found via Google's graph-based ranking algorithm) and the algorithm performs a breadth-first-search from there:

```
queue.add(startingUser)
while(True)
        target = queue.pop()
        for each user follower/following of target
                queue.add(user)
        write target\'s tweets/followers/following to db
        write program state to db
```

Given the low rate at which we could accumulate data, we decided to store all of it to allow for further, unforeseen uses. Simple scripts could then be used to extrapolate graphs in graphml format for our own use, and in MySQL suitable for use in last years visualiser.

5.2.2 Enron

5.2.3 Stanford Large Network Dataset Collection

Stanford University host a project called *Stanford Network Analysis Platform*, SNAP.

> "SNAP is a general purpose network analysis and graph mining library. It is written in C++ and easily scales to massive networks with hundreds of millions of nodes, and billions of edges. It efficiently manipulates large graphs, calculates structural properties, generates regular and random graphs, and supports attributes on nodes and edges."[1]

In addition to this library, the SNAP project also hosts a collection of large datasets in the *Stanford Large Network Dataset Collection*[2]. This collection of datasets includes a variety of different networks, examples including social networks, communication networks, citation networks, collaboration networks and web graphs.

[1] `http://snap.stanford.edu`
[2] `http://snap.stanford.edu/data`

The majority of these datasets are of the form $NodeId \rightarrow NodeId$, which indicates that these networks cannot be analysed from a temporal perspective. However, the sheer size of some of these networks, notably a LiveJournal network containing 4,847,571 nodes and 68,993,773 edges, and a network of Twitter users from 2009 [25] containing 41.7 million nodes and 1.47 billion edges, should prove DSNAT's viability in performing analysis on large-scale social networks.

5.3 Hadoop Cluster

5.4 SOAP

An important part of our project was connecting our code running on the Hadoop cluster to the C# visualiser created by last year's project group. Therefore, we had to define a means of communicating between these different modules – as these modules would be running on different machines, this interface had to be exposed as a web service. We chose to implement this communication interface in SOAP; the reason for this choice is detailed below.

5.4.1 Choice of SOAP over REST

[http://www.infoq.com/articles/rest-soap-when-to-use-each]

The two leading candidates for the communication interface were REST and SOAP. These are usually compared side by side, although strictly speaking SOAP is a communications protocol wheras REST is more of an architectural pattern that sits on top of HTTP.

Advantages of REST

REST, short for REpresentational State Transfer, is an architectural pattern for using HTTP as a web service. The motivation behind REST is that HTTP already contains a rich vocabulary for manipulating resources – GET, POST, PUT and so on – and thus, the logic goes, avoid reimplementing these commands in a complicated new protocol such as SOAP.

[http://www.ajaxonomy.com/2008/xml/web-services-part-1-soap-vs-rest]

The benefits of REST, as defined by [ajaxonomy] and [infoQ], are:

- It is language and platform agnostic.
- It is simpler than SOAP, as it does not require an additional messaging layer.
- It is closer to the philosopy of the web as originally defined by it's creator, Tim Berners-Lee.

- It is able to make use of the built-in capabilities provided by the HTTP protocol.

Advantages of SOAP

SOAP, short for Simple Object Access Protocol, is a protocol designed for exchanging structured data via XML. Normally the SOAP framework will abstract away the XML from the end user, using a remote procedure call (RPC) -based interface. An RPC interface is exposed to the application programmer in a simple way; they define a function or method on the server side, and then call that function on the client side. Creating and parsing XML messages is handled by the SOAP library used.

Unlike REST, SOAP can run on top of any transport layer – this may be HTTP or something else.

The benefits of SOAP [same sources as above], are:

- Unlike REST, it enables the developer to define their own vocabulary of verbs to suit the target domain.
- It is language, platform *and* transport agnostic.
- It is designed to handle distributed computing environments.
- It is the predominant standard for web services, meaning there is a greater level of support in different programming languages.
- It features built-in error handling.

For these reasons, we chose to use SOAP as the communications protocol.

5.4.2 Architecture Overview

[Insert Diagram Here]

The SOAP communication module followed a standard client-server model. The SOAP server (written in Python) would run on the same machine as the Hadoop head node. Python's subprocess module, which lets shell commands be executed from within Python scripts, would be used to allow the SOAP server to control the Hadoop cluster.

Two SOAP clients were written. One was a command line client, also written in Python. The second was the C#-based visualiser written by last year's project group, which would be modified to send SOAP commands.

During the implementation of the server, one major change to the design proved neccesary; this was modifying it so that it worked in a multi-threaded manner, with one Hadoop task being ran per thread. The reason for this change is detailed in the implementation section below.

Other changes that were made during the design phase included sending large files as base 64-encoding strings, rather than byte arrays, and giving the server the ability to execute algorithms as Python scripts as well as JAR files.

5.4.3 Message Structure

We decided that the API needed to support the following commands:

- `upload_data_set`

  Arguments data_set_name (string), data_set (string)

  Returns String

  Description Takes an data_set (as a base 64-encoded string), and writes it to a file in the snat_data_sets directory on HDFS. Returns success/failure message.

- `upload_algorithm`

  Arguments algorithm_name (string), class_name (string), jar_file (string)

  Returns String

  Description Takes an jar file (as a base 64-encoded string), and writes it to a file in the algorithms directory (on the regular file system). It also writes to a text file that keeps of a record of all the uploaded algorithms. Returns success/failure message. (Note: although the SOAP server can execute algorithms as either JAR files or Python scripts, currently only JAR files can be uploaded).

- `get_algorithms`

  Arguments None

  Returns List[String]

  Description Returns a list of all the algorithms contained in the algorithms directory.

- `get_data_sets`

  Arguments None

  Returns List[String]

  Description Returns a list of all the data sets contained in the snat_datasets directory.

- `execute_algorithm`

  Arguments algorithm_name (string), data_set_name (string), num_nodes (integer), command_line_args

  Returns Integer

Description Looks up the algorithm named algorithm_name. If it references a JAR file, run a Hadoop job with that JAR. If it references a Python script, run that script. All jobs are executed in background threads; returns an integer identifying the thread.

- `get_results`

  Arguments thread_id (integer)

  Returns String

  Description Takes a thread_id referencing a Hadoop job running in the background. If the job has terminated, returns the output as a base 64-encoded string.

- `show_statys`

  Arguments required_data (string)

  Returns String

  Description Currently, takes a port number for one of the Hadoop logging pages running on localhost, and returns the contents of that page.

5.5 GUI

5.6 Emergent Behaviour

The simulation will be written in Java. Java was chosen for many reasons, such as:

- it is available on all of the most popular computing platforms;
- it produces cross-platform applications;
- it is easy to package dependencies in one bundle;
- there is a wealth of third-party libraries and tools which can be used.
- it has good performance, which is necessary given the scale of the simulations.

The simulation will use the Java Universal Network/Graph Framework (JUNG), as the base for the graph implementation. JUNG is an open-source graph library and contains a lot of data structures for representing graphs and routines for manipulating the graphs. JUNG was chosen for many reasons, including:

- it is able to generate many different types of graphs;
- it is mature, and has no bugs which would impair the results;
- writing bespoke code to handle graphs would be challenging to get right;
- it includes many useful algorithms that can be useful in analysing the graphs.

A 'wrapper' class will be written around the JUNG `graph` class in order to introduce additional functionality that will be required in the simulation.

Graphs will be generated using JUNG's random graph generators.

Every vertex in the graph will represent an agent in the simulation. To allow individual vertices to be randomly access in a fast manner, each vertex will have a unique identifier and be stored in a lookup data-structure.

Each agent will have a tag and tolerance instance variable to be able to represent an agent in the graph. Every agent will also have a `score` instance variable which will be used to keep track of the success of the agent. When the agent donates, the cost of donation, c, will be subtracted from the agent's score variable. When the agent receives a donation, the benefit, b, will be added to the agent's score variable. This approach allows the success of a given agent to be compared to the success of another agent by comparing the score variables of each agent.

A class will be created to control the parameters of a simulation—such as the number of generations to simulate, or the number of interaction pairings per agent per generation—and to allow multiple simulations to be run with different parameters in order to compare the results.

Parameters which will be able to be set are:

Name	Type	Default Value	Remarks
Generations	Integer	1,000	Must be greater than 0
Cost	Double	0.1	Must be greater than 0
Benefit	Double	1	Must be greater than Cost
Context Influence	Double	0.6	Must be in $[0, 1]$
Learn Probability	Double	0.1	Must be in $[0, 1]$
Rewire Proportion	Double	0.5	Must be in $[0, 1]$
Cheater Proportion	Double	0.1	Must be in $[0, 1]$
Mutation Probability	Double	0.01	Must be in $[0, 1]$
Graph Type	Graph Type	Random	Null values are not permitted
Rewire Strategy	Rewire Strategy	Null	Null values are permitted

The simulation will be started by passing in a set of parameters. The simulation will then generate a graph and set the initial state of the simulation according to the parameters passed in. The simulation will then be run.

while iterations remain **do**
    Donation()
    Reproduction()
end while

Vertex
id:int
getId():int

Agent
tag:double
tolerance:double
score:double
getTag():double
setTag(tag:double)
getTolerance():double
setTolerance(tolerance:double)

5.7 Algorithms

The algorithms which are implemented for DSNAT will be stored on the cluster for use via the web services interface.

Giraph provides a comprehensive API to implement distributed graph algorithms, because of this we do not see a need to implement a new API to act as a basis for implementing algorithms.

For algorithms which are difficult to express using Giraph and its vertex oriented design, we are able to fall back to using a Hadoop-based approach for implementation.

5.7.1 Giraph

3-Cliques

For simplicity reasons, we are only concerning ourselves with identifying 3-cliques, or *triangles*, within graphs. Also, the identification of triangles within a graph allows more interesting metrics of a graph to be computed

Identifying triangles using Giraph is fairly simple. The nature of Giraph allows the structure of the graph itself to be used identify triangles. The algorithm to identify triangles uses three supersteps to output only identified triangles. A triangle can be thought of as a triple $\{x, y, z\}$, where $x, y, z \in \{vertexIDs\} \wedge x < y < z$. A connected triple is $\{x, y, z\}$, where $x, y, z \in \{vertexIDs\}$, which also shows the number of triangles in the graph can be no greater than the number of triples.

In the first superstep, each vertex constructs a list of connected vertices which have a vertex ID lower than itself. It then sends this list of vertices to each connected vertex which has a larger vertex ID than itself. In the second superstep, each vertex processes the list of vertices it receives. Using the described triangle representation, $\{x, y, z\}$, a vertex receiving a list of vertex IDs in this superstep represents z, the vertex sending the

list represents y, and each vertex ID in the list represents an instantiation of x. We know $\{x, y, z\}$ exists as a triple centered on y, so the vertex receiving the list checks existence of an edge between itself and each vertex in the list. If there is an edge, then $\{x, y, z\}$ represents a triangle within the graph [7]. The final superstep simply removes all edges which do not form a triangle within the graph.

The subgraph containing only triangles from the input graph is the output from the algorithm.

Local Clustering Coefficient

The local clustering coefficient algorithm makes use of a slightly modified triangle identification algorithm. After identifying the presence of a triangle, the vertex z sends a message to each of the vertices x and y in the triangle they form, indicating they are present in a triangle. In the following superstep, each vertex sums the number of messages it receives to calculate the number of triangles it forms. In the same superstep, the number of connected triples each vertex is centered on is computed. This is done by calculating the number of distinct pairs of vertices each vertex is connected to. The local clustering coefficient for each vertex can then be computed by dividing the number of triangles the vertex composes by the number of triples it is centered on.

The output from this algorithm is the original graph, but with the local clustering coefficients stored at the vertices.

Network Local Clustering Coefficient

Calculating network local clustering coefficient is a simple extension to the local clustering coefficient algorithm. After computing the local clustering coefficients for each vertex, these values need to be aggregated into a single value, which is then divided by the number of vertices present in the graph.

