UNIVERSITY OF
BIRMINGHAM

MSc Project

Web Application for Community Detection in Large Scale Networks

---

Andrew Walker

1336061

Supervisor: Dr Shan He

Date: August 2014

Abstract

Networks representing real systems commonly contain a community structure, in which groups of nodes are densely connected internally with sparser connections between different groups. The identification of community structure is important in many scientific fields because it provides insight into the relationship and interaction between network function and topology. As networks become increasingly large, the speed and scalability of algorithms used to find their community structures become very important. The majority of current applications providing implementations of these algorithms require programming knowledge to use and offer limited options to visualise the results. We provide a client-server web application which performs community detection on the server and allows visualisation of the resulting community structure on the client. This visualisation is interactive and is capable of displaying multi-level hierarchical community structures. Within this application, we provide implementations for three community detection algorithms designed for large graphs, namely: label propagation, the Louvain Method, and ORCA Reduction and ContrAction. We investigate the accuracy of these algorithms and find that all three algorithms are capable of finding established communities in a number of real world networks with good accuracy. By investigating the speed performance of these algorithms and the application as a whole, we discover that the performance and scalability is limited primarily by our efforts to visualise community structures efficiently rather than detect them.

Acknowledgements

Thank you to my project supervisor, Dr Shan He, for his continued guidance and support throughout the duration of this project.

Contents

1 Introduction

Networks representing real systems often display a number of topological features. Of particular interest is the presence of a community structure. Nodes may be organised in communities, with many edges joining nodes of the same communities and comparatively few edges joining vertices of different communities (Fortunato, 2010) such as in figure 1. For example, in social networks, communities naturally form around shared interests or through friendship groups, families and work environments. Protein-protein interaction networks in biology are often investigated as the proteins that interact with each other most frequently form communities where each community corresponds to a particular biological function. Being able to find these community structures can provide insight into how network function and topology affect each other.

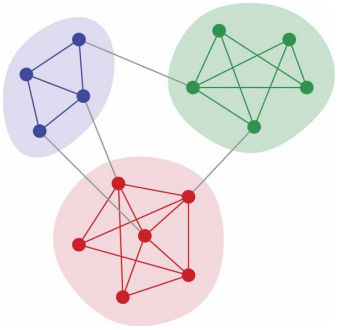


Figure 1: Example graph with three distinct communities shown by the three different colours.

The size of these real world networks are becoming increasingly large. For example, the social networks provided in the Stanford Large Network Dataset Collection (Stanford University, 2009) contain up to 65 million nodes and almost two billion edges. Finding community structures is computationally an expensive task, and many current algorithms are incapable of dealing with graphs of such great size. Their time complexities may be quadratic or worse in the number of edges or vertices in the input graph. This increased complexity often comes with advantages, such as improved accuracy or identification of overlapping communities (Danon et al., 2005). However, for large graphs on current hardware they are infeasible so algorithms must be used that have been shown to perform with better than quadratic complexity in the number of edges and nodes in the input graph.

Implementations of several algorithms that are capable of dealing with large graphs are currently available as part of numerous software libraries, but these require programming knowledge to use and often have limited options in terms of visualising the results of the algorithms. Furthermore, these implementations process graphs in main memory which limits the size of the input graph.

This work focuses on three main contributions:

1. A client-server web application for detecting community structures in large graphs that allows for visualisation of multi-level community structures (community hierarchy as described in section 2.2).

2. Implementations of three community detection algorithms in GraphChi, a graph processing library that processes graphs efficiently from disk rather than just main memory. This includes a modification and implementation of the ORCA Reduction and ContrAction detection algorithm which previously had no publicly available implementation

3. An investigation into the speed performance and community structure quality of three community detection algorithms that are intended for large scale networks.

Section 2 describes the methods used for evaluating community detection algorithms, explains the concept of a community hierarchy and looks at existing software that provide community detection. Section 3 describes the community detection algorithms chosen for this web application, looking at how they work and their potential performance on very large graphs. Section 4 outlines a specification for the web application along with user requirements, introduces the GraphChi graph processing library, and describes the design used to meet the specification. Section 5 describes some of the implementation details of this design, in particular the implementation details for the three detection algorithms, and the automated testing procedures for the application. Section 6 describes the user interface and provides an illustration of how this interface works for some example real world networks. Section 7 describes the results of the investigation into the performance and quality of the detection algorithms. Section 8 explains the software process model used for developing the application. In section 9 we evaluate the web application and development process, and finally we summarise the work we have done in section 10 and look at potential future work.

2 Background

In this section we describe how community detection algorithms are evaluated and look at currently available applications that can visualise community structures. The specific algorithms chosen for this project are outlined in section 3.

2.1 Evaluating Detected Community Structures

The quality of community detection algorithms can be evaluated in a number of ways. Networks can contain numerous different distinct acceptable community structures or even overlapping community structures where a single node belongs to more than one community. Lancichinetti and Fortunato (2009) argue that there is currently no agreed upon best test or measure of the quality of a community detection algorithm. There are, however, a number of methods that are commonly used which are described here.

Real world networks The algorithms are frequently tested on real world networks such as friendships within a small karate club (Zachary, 1977) as shown in figure 2 and relationships between a group of dolphins (Lusseau et al., 2003). These networks have established groups, and algorithms are tested to see if they are capable of detecting these established groups with no prior information other than the topology of the network.