Giraph makes this quite simple to achieve through use of the aggregators (Section 3.5.3). A *summation* aggregator can be used to receive all the local clustering coefficients from each vertex, and sum them into a single value. The sending of the local clustering coefficient to the aggregator occurs in the same superstep as the final computation of the local clustering coefficient. In the next superstep, this aggregated value can be retrieved and divided by the number of vertices present to compute the network local clustering coefficient.

PageRank

Giraph itself comes included with it an implementation of the PageRank algorithm, as shown in Listing 3.3. In each superstep, until termination criteria are met, each vertex sums the values of the PageRanks sent to it and applies the damping factor to produce the new PageRank for itself. This new PageRank divided by the number of outgoing

edges the vertex has is sent to each vertex it connects to. With the provided Giraph implementation, the damping factor has been set to 0.85, which could be user definable, but is normally set to 0.85 [5]. The halting criteria is also set to end the algorithm after 30 supersteps. Whilst this will produce reasonably accurate figures, a better approach is to keep the algorithm going until the PageRanks for each vertex converge to a value within some tolerance.

5.7.2 Influence Propagation

The influence propagation algorithm studied were conceptually simple. However, implementing them in a distributed computing environment like Hadoop proved complex. One major reason for this was that graph algorithms are inherently hard to parallelize.

Hadoop works best for "embarrasingly parallel" problems, that is, tasks which can be broken up into subtasks where there is little dependencies between individual tasks. Graph problems such as influence propogation are inherently about dependencies, as graphs express the relationships between different pieces of data. At some point any graph processing algorithm run in a distributed environment must parition the graph into portions to be run on each indiviual node, and then decide how to communicate between these nodes without causing too slow a bottleneck.

The Pregel framework is one solution to the problem of running graph algorithms on top of Hadoop. As we had used already Pregel for some graph processing problems, we decided to implement the influence propagation algorithms in vanilla Hadoop, to explore the difficulty of implementing graph algorithms without a specialised framework.

Independent Cascade

The first step towards implementing the independent cascade algorithm on Hadoop was realising that it was best implemented as multiple tasks, rather than a single task.

Specifically, a simple way of implementing independent cascade is to first "roll all the dice" – that is, decide if influence would successfully pass between each pair of connected nodes – and then run a standard connected components algorithm.

Therefore, the algorithm was split into two tasks, one to handle the "dice rolling", one to handle the connected components. The first task would be run once, whereas the second task would be run iteratively, with the output of one step being fed into the next step. More details can be found below.

Alternatives [http://blog.piccolboni.info/2010/07/map-reduce-algorithm-for-connected.html] One alternative way to implement the connected components task was using the PRAM algorithm, as described in [cite]. In theory this allows the connected components of a graph to be found in less iterations.

5.7.3 Linear Threshold

The Linear Threshold implementation follows the same basic pattern as the Independent Cascade implementation, though it only requires one task, which is performed iteratively. Each iteration corresponds to one iteration of the LT algorithm; that is, it counts all the converted neihbours of each unconverted node, and if it is greater than the node's threshold value, the node becomes converted.

More details of the implementation of both models can be found in the implementation section.

Chapter 6

Implementation

6.1 SOAP

6.1.1 Choice of Language/Library

We initially planned to use Java for the SOAP server. Java has a robust SOAP implementation in the form of the Apache Axis2 project. [http://axis.apache.org/axis2/java/core/index.html]. However, this project was highly configurable at the expense of simplicity. For example, the project features five different methods to deploy SOAP services.

We therefore decided to implement the SOAP server in Python, using the rpclib library for the server, and the suds library for the client. This allowed both the server and the client to be implemented within a single Python script each, leading to easier development and deployment.

6.1.2 Implementation

Implementing read-only commands such as show_status, get_data_sets, and get_algorithms proved straightforward. show_status merely echoed the data from one of Hadoop's built-in logging pages, get_data_sets returned the content of the snat_datasets folder on HDFS, and get_algorithms returned the algorithm names from a predefined algorithms.txt file.

One issue that arose with the implementation of upload_algorithm and upload_data_set was the best way to send large files over SOAP. The recommended way was to use the class rpclib.model.binary.ByteArray, which sends raw binary data. However, in practice there were issues retrieving the correct data as it was sent. We therefore instead chose to send data using base64 encoded strings.

The subprocess library was used to enable shell commands to be ran from Python, allowing the SOAP server to control the Hadoop cluster. The original design had execute_algorithm run the required jobs and then return the results directly. However,

for long running Hadoop jobs, the SOAP socket would close before any data could be sent. Reconfiguring the client and server to have a longer timeout period did not fix the problem.

Instead, it was decided to run Hadoop jobs via background threads. This was implemented as a dynamically-sized thread pool, where a new thread would be added for each job. execute_algorithm would no longer return results directly, but would return an integer corresponding to the id of the thread running that particular job.

A new api call, get_results, was added. If the relevant job had terminated, this call would return the results from that job. If not, an error message would be returned.

6.2 Hadoop Cluster

6.2.1 Requirements

Hadoop itself does not have any explicit hardware requirements, and is able to run on mainst

Hadoop also very few software requirements and is able to run on both Microsoft Windows and varieties of GNU/Linux. Use of Hadoop on a Windows environment is recommended only as a development platform, as Hadoop has not been well tested as a production platform for Windows. Both development and production have been tested using GNU/Linux environments, and is the recommended for use of Hadoop.

Hadoop only has two explicit software requirements: Java[1] and ssh[2]. Hadoop launches each of the services required to manage a cluster as Java processes, and as such requires the Java Runtime Environment 1.6.x to be present. As programs developed using Giraph are written in Java, the Java Development Kit is also required, and includes the Java Runtime Environment. Ssh is required, with sshd running on all nodes, to allow the NameNode to manage all nodes within the cluster.

Giraph has a few extra requirements of its own. It requires specific versions of Hadoop, Hadoop 0.20.203 or higher, which have security changes applied to them. At the time of the project commencement, Hadoop 0.20.205 had been released[3] as the latest version with the required security changes, and as such has been the version with which development of the project has been made with. During the development of the project, Hadoop reached version 1.0.0[4] but we chose to remain with the existing version of Hadoop so as not to introduce any conflicts with any possible deprecations or changes brought about by the newer version.

Giraph also requires Maven 3[5] to be installed to compile the Giraph source code into a

[1] http://www.java.com/
[2] http://www.openssh.com/
[3] 17th October 2011 - http://hadoop.apache.org/common/releases.html
[4] 27th December 2011 - http://hadoop.apache.org/common/releases.html
[5] http://maven.apache.org/

usable JAR for submitting jobs with to Hadoop.

6.2.2 Single-Node Cluster

For development purposes single-node clusters were created, using a guide written by Noll in [33]. Figure 6.1 shows the structure of a single-node cluster. All processes for Hadoop are required to be running on this single node, and as such hampers performance.

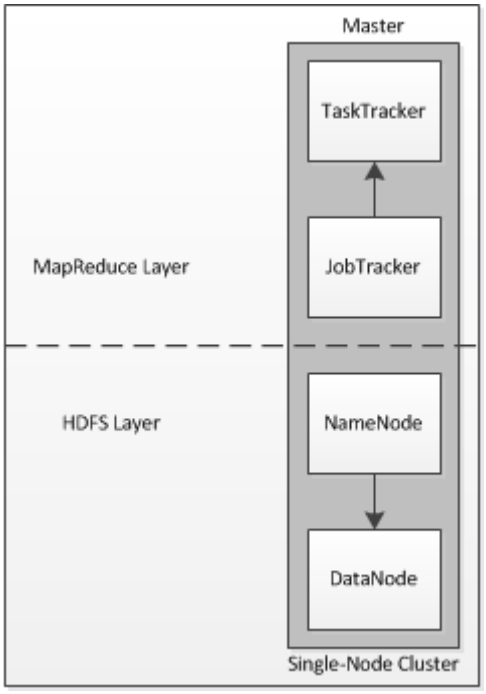


Figure 6.1: Single-node Hadoop cluster [33].

As a Hadoop cluster is trivially scalable, the performance of this single-node cluster is unimportant as if the program runs and produces the correct output for a small input, it should also run and produce the correct output on a much large cluster with a much larger input.

6.2.3 Multi-Node Cluster

At request, a nine machine Hadoop cluster was set up for use during this project in the department of computer science. This cluster composed a master head-node, `hadoopmaster`, and eight slave-nodes, `hadoop{0-7}`.

Being the master node, `hadoopmaster` runs the NameNode, JobTracker and SecondaryNameNode processes. A dedicated master node was implemented, firstly for performance, so that the node running the NameNode, JobTracker and SecondaryNameNode were not also running on the same machine as a DataNode or TaskTracker. Secondly, it allows inherent horizontal scalability in the cluster itself, as more slave nodes can be trivially

Listing 6.1: Giraph error message

```
java.net.ConnectException: Call to
    hadoop6.deepthought.hpsg.dcs.warwick.ac.uk/10.131.56.209:30002
     failed on connection exception: java.net.ConnectException:
    Connection refused
```

added into the cluster without requiring any complex modifications to the configuration of the cluster.

The `hadoopmaster` node operates using a quad-core processor running at 2.66GHz, 4GB of RAM, and a 500GB SATA hard drive. The master generally has better hardware than each of the slave-nodes because it is responsible for controlling the entire cluster.

Each of the slave-nodes run the DataNode and TaskTracker processes. Each slave-node operates using a dual-core processor running at 2.4GHz, 2GB of RAM, and a 250 SATA hard drive.

The nodes communicate with each other over gigabit Ethernet, with a dedicated switch, so that external network traffic to the cluster does not interfere with network traffic from within the cluster. The hardware configuration means the HDFS has a configured capacity of 1.46TB.

We did not have direct access to the Hadoop cluster, and were required to `ssh` to a machine located within the department, before `ssh`ing from this machine into the Hadoop cluster. This added an extra level of complexity to access the Hadoop cluster, yet was resolved though use of ssh-keys to allow this *machine-in-the-middle* to send and receive files from hadoopmaster with requiring a password.

The cluster was set up to meet the requirements of Hadoop and Giraph, and as such was configured to use Hadoop 0.20.205.0, Java JDK 1.7, Maven 3, and Giraph 0.2.

Issues

One major issue which surfaced when we gained access to the Hadoop cluster in the department of computer science, relating to using Giraph with more than one worker. When a Giraph job is submitted to the cluster using more than one worker, each node which gets used by the job starts a listener which receives messages sent. However, whilst Hadoop jobs submitted to the cluster were able to make use of all nodes within the cluster, Giraph jobs were only able to use one worker, and therefore one node, otherwise an error similar to Listing 6.1.

This error was indicating that there was something wrong with the network configuration, with the most likely problem being that the ports listed in the errors were closed due to a firewall, or similar. However, after some investigation, it turned out that the ports were not blocked and that there was some other issue occuring. Browsing through the user mailing list for Giraph, it became apparent that another user of Giraph was producing

the same errors as our cluster[6]. The issue with the use of Giraph on the cluster is how the workers work out which IP address they should listen on, which happens in a different manner to how Hadoop operates, hence why Hadoop jobs would successfully execute whilst Giraph jobs would not. The fix was to remove the mapping from `127.0.0.1` to `localhost` from the `/etc/hosts` file. With this change made, we were able to successfully able to execute Giraph jobs, making full use of the cluster available.

Another issue which occurred with the cluster is that node `hadoop3` began to produce error messages as show in Listing 6.2. What was confusing about this error was that all slave nodes, `hadoop{0-7}`, are identical in setup and configuration, so for only `hadoop3` to produce serious errors such as this was strange.

The approach we took to remedy this was to remove `hadoop3` from operating within the cluster. As `hadoop3` was also a DataNode, the HDFS needed to re-replicate the data held on `hadoop3` across the other seven nodes before they could continue operating as the cluster. This resulting in the cluster now being composed of one master node, and seven slave nodes.

6.3 Algorithms

6.3.1 Giraph

Giraph programs all follow a similar construct. As a minimum, the `compute()` method needs to be implemented because this is where the algorithms are executed on each vertex within the graph. Listing 6.3 shows the structure of most of the algorithms implemented using Giraph.

As explained in Section 3.5.3, a Giraph program is executed as a Hadoop job. This means that in addition to the `compute()` method being complete, other classes are required to read in and write out the graphs before and after processing to the HDFS.

The following briefly describes each of these components:

- `compute()`

  The method which implements the algorithm.

- `VertexInputFormat`

  Generates splits of the input file to distribute the graph across all workers.

- `VertexReader`

  Controls reading in the vertices from the input splits.

- `VertexOutputFormat`

  Controls the output of the graph after computation

[6] `http://mail-archives.apache.org/mod_mbox/incubator-giraph-user/201112.mbox/%3C4ED9DDAC.4070208@apache.org%3E`

Listing 6.2: hadoop3 error message

```
java.lang.Throwable: Child Error
        at org.apache.hadoop.mapred.TaskRunner.run
        (TaskRunner.java:271)
Caused by: java.io.IOException: Task process exit with nonzero
   status of 134.
        at org.apache.hadoop.mapred.TaskRunner.run
        (TaskRunner.java:258)

attempt_201204171536_0005_m_000005_0: #
attempt_201204171536_0005_m_000005_0: # A fatal error has been
   detected by the Java Runtime Environment:
attempt_201204171536_0005_m_000005_0: #
attempt_201204171536_0005_m_000005_0: #  SIGSEGV (0xb) at
   pc=0x00007f4035255dbc, pid=25782, tid=139913655404288
attempt_201204171536_0005_m_000005_0: #
attempt_201204171536_0005_m_000005_0: # JRE version: 7.0_02-b13
attempt_201204171536_0005_m_000005_0: # Java VM: Java
   HotSpot(TM) 64-Bit Server VM (22.0-b10 mixed mode linux-amd64
    compressed oops)
attempt_201204171536_0005_m_000005_0: # Problematic frame:
attempt_201204171536_0005_m_000005_0: # C  [libc.so.6+0x7fdbc]
   memcpy+0x4cc
attempt_201204171536_0005_m_000005_0: #
attempt_201204171536_0005_m_000005_0: # Failed to write core
   dump. Core dumps have been disabled. To enable core dumping,
   try "ulimit -c unlimited" before starting Java again
attempt_201204171536_0005_m_000005_0: #
attempt_201204171536_0005_m_000005_0: # An error report file
   with more information is saved as:
attempt_201204171536_0005_m_000005_0: #
   /home/hduser/hadoop/tmp/mapred/local/taskTracker/hduser/
jobcache/job_201204171536_0005/
attempt_201204171536_0005_m_000005_0/work/
hs_err_pid25782.log
attempt_201204171536_0005_m_000005_0: # [ timer expired,
   abort... ]
```

Listing 6.3: Basic structure of a Giraph program

```
public class Vertex<I, V, E, M> {
  public void compute(Iterator<M> msgIterator) {}

  public static class VertexInputFormat {}

  public static class VertexReader {}

  public static class VertexOutputFormat {}

  public static class VertexWriter {}

  public int run(String[] argArray) {}

  public static void main (String[] args) {}
}
```

Listing 6.4: JSON adjacency list representation of a graph

```
[1, 0, [[2, 1], [3, 1]]]
[2, 0, [[1, 1], [3, 1]]]
[3, 0, [[1, 1], [2, 1], [4, 1]]]
[4, 0, [[3, 1], [5, 1], [6, 1]]]
[5, 0, [[4, 1], [6, 1]]]
[6, 0, [[4, 1], [5, 1]]]
```

- `VertexWriter`

  Controls the output of the vertices for each worker after computation.

- `run()`

  Controls the running of the program, including handling of command-line options.

- `main()`

  The initial method called when the program is submitted as a Hadoop job. Calls the `run()` method.

Following on from the shortest paths example [2] provided by Apache for Giraph, the decision was taken to use the same JSON[7] style format to store the graphs as adjacency list structures in the HDFS.

Listing 6.4 represents the graph shown in Figure 6.2. The adjacency list representation can be split into three parts, the vertex ID, the vertex value, and a list edges this vertex

[7] JavaScript Object Notation `http://www.json.org/`

connects to. As explained in Section 3.5.3, Giraph only processes directed graphs, hence the need to represent edges in both directions to simulate an undirected graph. The use of this JSON format also maps quite neatly onto the structure of the `Vertex` class representing a vertex within Giraph, as the vertex ID, vertex value and edges are all easily obtainable.

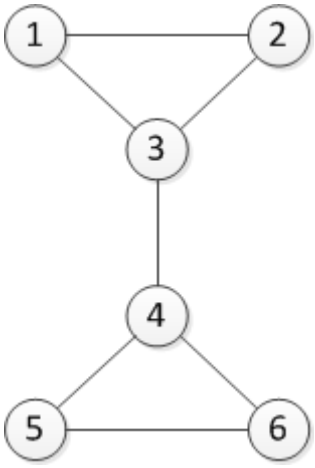


Figure 6.2: The graph represented by Listing 6.4

Listings 6.5-6.8 show example implementations of the classes VertexInputFormat, VertexReader, VertexOutputFormat and VertexWriter found in Listing 6.3. The VertexInputFormat and VertexOutputFormat classes are opposite in how they operate. The VertexInputFormat class found in Listing 6.5 splits the input graph into partitions for each Giraph worker to process on. Each of these partitions then uses the VertexReader in Listing 6.7 to parse the input graph and load it into memory for the worker to execute the algorithm on. Once all vertices have voted to halt, the reverse happens. The VertexOutputFormat class is called by each worker to create a VertexWriter which writes the output from each worker to a file.

Hadoop, and by extension Giraph, writes the output of jobs into a user specified output directory, and then into a series of files within this directory which contain the output. With Giraph, there is an output file for each worker which is used for the execution of the program, and these following a naming scheme such as `part-m-00001` up to the number of workers used.

As explained previously, the `compute()` method is where the implementation of algorithms are located. Extra methods can be produced in addition to the `compute()` method, but the `compute()` method must be implemented for the program to run.

Finally, the `run()` and `main()` methods control the execution of the job. The `main()` method is simply a passthrough method to the `run()` method. It is present because of how Giraph jobs are submitted to Hadoop for execution, and passes runtime arguments onto the `run()` method. The `run()` method sets and maintains the configuration of the job during execution through the `GiraphJob` class. The`GiraphJob` class allows configuration of settings to use each of the classes described in Listings 6.5-6.8. In addition to these, it allows the `Vertex` class to be set, as well as the input and outpout locations of where

Listing 6.5: Giraph example VertexInputFormat [2]

```
public static class SimpleShortestPathsVertexInputFormat extends
        TextVertexInputFormat<LongWritable, DoubleWritable,
            FloatWritable> {
    @Override
    public VertexReader<LongWritable, DoubleWritable,
        FloatWritable>
            createVertexReader(InputSplit split,
                               TaskAttemptContext context)
                               throws IOException {
        return new SimpleShortestPathsVertexReader(
            textInputFormat.createRecordReader(split, context));
    }
}
```

Listing 6.6: Giraph example VertexOutputFormat [2]

```
public static class SimpleShortestPathsVertexOutputFormat
   extends
        TextVertexOutputFormat<LongWritable, DoubleWritable,
        FloatWritable> {

    @Override
    public VertexWriter<LongWritable, DoubleWritable,
        FloatWritable>
            createVertexWriter(TaskAttemptContext context)
            throws IOException, InterruptedException {
        RecordWriter<Text, Text> recordWriter =
            textOutputFormat.getRecordWriter(context);
        return new
            SimpleShortestPathsVertexWriter(recordWriter);
    }
}
```

Listing 6.7: Giraph example VertexReader [2]

```
public static class SimpleShortestPathsVertexReader extends
        TextVertexReader<LongWritable, DoubleWritable,
           FloatWritable> {

    public SimpleShortestPathsVertexReader(
            RecordReader<LongWritable, Text> lineRecordReader) {
        super(lineRecordReader);
    }
    @Override
    public boolean next(MutableVertex<LongWritable,
                        DoubleWritable, FloatWritable, ?>
                            vertex)
            throws IOException, InterruptedException {
        if (!getRecordReader().nextKeyValue()) {
            return false;
        }
        Text line = getRecordReader().getCurrentValue();
        try {
            JSONArray jsonVertex = new
               JSONArray(line.toString());
            vertex.setVertexId(
                new LongWritable(jsonVertex.getLong(0)));
            vertex.setVertexValue(
                new DoubleWritable(jsonVertex.getDouble(1)));
            JSONArray jsonEdgeArray =
               jsonVertex.getJSONArray(2);
            for (int i = 0; i < jsonEdgeArray.length(); ++i) {
                JSONArray jsonEdge =
                   jsonEdgeArray.getJSONArray(i);
                Edge<LongWritable, FloatWritable> edge =
                    new Edge<LongWritable, FloatWritable>(
                        new LongWritable(jsonEdge.getLong(0)),
                        new FloatWritable((float)
                           jsonEdge.getDouble(1)));
                vertex.addEdge(edge);
            }
        } catch (JSONException e) {
            throw new IllegalArgumentException(
                "next: Couldn't get vertex from line " + line,
                    e);
        }
        return true;
}}
```

Listing 6.8: Giraph example VertexWriter [2]

```
public static class SimpleShortestPathsVertexWriter extends
        TextVertexWriter<LongWritable, DoubleWritable,
           FloatWritable> {
    public SimpleShortestPathsVertexWriter(
            RecordWriter<Text, Text> lineRecordWriter) {
        super(lineRecordWriter);
    }

    @Override
    public void writeVertex(BasicVertex<LongWritable,
       DoubleWritable,
                            FloatWritable, ?> vertex)
            throws IOException, InterruptedException {
        JSONArray jsonVertex = new JSONArray();
        try {
            jsonVertex.put(vertex.getVertexId().get());
            jsonVertex.put(vertex.getVertexValue().get());
            JSONArray jsonEdgeArray = new JSONArray();
            for (Edge<LongWritable, FloatWritable> edge :
                    vertex.getOutEdgeMap().values()) {
                JSONArray jsonEdge = new JSONArray();
                jsonEdge.put(edge.getDestVertexId().get());
                jsonEdge.put(edge.getEdgeValue().get());
                jsonEdgeArray.put(jsonEdge);
            }
            jsonVertex.put(jsonEdgeArray);
        } catch (JSONException e) {
            throw new IllegalArgumentException(
                "writeVertex: Couldn't write vertex " + vertex);
        }
        getRecordWriter().write(new Text(jsonVertex.toString()),
            null);
    }
}
```

the program should read and write to. It also allows the number of Giraph workers to be set, which specifies how many workers are required for the job to execute.

6.3.2 Issues

Difficulties have been found using Giraph at most stages of development. Giraph has had substantial development during the time frame of the project, which results in changes to the API in many areas, however few changes were made to the classes which were used from the API. We have used version 0.2 of Giraph for our development, however even since choosing this version to base all our development from, more development has occurred to Giraph. We have chosen not to use a newer version of Giraph to avoid introducing any bugs into our code caused by any changes to existing features.

Another issue which we found with Giraph is the actual process of designing and implementing algorithms for the Giraph paradigm. One problem, which is present in nearly all uses of distributed computing, is how a global data structure is decomposed across all nodes involved with the computation. With Giraph however, there is no inherent global data structure. The process of writing Giraph programs involves only considering what happens at each vertex within a graph, which makes problems such as the *all pairs shortest paths* problem much more difficult to implement, as this requires knowledge of more of the overall graph structure than just what occurs at each vertex. If we had been able to implement the algorithms for solving this problem, then we would have been able to implement a few more algorithms which build upon the results, such as betweeness centrality.