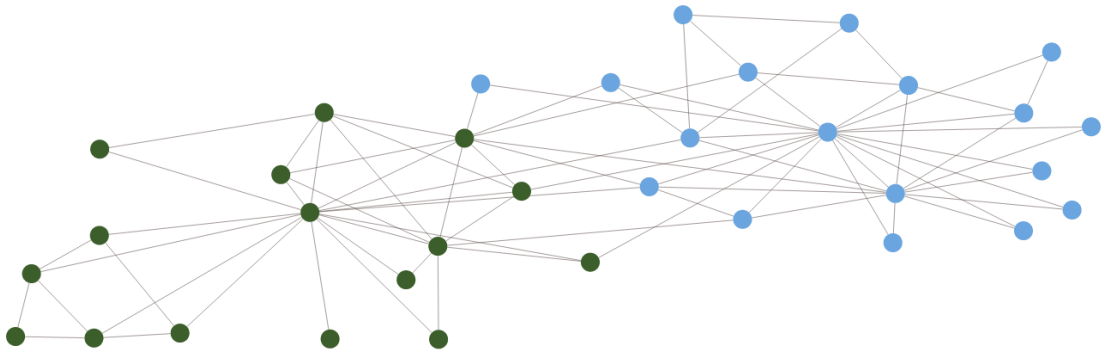


Figure 2: Social network of associations between members of a university karate club. Edges between members are present where these members were considered friends outside of the karate club. The club split in to two separate groups after disputes between key members. The two colours represent these two factions.

Generated benchmark graphs Artificial graphs generated with a clear community structure such as those in figure 3 are used to test that an algorithm can identify these structures. Lancichinetti et al. (2008) argued that these graphs have a structure that does not reflect the real properties of nodes and communities found in real networks. To counter this, they introduced a class of benchmark graphs that account for the heterogeneity in the distributions of node degrees and of community sizes. These are now commonly known as LFR benchmark graphs. An example is seen in figure 4.

LFR benchmark graphs are based on on the planted l-partition model. This involves partitioning a graph with N nodes into N/l partitions. Each node then has a probability p_{in} of being connected to nodes of its own partition and probability p_{out} of being connected to nodes outside

of its own partition. If $p_{in} > p_{out}$ then the graph has a community structure. The LFR benchmark graphs extend this model by making groups have different sizes and nodes have varied degrees.Lancichinetti et al. have made software available to generate these graphs which will be used in section 7.1 to test our implementations of detection algorithms.

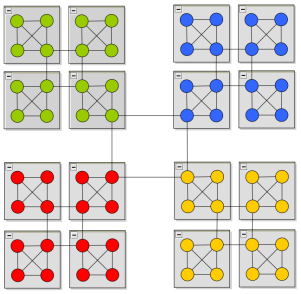


Figure 3: Generated graph with clear community structure. Image taken from Delling et al. (2009)

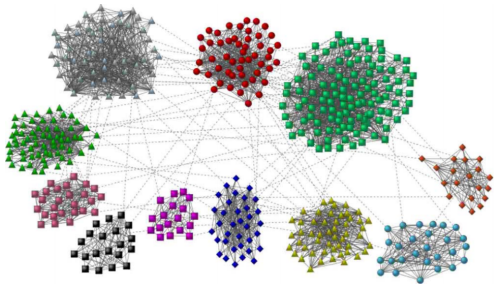


Figure 4: An LFR benchmark graph with 500 nodes. Image taken from Fortunato (2010).

Quality Functions Fortunato (2010) states that a quality function is "a function that assigns a number to each partition of a graph. In this way one can rank partitions based on their score given by the quality function." These quality functions are particularly important when there are no previously identified groups within the network, as is often the case, so there is no existing partition to compare to. Most quality functions are additive so that the quality Q of a whole community structure can be given by

$$Q(P) = \sum_{C \epsilon P} q(C),$$

where q is a quality function, and C is some community belonging to the full set of communities for this partition, P. For this project, we focus primarily on the quality function **modularity**. Modularity measures the fraction of edges in a network that connect vertices of the same type (i.e. within-community edges) minus the expected value of the same quantity in a network with the same community divisions but random connections between the vertices (Newman and Girvan, 2004). If the edges within communities are no better than random Q will be near to or less than 0, while a value of Q closer to the maximum of 1 indicates a strong community structure.

Optimising modularity is a frequently used technique for identifying communities in networks. However, modularity optimisation has been shown to have a resolution limit on the size of community structures found. Fortunato and Barthelemy (2007) tested modularity optimisation on a number of real world examples and found in many cases that they were unable to identify smaller communities beyond a certain size relative to the size of the input graph. This was a problem even when the real communities in the network were unambiguously defined. This limitation is something to consider both when using modularity to measure the quality of a community structure and when using modularity optimisation as a technique for identifying communities.

2.2 Community Hierarchy

Many algorithms will identify a hierarchical community structure, where a node may initially belong to a small community which itself is then grouped together with some other communities. For example, in figure 3 the first level of its community structure is shown by the 16 square boxes, with each being its own community. The second level is then identified by the four different colours of nodes. This hierarchical structure is often evident in real world networks. In a network of employees in a large organisation spread across multiple locations, one level of the hierarchy may be evident by grouping employees by location. Within each location group, the employees may also be grouped further by department or some other common function.

2.3 Existing Community Detection Software

At present, most software packages with community detection algorithms available are part of software libraries requiring programming knowledge to use. A number of these libraries are described in section 4.4.2 as they are considered for use within our web application. There are also a limited number of applications available with a graphical user interface for performing graph analysis, one of which is a web application that can perform community detection on networks.