Another area of the Hadoop/Giraph infrastructure which we had unexpected difficulties with was related to what data structures could be sent across the network. For the triangle detection algorithm, and subsequent algorithms based on the results of this, we needed to send an array of `LongWritable` objects in messages between vertices. Java primitives are not used because at each superstep, Giraph writes to the disk to create a checkpoint in case of error. With the single-node setup described above, we used a `LongWritable[]` object as the data structure to be sent and received by each vertex. This worked as intended, producing the correct results, however when the algorithms were run on the dedicated Hadoop cluster, the programs would not run and crash. From reading the Hadoop Javadocs[8], it became clear that instead of a `LongWritable[]` object being used, a new class `LongArrayWritable` would need to be implemented, as shown in Listing 6.9. After implementing this new class, and refactoring code to make use of it, the algorithms successfully completed on the cluster.

[8] `http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/ArrayWritable.html`

Listing 6.9: The LongArrayWritable class

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.ArrayWritable;

public class LongArrayWritable extends ArrayWritable {

        public LongArrayWritable() {
                super(LongWritable.class);
        }

        public LongArrayWritable(LongWritable[] values) {
                super(LongWritable.class, values);
        }
}
```

6.4 Influence Propogation

6.4.1 Independent Cascade

The Independent Cascade model was implemented as two tasks; one to handle the "dice rolling" phase, and one to calculate the connected components of the resulting graph. The second task would be performed iteratively.

Dice Rolling

The input to this task was a file representing a graph in trivial graph format (TGF). This is a straightforward format that represents one node or edge per line; a representation which allowed the TextInputFormat and TextOutputFormat classes to be utilised for reading and writing the data. These classes allow Hadoop to easily handle text files, where each key-value pair represents one line of text (the key representing the position in the file, and the value representing the actual text).

Example TGF graph:

```
1 A false
2 B false
3 C false
4 D true
5 E false
1 2 0.19620324034685735
3 5 0.19051880402243856
3 4 0.12860629863762937
1 2 0.1707108546245374
4 5 0.12539404190086925
```

```
2 4 0.18033526297221486
1 5 0.18269135483394416
1 3 0.07440361004495522
2 5 0.13402102761559026
1 4 0.15267731629694137
```

The first five lines represent the nodes of the graph, in the format: [node id] [node name] [converted at start?]. This graph has 5 nodes, and only node D is set to be converted at the start. The next ten lines represent edges, in the format [node 1] [node 2] [edge weight].

The dice rolling step was then straightforward to implement as a single Hadoop task. All the work was implemented in the map step; the basic algorithm looked like this:

```
for each key, value {
  # key is a line number and can be discarded, we only care about the value

  values = parse(value)
  if values represent a node {
    if node is already influenced {
      write(nodeId, 0) # record this node is preinfluenced
    } else {
      # do nothing - we don't need to record this node
    }
  } else if values represent an edge{
    rand = rand(0,1)
    if rand < values.edgeWeight {
      write(node1id, node2id) # this edge is ``live'' and is recorded
      write(node2id, node1id)
    } else {
      # this edge is not live, so we don't need to record it.
    }
  }
}
```

The reduce step merely implements the identity function and copies over the data from the map step.

The output of this task will then look something like:

```
4 0
1 2
2 1
3 4
4 3
4 5
5 4
```

This might look like a strange way of representing a graph, but it was the most straight-

forward way to represent a graph using only key/value pairs of integers, and in a way that allows Hadoop to run a connected components algorithm on it.

Lines that end in zero (or a negative number) represent converted nodes. The line "4 0" shows that node 4 has been converted. Note that we do not need to record unconverted nodes – either they are implicitly recorded in the edges or they are completely disconnected from the spanning subgraph created after the dice are rolled, and are irrelevant.

Edges are recorded in the form:

```
[node1id] [node2id]
[node2id node1id]
```

Note that edges are recorded in both directions. The reason for this is explained in the next section.

Connected Components

[http://blog.piccolboni.info/2011/04/map-reduce-algorithm-for-connected.html]

This step was responsible for taking the spanning subgraph output by the last step and determining which nodes were connected to pre-converted nodes. This is equivalent to the standard connected components problem.

Our implementation for the connected components algorithm was based on a breadth first search. Every node that was connected to a converted node would be marked as converted. This process would then be repeated until all edges had been traversed or were unreachable.

Of course, such an algorithm could not be implemented as a single Hadoop job, and so this task had to be performed iteratively, with the output of one task being fed into the input for the next task. This process would be repeated until the termination condition was met.

This meant that each Hadoop task would represent one iteration of the algorithm; that is, each task would convert all the nodes that were direct neighbours of already converted nodes. The implementation of this was as follows:

The map step merely parsed each line of the file, each containing two integers, and split each line into a key-value pair.

The reduce step performed most of the work. It took advantage of the Hadoop framework, which passes to each reduce task a collection of all the values corresponding to a particular key. Since in our representation keys were nodeIds, that meant that each node would be processed by a different task, and each task would have all the relevant information for it's corresponding node.

This information consisted of: - whether the node was already converted - all the edges incident on the node (this was the reason that edges were recorded in both directions).

For example, node number 4 in the above graph corresponded to the following key-value pairs:

```
4 0
4 3
4 5
```

This represents the fact that node 4 had been converted, and that 3 and 5 were neighbouring nodes.

The actual algorithm worked as follows:

```
Input(key:integer, values:list(integers)
node1converted? = false
for each value in values {
  node1 = key
  node2 = value

  if node2 <= 0 {
    node1isConverted = true
    convertingNeighbour = node2
  } else {
    neighbours.add(node2)
  }
}

if node1isconverted {
  write(node1, convertingNeighbour) # copy this data from the last iteration
  for neighbour in neighbours {
    write(neighbour, -node1)
    changedNodes++
  }
} else {
    # node1 is not converted, so nothing changes, so just copy the previous values
    for neighbour in neighbours {
      write(node1, neighbours)
    }
}
```

(neighbour, -node1) is recorded for each converted neighbour. The presence of [nodeId] [negativeNumber] indicates that particular node has been converted, and inverting the negative number tells us which node converted it.

changedNodes is a counter, a global variable shared across Hadoop nodes. This counter is used to check for the termination condition; if changedNodes remains zero after an iteration, then execution halts.

A Ruby script was written to run and coordinate the different Hadoop tasks.

6.4.2 Linear Threshold

Once again, the trickiest part of the algorithm proved to be representing the state of the algorithm in a concise form that could handle both edges and nodes. The data format worked as follows:

Nodes were represented by [nodeId] [threshold]. If the threshold was equal to -1, this represented a node that had already been converted (and so we no longger cared what the threshold value was).

Edges are represented by [node1id] [node2id] [node2converted?]. Once again, edges were recorded in both directions, to ensure that each node would have all the neccesary information at the reduce step.

The algorithm then worked as follows:

The map step simply parsed the input file, setting the key to be the first integer on each line and the value to be the rest. As the value was therefore a Text object, further parsing had to be done in the reduce step. Checking whether this had a noticable effect on performance could be a future avenue for exploration.

The reduce step worked as follows:

```
Input: key:integer values:list(text)

node1isOn = false
influence = 0

for value in values {
  data = parse(value)
  if data represents node {
    if data[0] == -1 {
      node1isOn = true
    }
  } else if data represents edge {
    neighbours.add(data[0])
    if data[1] == 1 {
      influence++ \# count converted neighbours
    }
  }
}

if node1isOn \&\& influence >= threshold {
  changedNodes++
  write(key, -1) \# mark this node as converted
  for neighbour in neighbours {
    write(neighbour, ‘‘key 1’’) \# mark each neighbours edge so the neighbour’s influ
  }
} else {
```

```
  \# nothing happens, so just copy over previous values
}
```

6.5 Emergent Behaviour

The implementation of the research component began at the end of the first term. At first, a prototype implementation of the algorithm presented in Griffiths 2008 was implemented in Java. This was completed before the end of the first term. The purpose of this implementation was to check that the results in the paper could be replicated, to check that an implementation in Java was feasible and to gain a better understanding of the algorithms. This implementation was completed before the end of the first term.

This implementation was used to ensure that we had a good understanding of how the algorithms in the original paper worked, and to this effect was successful. We found that we had made a couple of incorrect assumptions based on the original paper, such as using directed edges, where the original undirected edges.

The code for this prototype implementation is available on the attached CD. (https://github.com/warwi
of-Tag-Based-Cooperation/tree/4008d994142c06501a0baa577568249d4ecda91f). The following graph shows the results we obtained using this implementation. They show that (TODO).

The prototype implementation did not allow changes to be made easily. We decided that over the second term we would start anew and re-write the Java code in order to provide a better implementation which was more amenable to adding new functionality.

In the second term, we started to write the final implementation. We decided to base the graph on JUNG for the reasons mentioned in the design section. In this implementation, agents are represented as vertices in the JUNG graph. In order to facilitate this, we had to write a custom subclass to the Apache Factory interface to generate new agents in the graph.

The basic algorithm is run over a number of iterations (generations). In each iteration, the donation phase is run, followed by the reproduction algorithm. These algorithms were designed to allow the user to provide an instance of an object which will provide some of the key features such as rewiring.

A system diagram representing the key data structures and how they interact is shown in figure 6.3. The entire system is encapsulated by a instance of the SimGraph class. This class contains an instance of a JUNG graph. Each vertex in this graph is an instance of the Agent class, which is itself an instance of the Vertex class. Each Agent has a unique identifier which is used to access the isObserving and isBeingObservedBy data structures. These data structures are used to handle the observations each agent makes of its neighbouring agents. Each agent can be observing multiple agents therefore, these lookups return a mapping of agents that the agent is observing to the instance of the Observer class that stores the observations the observer made on that agent. The Observer class encapsulates a queue data structure of boolean observations. The two ob-

serving data structures, isObserving and isBeingObservedBy are used to provide access to the Observer instances quickly. When a donation is being made, the observations can be quickly added using the isBeingObservedBy data structure, where the isObserving data structure allows the context assessment to be calculated quickly. Two methods, registerObserver and deregisterObserver are used to quickly manipulate the isObserving and isBeingObservedBy data structures, these operations are called during the rewiring operation, as the neighbourhoods and therefore the observers change as a result of the rewiring operation. The SimGraph class offers a delegate interface, GraphDelegate. The graph instances call methods on a delegate which implements this interface, and is used to notify the delegate of changes to the graph, such as edges being added or removed. It is in these methods that the isObserving and isBeingObservedBy data structures are modified.

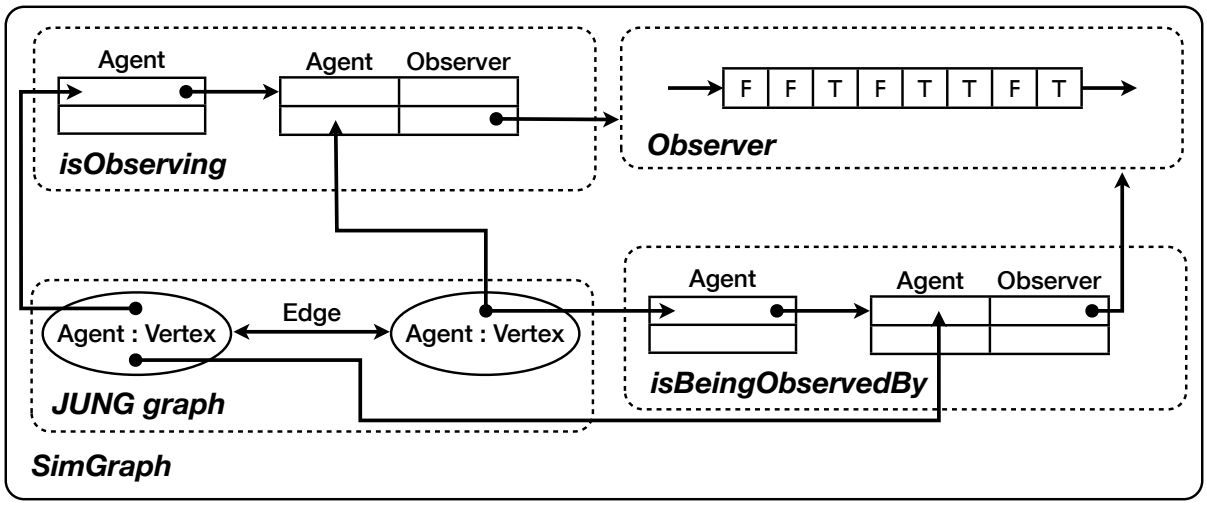


Figure 6.3: Graph Data Structures

The donation algorithm used is presented in figure 6.4. The main loop runs once for each interaction pairing per generation. In each iteration, every agent is selected (agent A). A random neighbour of agent A is selected (agent B). The tag and tolerances are compared along with the context assessment to determine if agent A will donate to agent B. If agent A has chosen to donate to agent B, then agent A will incur the cost of donating which agent B will gain the benefit of receiving a donation. Once the choice to donate has been made, every observer of agent A will be informed of agent AâĂŹs decision to donate or not to donate. After every agent has had the opportunity to donate once for each interaction pairing per generation, the donation phase is complete.

The learning algorithm is presented in figure 6.5. Every agent is selected (agent A). Agent A will select a random agent (agent B) at random. Agent A compares his score to the score of Agent B. If Agent A has a higher score than Agent B, then no learning takes place and the loop continues to the next iteration. If Agent A has a lower score than Agent B, i.e. Agent B is more successful than Agent A, then Agent A will learn from Agent B. Agent A adopts Agent BâĂŹs tag and tolerance, with a possibility of donation. After this, Agent A will then use the current rewiring algorithm to rewire its local neighbourhood. Once every agent has had the chance to learn, the learning phase

```
pairings remaining = interaction pairings per generation
while pairings remaining > 0 do
    for all agent ∈ Graph.vertices do
        neighbours = agent.neighbours()
        random neighbour = neighbours.random()
        will donate = agent.shouldDonateTo(random neighbour)
        if will donate = true then
            agent.score = agent.score - cost
            random neighbour.score = random neighbour.score + benefit
        end if
        observeDonation(agent, random neighbour, will donate)
    end for
end while
```

Figure 6.4: Donation Algorithm

is complete.

```
for all Agent ∈ Graph.vertices do
    random agent = Graph.random vertex()
    if random agent.score > agent.score then
        agent.tag = mutate(random agent.score, mutation probability)
        agent.tolerance = mutate(random agent.tolerance, mutation probability)
        agent.rewire()
    end if
end for
```

Figure 6.5: Learning Algorithm

At this point, the donation and learning phases have complete, and the single generation has occurred. The donation rate during this donation is calculated using the formula number of donations/opportunities to donate, and these values are reset for the next generation. The calculated donation rates are appended to a comma-separated file database, which can be easily loaded into a spread sheet for examining the results.

Before each simulation, the graphs are generated using the JUNG random graph generating algorithms.

Parameters are loaded into each simulation using an instance of the Parameters class.

The following provides a short description of each class in each package.

- `sim`

  Sim The application entry point, sets the parameters and runs a simulation.

- `sim.generators`

  GraphGenerator Abstract base class, used to generate graphs.

RandomGenerator Extends `GraphGenerator`, creates a random graph using an *Erdos-Renyi Generator*. Requires target vertex and edge counts.

ScaleFreeGenerator Extends `GraphGenerator`, creates a scale-free graph using a *Barabasi-Albert Generator*. Requires the initial vertex count, the number of edges to add per timestep, and the number of timesteps.

SmallWorldGenerator Extends `GraphGenerator`, creates a small-world graph using a *Kleinberg Generator*. Required the lattice size, the number of long distance connections, and the clustering exponent.

- `sim.io.graphml`

  GraphMLReader Reads graphs from GraphML files.

- `sim.model`

  Observer Records an Agent's observations of another Agent.

  Parameters Stores the parameters of a simulation.

- `sim.model.edge`

  Edge Empty class—used to represent edges in the graph

- `sim.model.graph`

  GraphDelegate An interface defining methods which can be used to respond to changes to the graph.

  GraphType Abstract base type, used to specify the nature of the graph to generate in a simulation.

  RandomGraph Extends `GraphType`.

  ScaleFreeGraph Extends `GraphType`.

  SmallWorldGraph Extends `GraphType`.

  SimGraph Wrapper around the JUNG `Graph` class. Provides notifications to a delegate object of changes to the underlying graph.

- `sim.model.rewiring`

  RewireStrategy Abstract base class for rewiring algorithms. Provides methods to rank neighbours and perform basic rewiring operations.

  RandomRewire Extends `RewireStrategy`, implements the Random rewiring algorithm.

  RandomReplaceWorstRewire Extends `RewireStrategy`, implements the Random-Replace-Worst rewiring algorithm.

  IndividualReplaceWorstRewire Extends `RewireStrategy`, implements the Individual-Replace-Worst rewiring algorithm.

GroupReplaceWorstRewire Extends `RewireStrategy`, implements the Group-Replace-Worst rewiring algorithm.

- `sim.model.vertex`

  Vertex Used as a vertex in the JUNG graph.

  Agent Extends `Vertex`, used to store data about each agent, such as the tag and tolerance.

  VertexFactory Used to generate new `Agent` objects as Vertices in the graph, allows lookup of generated `Agent` instances by id.

- `sim.simulation`

  Agents Keeps track of all the agents in a graph, along with their observations.

  DonateAlgorithm Implementation of the donation phase.

  LearnAlgorithm Implementation of the learning phase.

  Simulation Main code for the simulation, takes a set of parameters, generates the graph and initial state of the simulation, and then runs the simulation.

- `sim.util`

  RNG A singleton subclass of java's `Random` class.

Chapter 7

Testing

7.1 Blah

7.2 Emergent Behaviours

In order to validate the results, we compared the results we obtained against the results presented in the papers by Griffiths and Luck. We consistently found that the donation rate followed the same trends as the original papers, however we noticed that in our implementation the donation rate was higher in all the experiments.

In the following experiments, the donation rate was averaged over 10 runs, each with 1000 generations.

The first test we performed was to test the basic RCA algorithm against the modified one presented in Griffiths 2008. We based our experiments off the same experiments presented in the paper, each of which plot donation rate against: context influence (figure 7.1), rewire proportion (figure 7.2), population size (figure 7.3), interaction pairings per generation (figure 7.4) and neighbourhood size (figure 7.5).

The donation rate against context influence shows that the donation rate increases with context influence, and closely mirrors the values for donation rate presented in the original paper. These results can be seen in figure 7.1.

In a population of 10% cheaters, the rewire proportion exhibited a similar effect on the donation rate as was presented in the original paper, as shown in figure 7.2. It can be seen in the graph that with the random rewire strategy, the rewire proportion has no effect on the donation rate. The random replace worst, individual replace worst and group replace worst exhibit a similar behaviour, when the rewire proportion is 0, the donation rate is the same as the random rewire strategy but when the rewire proportion is increased, the donation rate jumps higher. As presented in the original paper, when the rewire proportion nears 1, the donation rate begins to drop again.

The graph of population size against donation rate, figure 7.3, shows that the population size makes no difference to the overall donation rate.

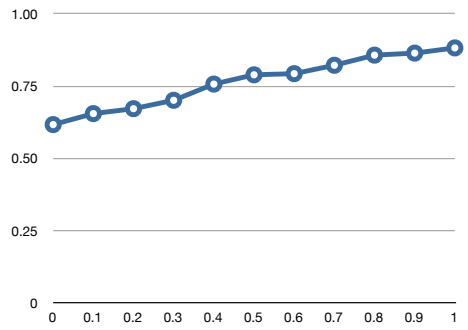


Figure 7.1: Context Influence against Donation Rate

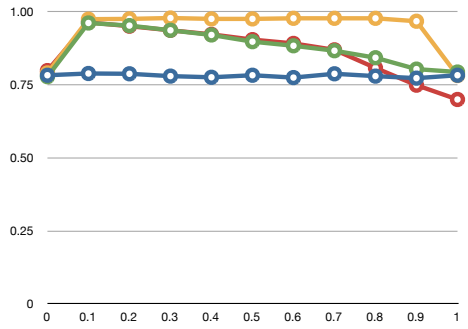


Figure 7.2: Rewire Proportion against Donation Rate

The number of interaction pairings per generation also has little effect on the donation rate as shown in figure 7.4.

From the graph that shows the neighbourhood size against donation rate, figure 7.5, it can be seen that the donation rate is fairly similar for small neighbourhoods, but there is a drop in the donation rate when the neighbourhood size reaches 50.

From these results, we can see that the implementation performs similarly to that of the original papers.

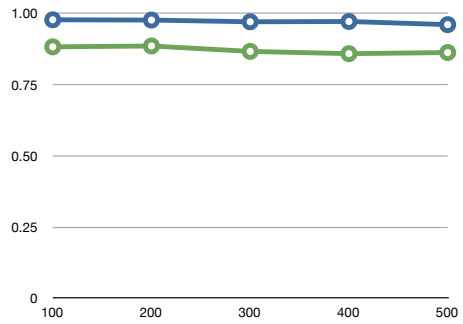


Figure 7.3: Population Size against Donation Rate

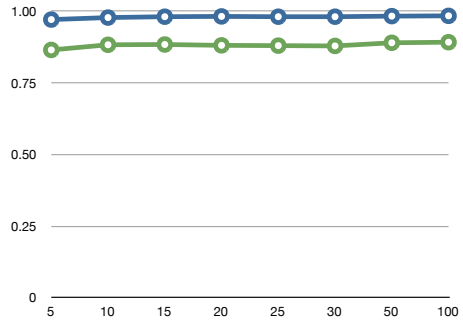


Figure 7.4: Interaction Pairings against Donation Rate

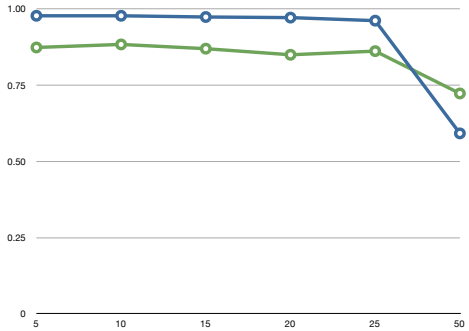


Figure 7.5: Neighbourhood Size against Donation Rate

Chapter 8

Results

8.1 Data Sets

This section details results we observe about various datasets. We have performed analysis on the Twitter datasets which were collected on the English Defence League and Unite Against Fascism.

The Stanford Network Analysis Project website, which hosts various datasets we are performing analysis on, provides statistics about each dataset available, including the number of triangles within the dataset (assuming the network is undirected), and the average clustering coefficient. The average clustering coefficient is the network local clustering coefficient.

8.1.1 English Defence League

With the data collected on the English Defence League, we applied some of the algorithms implemented to perform analysis.

PageRank

There is a bias present within the results of the performing the PageRank algorithm on the data collected relating to the EDL. As can be seen in both tables 8.1 and 8.2, the Twitter account with the highest PageRank is the edlsupportgroup. This is unsurprising given that the edlsupportgroup account was chosen as the root node for the data collecting.

The most surprising outcome from performing PageRank is that TwitPic[1] and TwitPic's founder Noah Everett are ranked second and third. TwitPic is a service to share photos and videos using Twitter, so it is likely that many of the accounts of which data was collected make use of TwitPic resulting in it's high placement.

[1] `http://twitpic.com/`

Twitter Name	PageRank
edlsupportgroup	0.08416403924447517
TwitPic	0.05975756896253487
noaheverett	0.058158047141795156
everythingEDL	0.0257624790515872
Official_EDL	0.019653741734420287
hopenothate	0.016939681748134754
TommyRobinson3	0.016757784171303376
leicspolice	0.015009582286344358
jihadwatchRS	0.011358838738844794
OneMillionUtd	0.010758557526296803
british_freedom	0.010247425883370982
USoBritain	0.008615666297945848
EDLGommy	0.008447834177284598
edlnews	0.0082414322437749
SuptPayneWMP	0.007883869657789162
KateUSoB	0.007613706376677299
Englandstruth	0.007590315713051853
PboroCops	0.007534644943547635
exposetweets	0.007522329904304726
MuslimCouncil	0.007313457877227599

Table 8.1: PageRank on EDL data

The majority of the accounts listed in tables 8.1 and 8.2 are however related to the EDL, with an approximate equal number for and against the EDL.

Clustering Coefficients

Due to the approach of collecting the data, we felt that the results of the clustering coefficients would be quite heavily biased.

8.1.2 Unite Against Fascism

The analysis performed on this data is similar to the analysis performed on the data collected about EDL.

Clustering Coefficients

As the data collected about the UAF was collected in an identical manner to the EDL, we again felt that the results of the clustering coefficients would again be quite biased.

Twitter Name	PageRank
edlsupportgroup	0.09027442551864633
TwitPic	0.05159443925208596
noaheverett	0.050126221757179935
everythingEDL	0.02798669165838786
Official_EDL	0.025266481999928907
leicspolice	0.020755126860433737
TommyRobinson3	0.019016391652185494
hopenothate	0.014987830036646874
edlnews	0.012970290691972114
jihadwatchRS	0.012497551750545299
brianoflondon	0.011623243499760285
MuslimCouncil	0.011357910646713614
british_freedom	0.011315839428723318
Englandstruth	0.011155872770274453
SuptPayneWMP	0.011037133218573704
EDLGommy	0.009545490480397533
PboroCops	0.009501841973798724
ChrisWebbo	0.009457741742444896
snidey_UK	0.008122928964687613
OneMillionUtd	0.007613901445608443

Table 8.2: PageRank on EDL considering edges as tweets

8.1.3 Enron

One dataset which we needed to check the performance of DSNAT with was the Enron email network[2]. This is a dataset containing nodes representing email addresses, and connections existing between the nodes if at least one email was sent between them. The dataset contains 36,692 nodes and 367,662 edges.

For the Enron dataset, SNAP lists the number of triangles existing within the network as 727,044 and the average clustering coefficient as 0.4970.

3-Clique

As stated previously, the identification of 3-cliques within a graph is also known as triangle identification. Using the triangle identification algorithm, we were able to mutate the network such that the only nodes and edges remaining within the graph were those nodes constructing at least one triangle two other nodes.

The number of triangles existing within the network is simply the sum of the values stored at all of the nodes within the network, as the node with the highest ID in the triangle contains the count of the triangle.