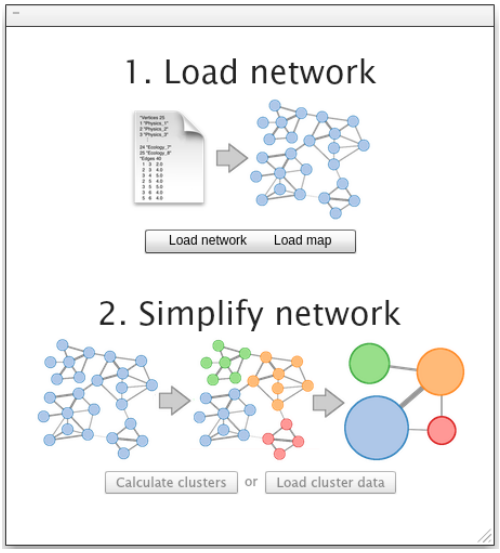


Figure 5: The input page for community detection in MapEquation.

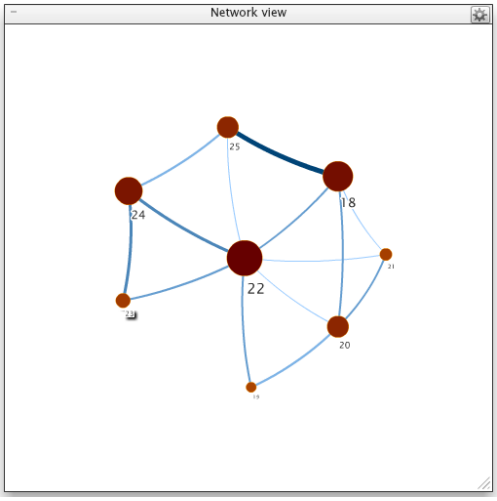


Figure 6: Community structure results in MapEquation.

MapEquation is an organisation focused on providing tools for simplifying and highlighting important structures in networks (Rosvall et al., 2009). One such tool is an online application for community detection (MapEquation, 2014). This application allows users to upload a graph in a specified format, detect communities in this graph and view the results in a hierarchical way. However, this application is limited to the use of a single community detection algorithm that is focused on information flow through the network as opposed to just the topological structure of the network. Furthermore, all graph processing and community detection is performed locally on the client's web page which limits the scalability of the application.

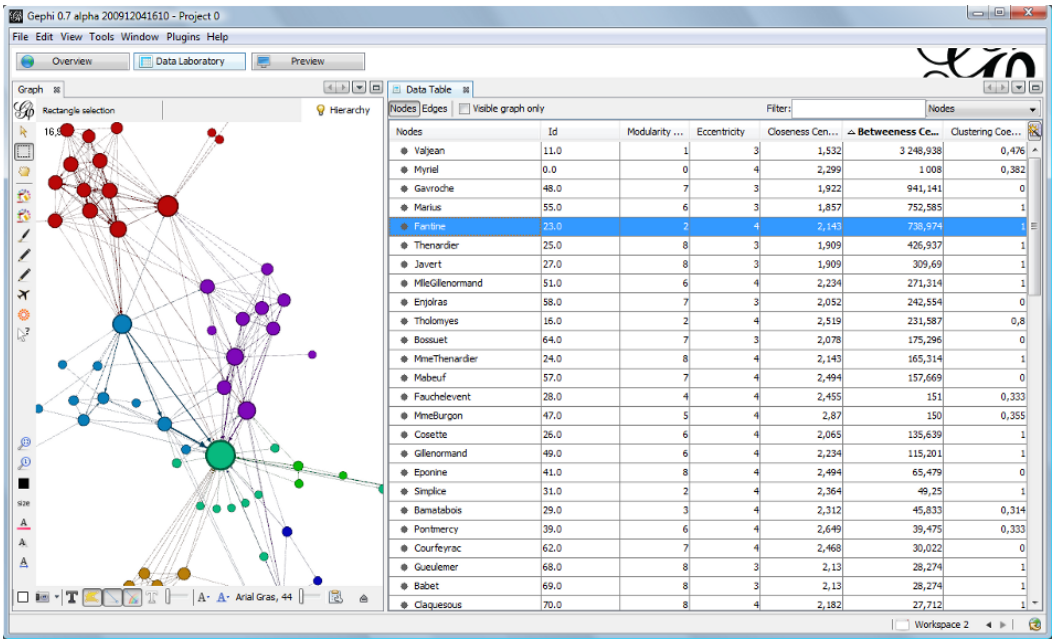


Figure 7: Screenshot of Gephi community detection results.

Gephi is "an interactive visualization and exploration platform for all kinds of networks and complex systems, dynamic and hierarchical graphs" (Gephi, 2014). Gephi includes one community detection algorithm, an implementation of the Louvain Method which is an algorithm we use in our own web application and is described in section 3.2. Gephi allows different levels of a community hierarchy to be viewed and also displays detailed information about each node in an input graph. This application is not a web application and all processing is done locally which, like MapEquation, limits the scalability of the application. This application will be used in section 7.1.1 to compare its performance against our own application.

Cytoscape is "an open source software platform for visualizing complex networks" (Cytoscape, 2014) that offers a series of plugins for performing numerous algorithms on networks, often with options to visualise the results. Some of these plugins include algorithms for community detection, including the clusterMaker plugin. ClusterMaker offers several community detection algorithms, the fastest of which is an algorithm called GLay. GLay is an algorithm based on the modularity optimisation algorithm by Girvan and Newman (2002) which has complexity $O(n^2m)$ where n is the number of nodes in the graph and m is the number of edges. This is the fastest algorithm included in any Cytoscape plugin we found, but is still quadratic in its complexity whereas the algorithms we use that are described in section 3 perform with better than quadratic complexity in terms of the size of the input graph. For this reason, we do not compare the performance of our own application with any Cytoscape plugins.