[2] `http://snap.stanford.edu/data/email-Enron.html`

Node ID	Number of Triangles
136	17744
195	15642
76	13767
370	13671
273	13401
1028	13064
416	11957
292	11415
140	11265
175	10775
734	10621
458	9217
1139	8780
520	8662
188	8450
444	8199
478	8068
241	7874
639	7783
823	7590

Table 8.3: Top 20 nodes forming triangles

After summing the number of triangles, we also arrive at the same figure of 727,044 triangles present within the Enron network. Table 8.3 shows the node ID of email addresses present in the Enron network forming the most triangles.

Clustering Coefficients

There are 12499 nodes with a local clustering coefficient of 1, which indicates that the direct neighbours of these nodes are also themselves direct neighbours of each other.

There are also 12240 nodes which have a local clustering coefficient of 0, which indicates that nearly one third of the nodes within the network do not form any triangles, and as such are likely to be email addresses external to Enron.

The network local clustering coefficient is the average of all local clustering coefficients. For the Enron network, DSNAT returns the result of performing the network local clustering coefficient algorithm as: `0.49698255959950327`.

This value is near identical to the average clustering coefficient provided by with the dataset by SNAP, so shows that our tool, and these implemented algorithms are working correctly.

Node ID	PageRank
5038	0.013662506409208037
273	0.0032385323056388035
140	0.0029987944502075873
458	0.0029642208948891463
588	0.0029383621879706633
566	0.0029094845599689167
1028	0.0027870875267837626
1139	0.0025456338556696773
370	0.0023498594485031215
893	0.002199691021020677
195	0.002105444989925047
823	0.002006310315755743
136	0.001883568213027711
286	0.0016956990694867655
95	0.001694141051220431
5069	0.0016638405778172485
5022	0.0016023313239255674
76	0.0015252624602667306
292	0.0015224551091379053
1768	0.0015039521949740957

Table 8.4: Top 20 nodes ranked by PageRank values

PageRank

Table 8.4 shows the top twenty nodes in the Enron email network, as ranked by the PageRank algorithm.

What is interesting from the results of running PageRank is that eleven of the top twenty nodes ranked by PageRank also feature within the top twenty nodes connected to the highest number of triangles. These nodes are: 76, 136, 140, 195, 273, 292, 370, 458, 823, 1028 and 1139

8.1.4 Amazon Product Co-Purchasing

The Amazon product co-purchasing network[3] is data collected from the crawling of the Amazon website. It is based on the *Customers Who Bought This Iterm Also Bought* feature of the Amazon website. A node within the network represents a product on sale, and a directed edge between nodes represents that the product has been co-purchased with the product the edge connects to. The data was collected in March 2003 [22].

The datasets contains 262,111 nodes and 1,234,877 edges, and is listed with an average clustering coefficient of 0.4240 and 717,719 triangles present.

[3] `http://snap.stanford.edu/data/amazon0302.html`

8.1.5 Wikipedia Talk

The Wikipedia Talk network[4] contains 2,394,385 nodes and 5,021,410 edges. There are 9,203,519 triangles present within the network, and the network has an average clustering coefficient of 0.1958.

8.1.6 Patent Citation

The Patent citation network[5] contains 3,774,768 nodes and 16,518,948 edges. There are 7,515,023 triangles present within the network, and the network has an average clustering coefficient of 0.0919.

8.1.7 LiveJournal

The LiveJournal social network[6] contains 4,847,571 nodes and 68,993,773 edges. There are 285730264 triangles present within the network, and the network has an average clustering coefficient of 0.3123.

Unfortunately, this dataset is too large to be analysed on the Hadoop cluster we have access to, so no extra analysis was able to be performed.

8.1.8 Twitter

The large Twitter dataset available indirectly from the SNAP website[7][8] consists of 41.7 million nodes and 1.47 billion edges. However, since the LiveJournal dataset was too large to process on the cluster we had available, it was decided not to attempt to process the social network presented.

8.1.9 Conclusions

We have shown that the system we have produced has been able to process social networks of a much larger size than possible using the previous SNAT tool developed last year. The actual results from the datasets not directly related to the intended use of our tool are meaningful to us in anyway, other than confirming that the tool is working correctly.

It is a shame that the LiveJournal dataset was too large to process, and similarly so with the large Twitter dataset. These networks represented large social networks, of which identifying clusters and users of importance would have been interesting and insightful.

[4] `http://snap.stanford.edu/data/wiki-Talk.html`
[5] `http://snap.stanford.edu/data/cit-Patents.html`
[6] `http://snap.stanford.edu/data/soc-LiveJournal1.html`
[7] `http://snap.stanford.edu/data/twitter7.html`
[8] `http://an.kaist.ac.kr/traces/WWW2010.html`

However, due to the approach undertaken in implementing a distributed social network analysis tool, we know that with a large cluster analysis of these networks is possible. A larger cluster would also be beneficial with the analysis of the smaller networks. Executing the PageRank algorithm on the Patent Citation network took on average twenty-five minutes to complete, which contrasts with the forty seconds for PageRank to complete on the Enron dataset.

A larger cluster will allow processing of larger datasets, whilst at the same time, it will also decrease the running time of performing algorithms on smaller datasets.

8.2 Cluster Performance

There are two important areas of measuring the performance of the cluster. The first being that as the size of the cluster is increased, the running time of the algorithms being executed should decrease as more computation is occurring in parallel. The second metric is that for a fixed-size cluster, the running time of the algorithms should only be affected by the size of the input.

8.2.1 Size of Cluster

Using Giraph, a runtime parameter for the job being submitted to the Hadoop cluster is to specify the number of workers to be included within the execution of the job. This allows us to analyse the running time of the jobs utilising differing amounts of computation power available from the cluster.

Enron

Using the Enron dataset as fixed size input, we are hoping to show that as the number of workers increase, the runtime of the algorithms decrease. From observations, the PageRank algorithm has shown to be one of the more time consuming algorithms, and as such has been chosen to perform analysis of the cluster.

Figure 8.1 shows the running time of the PageRank algorithm on the Enron dataset consisting of 36692 nodes and 367662 edges. As expected, the running time of the algorithm initially decreases quite rapidly as the number of workers increases up to a fourth worker. There is minimal decrease in running time from the fourth to the seventh worker, followed by a jump after an eight worker is added. After this, the number of workers added do not affect the running time of the algorithm.

The longest running time, as to be expected, was with one worker. The shortest running time was with seven workers, again this is possibly to be expected as the cluster is in operation with seven slave nodes, so each node has an equal but small share of the original network. The running time does not decrease much from four to seven workers,

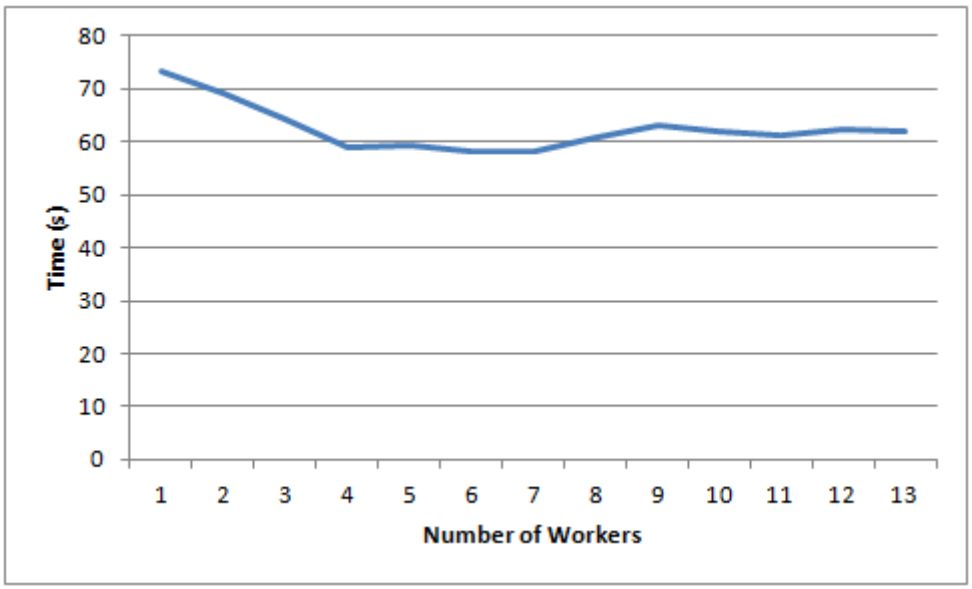


Figure 8.1: Runtime of PageRank algorithm on Enron dataset with varying number of workers

most likely due to the increased network costs of having these extra workers offsetting the decrease in running time each worker will have from having less nodes to process.

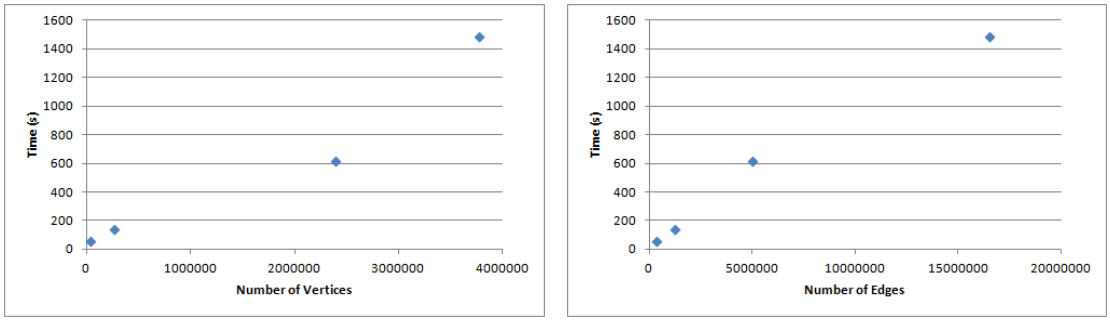
After an eight worker is included, the running time then increases and stabilises just above the sixty second mark. This is most likely due to the nodes requiring to launch extra processes for the extra workers, which slows down the execution of workers on nodes with more than one worker.

8.2.2 Size of Input

The other metric which we want to analyse performance of the cluster is to observe how the running time for algorithms change as the size of the input is increased. To achieve this, we have opted to use seven workers for the Giraph jobs, as this has shown to produce the lowest running time for the PageRank algorithm on the Enron dataset. The algorithm which will be used for this analysis is also the PageRank algorithm, due it being a fairly time-consuming algorithm in relation to the other algorithms implemented.

We will make use of each of the datasets from Section 8.1 which we have been able to perform analysis on.

Figure 8.2 shows the results of running the PageRank on different size inputs. Figure 8.2a and Figure 8.2b show that as the size of the input graphs are increased, then the cost to compute the PageRank values increases proportionally with the size of the input. For this benchmark, the PageRank algorithm was set to finish after 30 supersteps so that the dynamics of the networks were not influencing the runtime of PageRank too greatly.



(a) (b)

Figure 8.2: Running times of the PageRank algorithm on various size datasets using seven Giraph workers.

8.2.3 Conclusions

We have shown that for current state of the cluster, with seven slave nodes executing algorithms that the optimum running time for algorithms is achieved when all seven of the slave nodes are used with one worker task assigned to each. As the cluster grows, then it is quite likely that the optimum number of workers will equal the size of the cluster. We have also shown that the system is able to cope with a wide range of input sizes of graphs, which do noticeably increase the running time of the algorithms, but with an increase in the size of the cluster

8.3 Emergent Behaviours

We ran the rewiring algorithm as introduced by Griffiths and Luck on scale-free and small-world graphs. In their paper, Griffiths and Luck had only ran their algorithm on graphs with a random topology, like graphs generated by the Barabási-Albert generator. We decided that the scale-free and small-world graphs better represent the social networks, where you have dense clusters where every member knows each other and each member of a cluster might have connections to other members who are not in that cluster.

When we ran the same algorithms on the scale-free and small-world graphs, we found that the donation rate remained at a similar level, as shown in figure. However, we noticed that the topology of the graph underwent changes due to the rewiring and thus the topology morphed and became more like the random graphs, this is shown in figure.

We decided that we should look at preserving the initial topological nature of the graphs throughout the simulation, so we decided to look at alternative methods of rewiring that could manage to maintain a similar topology but preserve a donation rate as high as the current rewiring algorithms.

To this end, we decided to create an algorithm that attempts to maintain the same number of small-world connections and long-distance connections during the rewiring. This was achieved by measuring the shortest path length between two neighbours after

the neighbour has been removed and then attempting to find a neighbour that has a similar shortest path length to add.