3 Chosen Community Detection Algorithms

In this section, we introduce the algorithms used in this application and discuss how they work and their potential performance on large input graphs. These algorithms have been shown to perform with better than quadratic complexity in the number of edges and nodes in the input graph. Further implementations details are described in section 5.

3.1 Label Propagation

Algorithm as described by Raghavan et al. (2007):

> "In label propagation every node is initialized with a unique label and at every step each node adopts the label that most of its neighbors currently have. In this iterative process densely connected groups of nodes form a consensus on a unique label to form communities."

Algorithm 1 Label propagation

```
for each node in graph:
  | set node label = node id
while label changes occur:
  | for each node in graph:
  |   | find most frequently occurring label among neighbours
  |   | set node label = most frequent neighbour label
```

In early iterations of this algorithm, small groups tend to form where a consensus is reached and a mutual label is adopted. As these dense groups grow, they start to come in to contact with other dense groups and compete for members. Nodes on the periphery of these dense groups will be at risk of joining a neighbouring dense group. At this point, the number of within-community edges in a group may prevent nodes being adopted by the other competing groups and this particular group will stabilise and local changes will no longer occur. Alternatively, the within-community edges will not be strong or numerous enough to prevent one group consuming the other and a larger community will form. Eventually a consensus will be reached across the whole graph.

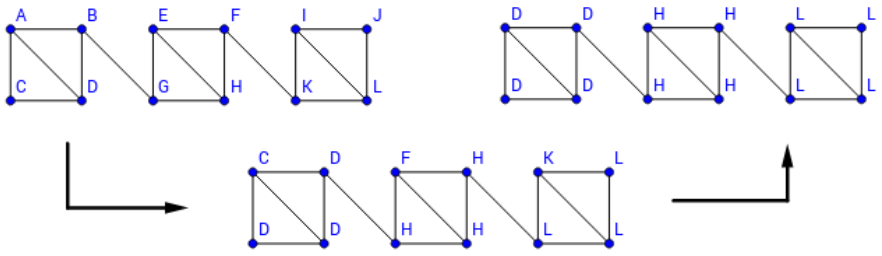


Figure 8: Example label propagation algorithm coming to a consensus after three total iterations. Note that the top row nodes were visited first, left to right, then the bottom row (explicitly, A B E F I J C D G H K L).

The order in which the nodes are processed in this algorithm is generally set randomly for the first iteration, and this order is then maintained for the remainder of the iterations. This ordering has an effect on the final result, as different orders will often produce differing final community structures. Raghavan et al. (2007) suggest performing the algorithm multiple times and comparing the results to find an aggregate solution. Due to the large size of the graphs looked at in this project, we do not consider using this aggregate solution though it may be of interest for future work.

Raghavan et al. claim that the algorithm runs with near-linear time complexity. The first iteration where initial labels are assigned requires $O(n)$ time, where n is the number of nodes in the graph and m is the number of edges. Each subsequent iteration requires $O(\triangle m)$ time, where $\triangle$ is the maximum degree of any node in the graph. In experiments, Raghavan et al. found that regardless of the number of nodes in the graph, 95% of nodes were correctly classified after 5 total iterations. However, they acknowledge that is it difficult to prove a mathematical convergence to any particular number of iterations required.

3.2 Louvain Method

This algorithm focuses on maximising the quality function modularity and produces a hierarchical community structure. As described by Blondel et al. (2008):

> "First, each node of the network is assigned to a different community. Then, for each node we evaluate the gain of modularity that would take place by removing this node from its community and by placing it in a neighbouring community. This process is applied repeatedly and sequentially for all nodes until no further improvement can be achieved. All communities are then contracted into super-nodes and the steps are repeated."

This algorithm uses two distinct steps in each iteration. First, local modularity optimisation is performed on the input graph as described in algorithm 2 and then the graph is contracted as described in algorithm 3. The combination of these two steps constitutes a single pass. The contraction step at the end of each pass produces a graph which is used as the input for next pass and is also considered as a level of the community hierarchy structure.

Algorithm 2 Louvain Method: modularity optimisation

```
for each node in graph:
    assign node to a community containing only itself

while modularity improvement occurs:
    for each node in graph:
        remove node from current community
        for each neighbouring community:
            calculate modularity gain if node joined this community
        if maximum modularity gain found > 0:
            insert node into new community
        else:
            insert node back into previous community
```

Algorithm 3 Louvain Method: contract communities

```
for each community:
    | contract all members into single node
    | set self loop = sum of all internal edges in this community
    | for each neighbouring community:
    |     | set external edge = sum of external edges between members
```

Much of the efficiency of this method rests on the ability to calculate modularity gain for each individual node in constant time. Blondel (2011) states that exact computational complexity for the Louvain Method is unknown, but that it appears to run in $O(n \log n)$ time. More importantly, Blondel et al. (2008) found the algorithm had excellent performance compared to other algorithms on very large graphs, some with more than a billion edges. The Louvain Method was shown to find communities in a graph with 118 million nodes and 1 billion edges in under 3 hours.

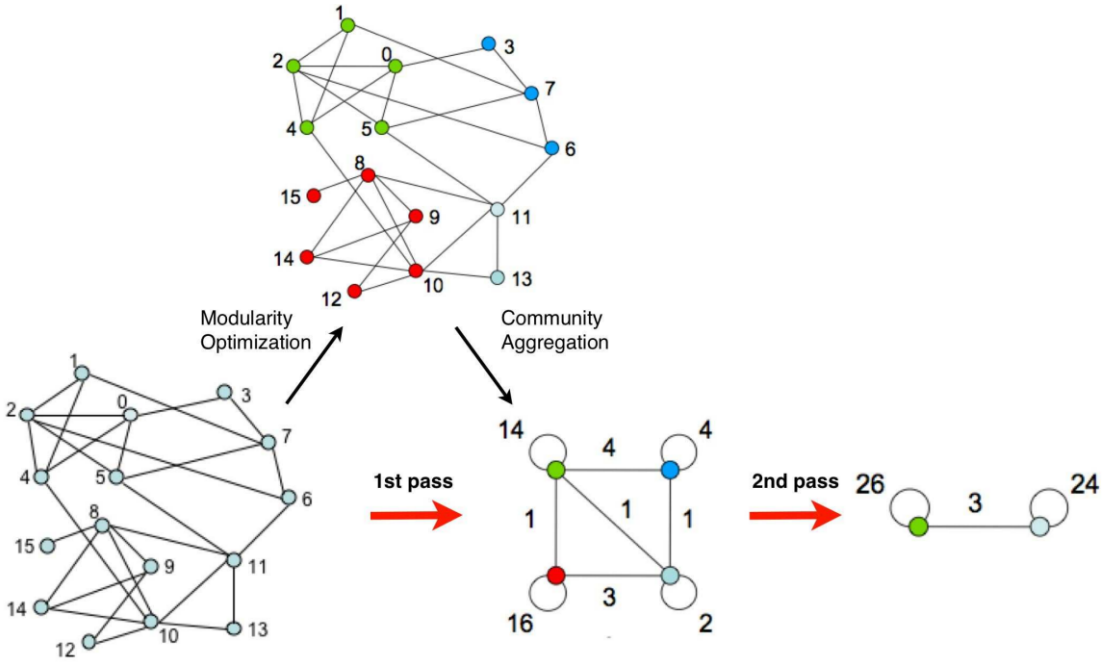


Figure 9: Example Louvain Method algorithm with two passes. Initially, 4 communities in the input graph are found with modularity optimisation and then these communities are contracted into single nodes and the process is repeated. Image taken from Blondel et al. (2008)

3.3 ORCA Reduction and ContrAction

This algorithm focuses on identifying dense neighbourhoods and produces a hierarchical community structure. As described by Delling et al. (2009):

> "Preliminarily prune the graph of irrelevant vertices, then, iteratively identify dense neighborhoods and contract them into super-nodes; after contraction repeat the

second step on the next hierarchy level or, if this fails, remove low-degree nodes and replace them by shortcuts. Do this until the whole graph is contracted."

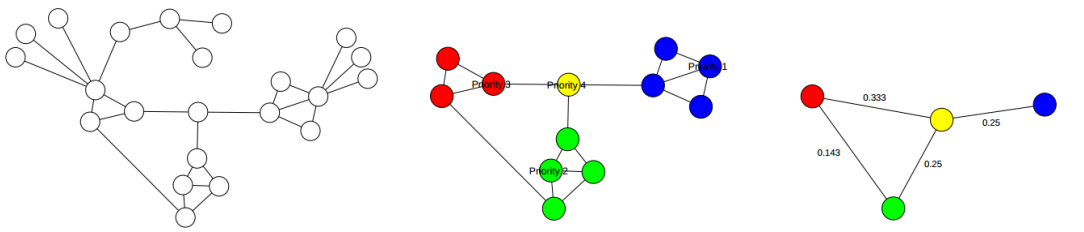
Algorithm 4 ORCA

```
Two–core reduction on input graph
while number of communities > 2:
  | find and contract dense regions in the graph
  | if contracted node count > (0.25 * previous node count):
  |   | perform graph shortcuts
  | else:
  |   | store current clustering
```

The initial step of this algorithm involves pruning irrelevant vertices which is done by taking the two-core of the graph. The two-core of a graph is the largest possible subgraph such that every vertex has a minimum degree of two (a minimum of two edges). These pruned vertices are contracted to the nearest community in the two-core. Once this is complete, dense regions within the resulting graph are identified. Delling et al. (2009) describe a dense region as follows:

> "...a dense region $R \subseteq V$ is a set of c nodes within distance d of some seed node v, such that each node $w \epsilon R$ is within distance at most d of at least $\mid N_d(v) \mid / \gamma$ other nodes of $N_d(v)$."

Distance d is most commonly 1, meaning only immediate neighbours are considered. The value γ defines how dense the region must be. If γ is small then nodes in the local region must be neighbours with many of the other nodes in the region. For example, with a γ value of 1 each node in the region must be neighbours with every other node in the region. Delling et al. suggest imposing a less strict value of 2 for this parameter. Once all of these regions are identified, they are then each given a priority based on the average weight within the region. Communities are then built from these regions in order of priority given to the regions. Nodes are often likely to be part of several different regions, in which case they are simply assigned to the region with the highest priority and then ignored in any lower priority region. Pseudo-code for this stage of the algorithm is found in appendix B.



(a) The input instance, first it will be reduced to its 2-core (b) Now, dense regions are identified (colors) (c) Each dense region is contracted

Figure 10: Example of ORCA algorithm taken from Delling et al. (2009)

After each time the global contraction is performed, it is most often found that the number of total communities has reduced significantly. In the rare case that the contracted graph is not

four times smaller than the previous iteration, Delling et al. (2009) suggest densifying the graph by looking at nodes with exactly two edges that connect two distinct communities. They then perform a shortcut by removing this connecting node and leaving a direct connection between the two distinct communities. This increases the likelihood of these two communities forming part of a dense neighbourhood in the next iteration. However, they acknowledge that this operation is rarely needed and it is not considered in our implementation of the algorithm.