We created two algorithms, the first would be based on the random rewiring strategy and was designed in order to test how the topology changes over time. This differs from the random rewiring strategy in two ways. When a neighbour is about to be removed, the shortest path length between the two agents is measured and stored. Next, when the agent begins to add neighbours to replace those it has removed, it will pick an agent at random and calculate the length of the shortest path between the agent and the random agent, adding a connection only if the length of the shortest path between these two agents is similar to the length of the shortest path between the agent and its neighbour which was removed.

The second algorithm is based on the group replace worst algorithm. Like the algorithm above, when a neighbour is removed, it measures the shortest path between the two agents. Unlike the above algorithm, it then uses an approach based on the group replace worst algorithm to add new nodes. It will query its best neighbours for their best neighbours and will only add a connection to these new agents if the length of the shortest path between them is similar to the length of the shortest path between the agent and its neighbour which was removed.

The results in figure show the impact that these new rewiring algorithms have on the donation rate. It can be seen in these graphs that the new donation rate for the random algorithm performs similarly to that of the existing random rewire algorithms. It can also been seen that the algorithm based on the group replace worst also exhibits similar behaviour to that of the existing group replace worst algorithm.

The results in figure show how the topology changes over time. These results show that the end topology is much closer in nature to the original topology than with the existing rewiring strategies.

Chapter 9

Evaluation

9.1 Project Management

For a project of this size, involving multiple people, a structured approach to organising how development of the project occurred was required

9.1.1 Communication

An important aspect of any project is communication between all parties involved. For this project, this included ourselves, but also our project supervisors.

Once a week we met with our project supervisors to discuss progress made from the previous meeting, and how progress would develop during the next week. These meetings also served as platform to display results of development of some aspects of the project, and directions for research in these areas could be suggested. Minutes from meetings can be found in Appendix A.

A single group email account was created: warwicksna@gmail.com. This allowed members of the group, and our supervisors, to quickly contact each member of the group as any emails sent to this account were forwarded to all group members.

During the Easter vacation, the group continued the weekly meetings, without direct communication with our supervisors, using Google+ Hangouts. Google+ Hangouts provided an online system where each member of the group could join a chatroom style environment to discuss project development use voice and/or video communications. These meetings within the holiday were followed up with a summary email to the group email account and our project supervisors to ensure that anyone unavailable to make the meeting could be informed of what had been discussed.

9.1.2 Group Management

Each member within the group had a defined role for the development of the project as explained below:

James Michael

James Kettle This member, in addition with Isaac Lewis, was involved with the design and implementation of the SOAP server and client.

This member also implemented scripts which used the Twitter API to collect data on users related to the EDL and UAF, and process the results into the `graphml` format.

Matthew Cranham This member of the group was involved with the design and implementation of the majority of the algorithms running on the cluster.

This member also became responsible for maintaining the Hadoop cluster to ensure that it performed correctly. One serious issue which occurred with the cluster was that algorithms implemented using the Apache Giraph library would not run, which was fixed through editing a networking configuration file on each machine within the cluster.

Following the implementation of the algorithms and the acquisition of data, it fell to this member to perform analysis on multiple datasets to investigate not only the structure and properties of these datasets, but also the tool developed as an environment to perform analysis of large social networks.

Isaac Lewis This member was cheifly involved with implementing influence propagation algorithms in Hadoop.

This member, in addition to James Kettle, was involved in designing and implementing the SOAP server and SOAP client.

Simon Burns This member of the group worked on further developing the user interface developed by last year's group. This involved attempting to making the visualisation component of the user interface a stand-alone component so that it could be reused for this project to visualise networks being analysed on the cluster.

9.2 Lessons

9.2.1 Hadoop

9.2.2 Emergent Behaviour

9.2.3 SOAP

9.2.4 Influence Propogation

- Can use Hadoop to implement graph algorithms without Giraph/Pregel, but its tricky.
- Most effective to work out algorithms with pen and paper before starting to code. Much trickier than single-thread coding.

9.2.5 Visualiser

9.2.6 Algorithms

9.3 Future Extensions

9.3.1 Hadoop

9.3.2 Emergent Behaviour

9.3.3 SOAP

- Extending the showStatus method so that it does more than merely echo the logging pages. A more useful method might be able to parse these logging pages, to return a particular piece of information.

- Allowing algorithms to be uploaded as Python scripts (that is, Python scripts that can call Hadoop jobs). One reason this was not done was because an implementation would likely be fairly brittle and depend on scripts being written for the particular installation of Hadoop. If a robust solution could be implemented, it would be useful for anyone who wished to develop a multi-stage Hadoop task.

9.3.4 Influence Propogation

- One influence propagation model that was not explored was the diminishing cascade model. As this is a realistic model for many situations – ie, people are often unlikely to

be converted to an idea after a previous failed conversion attempt. Implementing this model in Hadoop could be interesting.

- Running more experiments on the implemented alorithms. Graphs with different properties would be an interesting avenue to explore, ie, varying the clustering coefficient, average node degree, or edge weighting. Some papers found there to be a "tipping point" in properties such as average edge weights where influence cascades suddenly turned into epidemics; it would be interesting to explore if the same held true for other properties. In [digg paper], it was found that the presence or absence of influence "epidemics" depended strongly on the structure of the networks.

- Performance testing the implemented algorithms.

9.3.5 Visualiser

9.3.6 Algorithms

One area of the project which is a bit lacking, is the number of algorithms implemented to execute on the cluster.

Bibliography

[1] Richard D. Alba. A graph theoretic definition of a sociometric clique. *The Journal of Mathematical Sociology*, 3(1):113–126, 1973. doi: 10.1080/0022250X.1973.9989826. URL `http://www.tandfonline.com/doi/abs/10.1080/0022250X.1973.9989826`. 11

[2] Apache. Shortest paths example, March 2012. URL `https://cwiki.apache.org/confluence/display/GIRAPH/Shortest+Paths+Example`. 60, 62, 63, 64

[3] Apache. Hdfs Architecture Guide, March 2012. URL `http://hadoop.apache.org/common/docs/current/hdfs_design.html`. 31

[4] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem, 1999. 11

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998. ISSN 0169-7552. doi: 10.1016/S0169-7552(98)00110-X. URL `http://dx.doi.org/10.1016/S0169-7552(98)00110-X`. 52

[6] C. Bron, J.A.G.M. Kerbosch, and H.J. Schell. Finding cliques in a undirected graph, 1972. 11

[7] Inci Cetindil and Eugenia Gabrielova. Graph computation algorithms for the giraph framework, 2011. 51

[8] Avery Ching. Giraph: Large-scale graph processing infrastructure on Hadoop. Presented at Hadoop Summit 2011, 2011. 32, 34, 35, 37

[9] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science Engineering*, 11(4):29 –41, july-aug. 2009. ISSN 1521-9615. doi: 10.1109/MCSE.2009.120.

[10] Jonathan Cohen. Trusses: Cohesive subgraphs for social network analysis, 2008. URL `http://www.cslu.ogi.edu/~zak/cs506-pslc/trusses.pdf`. 11, 12

[11] Francesc Comellas and Alicia Miralles. A fast and efficient algorithm to identify clusters in networks. *Applied Mathematics and Computation*, 217(5):2007 – 2014, 2010. ISSN 0096-3003. doi: 10.1016/j.amc.2010.06.060. URL `http://www.sciencedirect.com/science/article/pii/S0096300310007459`. 7

[12] L. D. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas. Characterization of complex networks: A survey of measurements. *Advances in Physics*, 56:167–242, January 2007. doi: 10.1080/00018730601170527.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492. 29, 30

[14] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(98):298–305, 1973. 9

[15] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3Ű5):75 – 174, 2010. ISSN 0370-1573. doi: 10.1016/j.physrep.2009.11.002. URL http://www.sciencedirect.com/science/article/pii/S0370157309002841.

[16] Linton C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, March 1977. URL http://links.jstor.org/sici?sici=0038-0431%28197703%2940%3A1%3C35%3AASOMOC%3E2.0.CO%3B2-H.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945450. URL http://doi.acm.org/10.1145/1165389.945450. 30

[18] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, 2002. doi: 10.1073/pnas.122653799. URL http://www.pnas.org/content/99/12/7821.abstract. 7

[19] Nathan Griffiths. Tags and image scoring for robust cooperation. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 2*, AAMAS '08, pages 575–582, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9817381-1-6. URL http://dl.acm.org/citation.cfm?id=1402298.1402305. 24, 26

[20] Nathan Griffiths and Michael Luck. Changing neighbours: improving tag-based cooperation. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 249–256, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9826571-1-9. URL http://dl.acm.org/citation.cfm?id=1838206.1838241. 27

[21] D. Hales and B. Edmonds. Applying a socially inspired technique (tags) to improve cooperation in P2P networks. *Systems, Man and Cybernetics, Part A, IEEE Transactions on*, 35(3):385–395, 2005. doi: 10.1109/TSMCA.2005.846399. URL http://dx.doi.org/10.1109/TSMCA.2005.846399. 25

[22] Jure Leskovec. Stanford Network Analysis Project, March 2012. URL http://snap.stanford.edu. 83

[23] Leo Katz. A new status index derived from sociometric analysis. *PSYCHOMETRIKA*, 18(1):39–43, 1953.

[24] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970. 8

[25] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: http://doi.acm.org/10.1145/1772690.1772751. 45

[26] R. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15:169–190, 1950. ISSN 0033-3123. URL `http://dx.doi.org/10.1007/BF02289199`. 10.1007/BF02289199. 11

[27] R. Luce and Albert Perry. A method of matrix analysis of group structure. *Psychometrika*, 14:95–116, 1949. ISSN 0033-3123. URL `http://dx.doi.org/10.1007/BF02289146`. 10.1007/BF02289146. 9

[28] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL `http://doi.acm.org/10.1145/1807167.1807184`. 32, 34

[29] Claudio Martelli. Apache giraph: Distributed Graph Processing in the Cloud. Presented at FOSDEM 2012, 2012.

[30] Robert J. Mokken. Cliques, clubs and clans. *Quality & Quantity*, 13:161–173, 1979. ISSN 0033-5177. URL `http://dx.doi.org/10.1007/BF00139635`. 10.1007/BF00139635. 11

[31] M. E. J. Newman. *Networks: An Introduction*. Oxford University Press, Great Clarendon Street, Oxford, OX2 6DP, 2010. 8, 9, 12, 13, 15, 16