Delling et al. (2009) state that the time complexity of the two core reduction stage is linear in terms of m and n and each iteration of finding and contracting dense regions has complexity $O(m + n(\triangle^2 + log\, n))$, where m is the number of edges, n is the number of nodes and $\triangle$ is the maximum degree of any node in the graph. Their experiments showed that ORCA performed slower than the Louvain Method on all tested input graphs. For a graph with 18 million nodes and half a billion edges, ORCA took over 2 hours where the Louvain Method took 7 minutes.

4 Application Specification and Design

In this section we set out a specification for the application with user requirements elicited from this specification. We then look at the design used to meet these requirements and discuss some of the decisions made in the process of this design.

4.1 Specification

A single-page web application that allows a user to upload a graph and visualise the community structure of that network. The application will allow users to upload graphs in one of two main formats and visualise the community structure of these graphs in their browser. These two formats are GML (Graph Modeling Language) and edge lists, where each line of a text file represents an edge in the graph (examples are shown in appendix C).

A user can choose from three different community detection algorithms: label propagation, the Louvain method, and ORCA Reduction and ContrAction. Once these graphs are processed with the user's chosen algorithm, an interactive visualisation of the resulting community structure is displayed in their web browser. These detection algorithms produce a hierarchical community structure where single vertices are grouped into fewer, larger communities as the the height of the hierarchy increases. The user can choose to view the structure at any level of this hierarchy. The user may download the resulting graph at any level of the hierarchy as a GML file and may also view the internal structure of an individual community at any level.

4.2 User Requirements

The system shall allow the user to:

1. Upload a file.
    - 1.1. Upload a graph as an edge list file with the standard format 'source target weight' where source and target are text data and weight is numeric.
    - 1.2. Upload a custom edge list file and specify what column separator character to use and which column contains the source, target and weight information of an edge.
    - 1.3. Upload a Graph Modeling Language file.
2. Detect communities.
    - 2.1. Detect communities with the Label Propagation algorithm.
    - 2.2. Detect communities with the Louvain Method algorithm.
    - 2.3. Detect communities with the ORCA Reduction and ContrAction algorithm.
3. Visualise community structures.
    - 3.1. View communities at any level of a graph's hierarchical community structure.
    - 3.2. View a list of nodes found in any single community.
    - 3.3. View the community structure of nodes found inside any single community.
    - 3.4. View labels on each node describing information about that node such as its name, size and other data assigned by the user.

3.5. Customise settings affecting how the graph is laid out on the page.

3.6. Navigate back to previous levels of the graph they have viewed in a session.

4. Download results.

4.1. Download a GML file for the graph at any level of its hierarchical community structure, with nodes labeled by what community they belong to.

4.2. Download a GML file for nodes within any single community at any level of its hierarchical community structure, with nodes labeled by what community they belong to.

4.3 Design considerations

The most expensive operation that the software must perform is the detection of communities. The detection can be done locally on the client with Javascript libraries or on a server with a number of server side technologies such as Java, PHP or Python.

Detection on the client

The server is only responsible for serving the single page and JavaScript code, so multiple users from different clients results in only a minimal extra load on the server. However, larger graphs require large memory. The size of any graph to be processed is bound by the memory capacity of the local device, which is likely to be limited compared to standard server hardware.

Detection on the server

More processing power and memory on server hardware allows for both larger input graphs and shorter detection times. However, as detection requires large amounts of memory and processing power, requests from multiple clients at once will require very large server resources.

Conclusion

As one of the primary goals for this software is to scale to handle very large graphs, the detection is to be completed on the server. Dealing with multiple concurrent requests can be offset by increasing server resources.

4.4 External Software Libraries

A series of decisions were made prior to beginning to design the system on which external libraries were to be used for key aspects of the application. This was done first as the libraries chosen had a significant effect on the final architecture of the application.

4.4.1 Web-based Visualisation Library

Cytoscape JS Cytoscape.js is a JavaScript library for network visualisation (CytoscapeJS, 2014). There are two main ways to display graphs on a page, either by specifying an exact position for each and every node in the graph or by laying out the graph dynamically using

some simulation. Cytoscape has built in functionality for dynamic visualisation of graphs using force directed simulations. This initially places the nodes on the page at random and then attempts to minimise the number of edges crossing and nodes overlapping by moving the nodes around the screen. No other graph visualisation library was found with this built in functionality.

4.4.2 Graph Processing Libraries

In choosing a library for processing graphs, a number of requirements were considered:

- Given only a limited amount of time to produce this software, the library should be available in a language which I had experience with. This is Java (preferable) or Python.
- Must be scalable for use with graphs containing millions of nodes and edges.

igraph Igraph is a "collection of network analysis tools with the emphasis on efficiency, portability and ease of use"(igraph, 2014). It is written in C++ with a high level interface available in Python. Many detection algorithms are already implemented, including the Louvain Method. Igraph can handle many input types including both GML and edge list formats. Input graph sizes are limited by main memory.

NetworkX NetworkX is "a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks" (NetworkX, 2014). A limited number of detection algorithms are available, but none of those being used in this web application are available. NetworkX can handle many input types including both GML and edge list formats. Input graph sizes are limited by main memory.

GraphLab GraphLab is a "graph-based, high performance, distributed computation framework written in C++"(GraphLab, 2014). This software also supports code written in Java and Python. Low et al. (2012) showed that applications created using GraphLab either outperformed or matched equivalent implementations using other distributed frameworks.

GraphChi In order to process very large graphs, more main memory is needed than that of an average personal computer. Therefore, many read and writes are needed to secondary storage which can be very computationally expensive. For computing efficiently on billions of edges, Kyrola et al. (2012) developed GraphChi, a disk-based system using a method that breaks large graphs up into small parts in a way that minimises the number of random read and writes to disk. Implementations are available in C++ and Java. In terms of performance, GraphChi has been shown to perform on par with many other distributed frameworks. GraphChi is a spin-off of GraphLab and shares a similar model of processing graphs.

Conclusion For maximum scalability and performance, GraphLab is the best choice of the above options. However, learning to use this software would be have been too big of an undertaking for this project. Therefore, GraphChi was used as it still performs well on large graphs and implementations written in GraphChi will be easier to port to GraphLab in any future work. Although igraph and NetworkX provide benefits in terms of built in detection algorithms and handling of multiple graph input types, this is offset by the scalability of GraphLab and GraphChi.

4.5 Overall Architecture

The basic architecture involves a single web page on the client whose content is updated by making Asynchronous JavaScript and XML (AJAX) requests to Java servlets based on the server. This allows the visualisation to be done on the client with Cytoscape JS and the community detection to be done on the server with GraphChi Java. The response sent from the server to the client is in JavaScript Object Notation (JSON) format as is needed by Cytoscape JS.

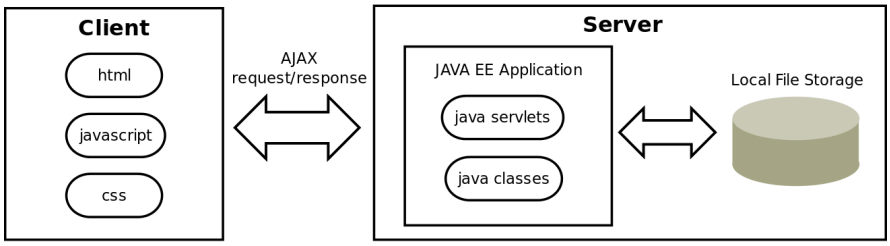


Figure 11: High level client-server architecture.

Figure 12 shows the main aspects of the system design. Roughly speaking, a user uploads a graph file to be visualised, the server detects communities within this graph and returns the graph in a format that can be visualised by Cytoscape JS on the client. The user can then navigate through different levels and communities within the graph's community structure which produces further requests for the server to process. The server then produces an updated graph to be visualised again by Cytoscape JS.

1. User makes request The user uploads a file in one of the specified formats, indicates what type of file this and which community detection algorithm they would like to use. This request is sent via AJAX to a Java servlet running on the server.

2. Parse request and store file The Java servlet parses the request to get the type of file and detection algorithm choice. The file they uploaded is then saved to local storage. A servlet session is also created at this stage which is an object associated with a specific user. This object can store data about the user and any requests they make, such as the location in local storage of the file they have uploaded.

3. Map external ids to internal ids Every graph file that a user uploads has to specify an id for each vertex in the graph. It is common for this id to be textual data such as a person's name or other identifier. GraphChi programs require the data type of the vertex id to be specified when writing the GraphChi program and requires this data to be of a fixed size in memory. Therefore it is more sensible to use a numeric value for the vertex id. GraphChi also creates a vertex in memory for every vertex id between 0 and the maximum value of any vertex id. For example, an input graph that simply contained three vertices with ids 0, 1 and 999 would create 996 extra vertices in memory to fill in the gap between 1 and 999. Therefore it is ideal for memory efficiency that numeric ids are sequential from 0 to n, where n is the number of vertices in the graph.