[32] Michael G. Noll. Running hadoop on ubuntu linux (multi-node cluster), March 2012. URL `http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/`.

[33] Michael G. Noll. Running hadoop on ubuntu linux (single-node cluster), January 2012. URL `http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/`. 56

[34] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL `http://ilpubs.stanford.edu:8090/422/`. Previous number = SIDL-WP-1999-0120. 14, 33, 34

[35] Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, May 1990. ISSN 0895-4798. doi: 10.1137/0611030. URL `http://dx.doi.org/10.1137/0611030`. 9

[36] James Ribi, Stephen Roberts, Matthew Seeley, and Phillip Taylor. Social network analysis. Master's thesis, University of Warwick, 2011. 42

[37] Rick L. Riolo, Michael D. Cohen, and Robert Axelrod. Evolution of cooperation

without reciprocity. *Nature*, 414(6862):441–443, November 2001. ISSN 0028-0836. doi: 10.1038/35106555. URL http://dx.doi.org/10.1038/35106555. 24

[38] SQLite. Limits in sqlite. URL http://www.sqlite.org/limits.html.

[39] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963491. URL http://doi.acm.org/10.1145/1963405.1963491.

[40] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL http://doi.acm.org/10.1145/79173.79181. 32, 36

[41] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of /'small-world/' networks. *Nature*, 393(6684):440–442, June 1998. ISSN 0028-0836. doi: 10.1038/30918. URL http://dx.doi.org/10.1038/30918. 13

[42] W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research*, 33:452–473, 1977. 6, 7

Appendix A

Minutes of Meetings

Meeting Minutes

Term 1

8/11/11

PRESENT

- Matt, Isaac, James K, James M

MINUTES

- We discussed the areas we had chosen to research from the previous meeting
- Met with Phil Taylor to discuss his project last year, demonstration of it tomorrow (9/11/11) at 12pm
- Hopefully write a new project spec after meeting
- Phil mentioned research areas of influence, importance and ranking, and analysis of the text of messages

ACTIONS

- Read each others researched information
- Meet with Phil Taylor tomorrow

9/11/11

PRESENT

- Matt, Isaac, James K, James M

MINUTES

- Met with Phil for demonstration of the project he helped produce last year
- Code online at snat.googlecode.com

ACTIONS

- Organise meeting with Sarab

10/11/11

PRESENT
Matt, Isaac, James K, James M

MINUTES
- possible ideas
    - turn existing code into cluster (with hadoop?)
    - analytics as a web service
    - partitioning a social network into groups
    - converge multiple networks (creating a mapping of one set of nodes to another)
    - text mining messages
- could port the software to something else
- independent cascade model
    - complicated algorithm, takes a long time
    - currently uses heuristics to speed up
    - could implement a definitive cluster version, to compare results against

ACTIONS
- come up with project idea to present to Sarab in Tuesday's meeting 3:30

18/11/11

PRESENT
Matt, Isaac, James M

MINUTES
- Read document RE: police project
- Will divide the project into two parts:
    - Developing software to run on a cluster (lead by Matt and Isaac)
    - Implementing algorithms to run on the software (lead by James & James)

ACTIONS
- Read document and references
- Talk with Phil Taylor regarding integrating cluster/separating interface from snat
- Need to have design document down soon

22/11/11

PRESENT
Isaac, Matt, James M, James K, Simon, Sarab

MINUTES
- James & James will look into Modelling Emergent Behaviours
- Simon is not getting messages to the group email, need to fix that
- Sarab discussed the clustering side of the project with Isaac and Matt:
    - Sarab recommended looking up CouchDB, and other document-store databases
    - Adam Chester, a PhD student may become involved with the project
    - Discusses the issues with running graph problems on Hadoop. Partitioning the graph will be the main issue. Sarab recommended looking up the betweeness algorithm

- Sarab said the West Midlands Police could not give us data at short notice, but they recommended using Twitter data and looking at two groups, the EDL and Unite against Fascism
- Simon will look into SOAP interface
- Simon will look at getting twitter data
- The poster (due for week 9) should mainly cover a literature review of the project area

ACTIONS
- James & James Meeting with Nathan to discuss his papers
- Isaac will look up running the independent cascade model on a cluster
- Matt will look up running the graph partitioning algos on a cluster
- Isaac will fix Simon's group email
- Deadline for poster to be ready: Next Thursday

23/11/11

PRESENT
James M, James K, Nathan, Sarab

MINUTES
- Twitter data: how to define success
    - number of followers/following ratio centrality, RT (propagating data), tweeting at (looking at content?)
    - Use #ff follow fridays for additional info
    - failure: unfollows
- Learning
    - identify topics in tweets to determine what topics imply success
    - learn what profile to show (what topics to tweet about, who to follow...)
    - what you tweet can be different to what you read
        - e.g. security might follow edl, but not tweet edl content
        - following does not mean interested in a supportive way
    - Individual nodes would have their own learning rates
        - these are initialised randomly
        - there is a chance of mutation (with a given distribution)
        - could change in response to current success level
            - don't learn if the node is doing well
            - learn quickly if the node is doing worse
- Assume noisy observations
    - can be made worse by mutations in 'reproduction/learning'
- How to define a cheater on twitter
    - discreditors
- Reward/Punishment
    - retweeting = reward
    - unfollow = punishment
- Look into negative effects: break down the organisation, reduce trust
- Sometimes useful to see more than immediate neighbours
    - go out a number of levels, e.g. 1 or 2 additional levels
    - learn a lot from local neighbourhood, a bit from outer neighbourhood
- Gathering twitter data
    - get unfollow data

ACTIONS
- reimplement Nathan's work
    - allow learning rates to differ
    - allow non-synchronous generations (learning)
    - James K: Design the Tags and Image Scoring model
    - James M: Design the Changing Neighbours model
    - meet 11 on Monday to merge
    - have a design to present on Tuesday
- read paper on gossip matters
    - destablization of an organisation by injecting suspicion
    - http://www.bus.emory.edu/prietula/prietula-carley-chapter-rev1.pdf

29/11/11

PRESENT
Isaac, Matt, James M, James K, Simon, Sarab
MINUTES

- clustering
- modelling emergent behaviours
- twitter data acquisition
- web service
- poster
    - will be ready for printing by Thursday, week 9
    - presentation on Thursday, week 10
- meeting with police, probably Friday week 10, maybe Thursday week 10

ACTIONS

- finish implementation of Nathan's algorithm
- do poster
- research more on Giraph
- have a finalised design for the SOAP message interface before the end of term

5/12/11

PRESENT
Isaac, Matt, James M, James K
MINUTES

- Poster is being printed tomorrow by Warwick Print and delivered back to dcs
- Decided which section each person will talk about
- We should be able to talk/present the poster for at least 30 minutes without any questions being asked by other people

ACTIONS

- practice section each person will talk about in the presentation

6/12/11

PRESENT
Isaac, Matt, James M, Simon
MINUTES

- Simon has got in contact with Twitter, they may be able to help with Data Acquisition
- Isaac has read some papers on Influence propagation, he will look into how to distribute it efficently and starting implementation
- Matt has looked at Giraph, will need to check if the graph is partitioned in a decent manner
- James showed the Java implementation, will look at import/exporting graphs, different network topologies, directed vs. undirected graphs, real world restrictions on rewiring and how it affects topologies, look at getting results, mapping twitter to a graph
    - relevant
        - JUNG - Java Universal Network/Graph framework
        - networkx
        - GraphML
        - SNAP - Stanford Network Analysis Project
- Discussed poster presentation, format will likely be 10 minutes of us presenting the poster, any additional time can be used for questions

ACTIONS

-

Term 2

13/1/12

PRESENT

- Matt, Isaac, James K, James M, Simon, Sarab

MINUTES

- Progress made over Christmas break
    - Matt got Giraph to work with PageRank algorithm, and now understands how Giraph jobs work
    - Hadoop is a bit inefficient when it comes to graph algorithms as it needs to model iteration in an algorithm using multiple map/reduce jobs, which write to file system after each job
- Post-Christmas Progress and fault tolerances of what needs to be done
- James and James to focus more on project once other coursework is done
- Simon looked at last year's interface

ACTIONS

- Research into how to adapt algorithms for Giraph paradigm
- James and James to start working from Tuesday
- Simon to start getting Twitter data

3/2/12

PRESENT

- Matt, Isaac, James K, James M, Simon, Nathan, Sarab

MINUTES

- Simon has made progress with previous work on GUI
- James and James presented results from implementing one of Nathan's papers
- Isaac been looking at implementing influence propagation on Hadoop
- Matt has been struggling to use Giraph
    - Errors when running example code
    - No obvious solution

ACTIONS

- Simon to work on Twitter data
- James and James to start adapting previous GUI for use
- Isaac to implement an influence propagation algorithm
- Matt to try and get a community detection algorithm working in either Giraph or Hadoop

10/2/12

PRESENT

- Matt, Isaac, James K, James M, Simon, Nathan, Sarab

MINUTES

- James and James have converted the graphs they've made using Jung and Peersim into SQL and inserted into the existing databases
    - Visualiser doesn't seem to display these properly
    - Talk to Phil about the database
- Simon worked on the visualiser
- Isaac is implementing an Influence propagation algorithm for hadoop
    - Not sure how to parse the graph data into Hadoop
- Matt has implemented a community detection algorithm for identify triangles within a graph
    - Not outputting correctly, but should be simple to fix
    - Needs to run on some other data sets
    - Analysis of algorithms on large dataset would be good with varying amounts of nodes to establish if distributing computation actually improves runtime of algorithms

ACTIONS

- James M to talk to Phil about the database
- James K to start gathering twitter data
- Matt to finish the community detection algorithm
- Matt to parse existing graphs into format suitable for Hadoop/Giraph
- Matt to run community detection algorithm on more data sets
- Matt to talk to Olly about Hadoop cluster in dcs
- Isaac to look at Stanford Network Analysis Project data
- Isaac to finish implementing influence propagation algorithm
- Simon to finish work on the visualiser

17/2/12

PRESENT

- Matt, James K, James M, Nathan, Sarab

MINUTES

- James M presented results of rewiring on JUNG generated graphs
- Matt has got Hadoop working on a set of livejournal data
    - but it runs out of memory
- Matt produced a parser for the SNAP twitter and livejournal dataset for use on Giraph
- James K presented twitter data

ACTIONS

- James M to email results to Nathan and Sarab
- James M to repeat experiment with different numbers of iterations
- James M to gather metrics about how the graph changes over time
- Matt to talk to Oliver about getting a Hadoop cluster set up
- James K to work on getting more twitter data, have a look at visualising the data