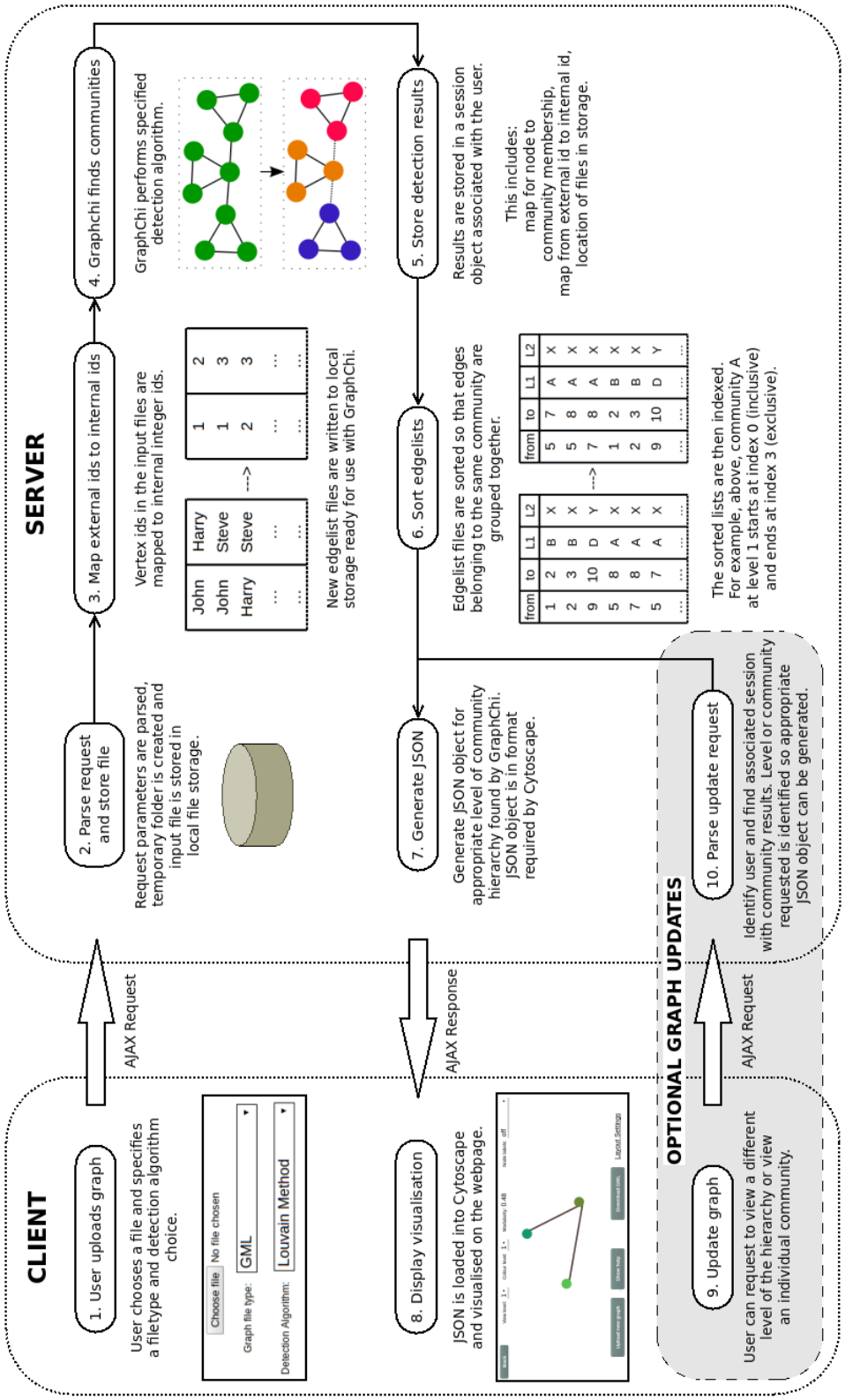


Figure 12: Flow through the program.

Rather than enforcing the user to upload only graphs in this format, we map all vertex ids in the input file, whether they are text or numeric, to an internal id from 0 to n. New edge list files are written with these newly mapped internal integer ids ready to be processed with GraphChi. A map relating external ids to internal ids is also stored in the user's session created in step 2.

4. GraphChi finds communities A different GraphChi program exists for each of the three possible detection algorithms. The correct program is initialised depending on what the user requested. Each of these programs implements a Java interface `DetectionProgram` that specifies it must take the location of an edge list file and produce a `GraphResult` object which contains details about the community structure found. The GraphChi program will also write a new edge list to local storage for each level of the community hierarchy it produces. In these edge lists, each vertex represents a community found by the algorithm and each edge is an edge between two communities rather than individual vertices.

5. Store detection results The `GraphResult` object created in step 4 is stored in the session created for the user in step 2. This allows future requests from the client to view different levels of a community hierarchy or individual communities.

6. Sort edge lists This step is taken to allow efficient retrieval of an individual community within the graph. In the initial edge list files input into GraphChi, the edges are not in any specific order. If the user wanted to visualise a particular community, the entire edge list would need to be traversed checking if each edge was an edge within the specified community. For very large graphs, this is an expensive operation. A community may have only 50 internal edges, but the entire graph may contain millions of edges.

Instead, the edge lists are sorted in order of the community that the edges belong to. An index is then built that specifies where each community starts and ends within the edge list file. Now a community of m edges can be read from the edge list file by only having to read m lines from that file instead of reading the entire file.

7. Generate JSON Cytoscape JS expects graphs to be input in a particular JSON format. Now the graph found by the detection algorithm is written in the specified JSON format, with each community labeled and coloured differently. This JSON is then sent via AJAX as a response to the client's initial request.

8. Display visualisation The client receives the JSON response and loads it into Cytoscape JS which displays the graph on the web page.

9-10. Navigating and updating the graph The user can choose to view the graph at different levels of its community hierarchy or view individual communities. When a user makes this request, the server identifies the session associated with the user so it can access the graph's edge list files and community structure. A new JSON object is built based on the user's request to be visualised again by the client.

5 Implementation and Testing

In this section we look at some of the implementation details of the design outlined in the previous section, particularly the implementations of the community detection algorithms using GraphChi. We then describe the automated testing procedure for the application.

5.1 Detection Algorithms

The three detection algorithms used in this application were introduced in section 3. GraphChi is used to implement these algorithms.

How GraphChi works GraphChi requires an edge list file for input. For an example edge list file see appendix C.1. In a GraphChi program, each vertex in the graph is loaded in turn and some update function is performed with that vertex. When a vertex is loaded, you only have access to data about the edges connected to that vertex and the ids of immediate neighbours. Each pass over the entire graph is counted as one iteration in GraphChi. For each GraphChi program you specify how many iterations to perform. You have the option to either visit every vertex on every iteration or to schedule vertices manually, where you can choose which vertices get visited on the next iteration.

How the community structure is stored The implementations of each algorithm share one common object that is always updated, an instance of `GraphStatus`. This object is used to store the community structure of the graph and contains the following key structures:

- A `HashMap` mapping a node in the input graph to a list of community ids, one for each level of the hierarchy. For example, a node with id 0 may map to the list of communities $\{4, 7\}$ meaning that in the first level of the hierarchy it belongs to a community with id 4 and a community with id 7 at the next level of the hierarchy.
- A list containing the modularity of the community structure at each level of the community hierarchy.

These structures are updated by a GraphChi program so the final community structure can be returned at the end. In addition to this `GraphStatus` object, when a GraphChi program finishes finding a new level of a community hierarchy structure a new edge list file is written to memory that represents the contracted graph. An example of this is shown in figure 13. This results in a contracted edge list file for each level of the community hierarchy found. These contracted edge lists are later used to build the JSON objects that are sent to client for Cytoscape JS to visualise.

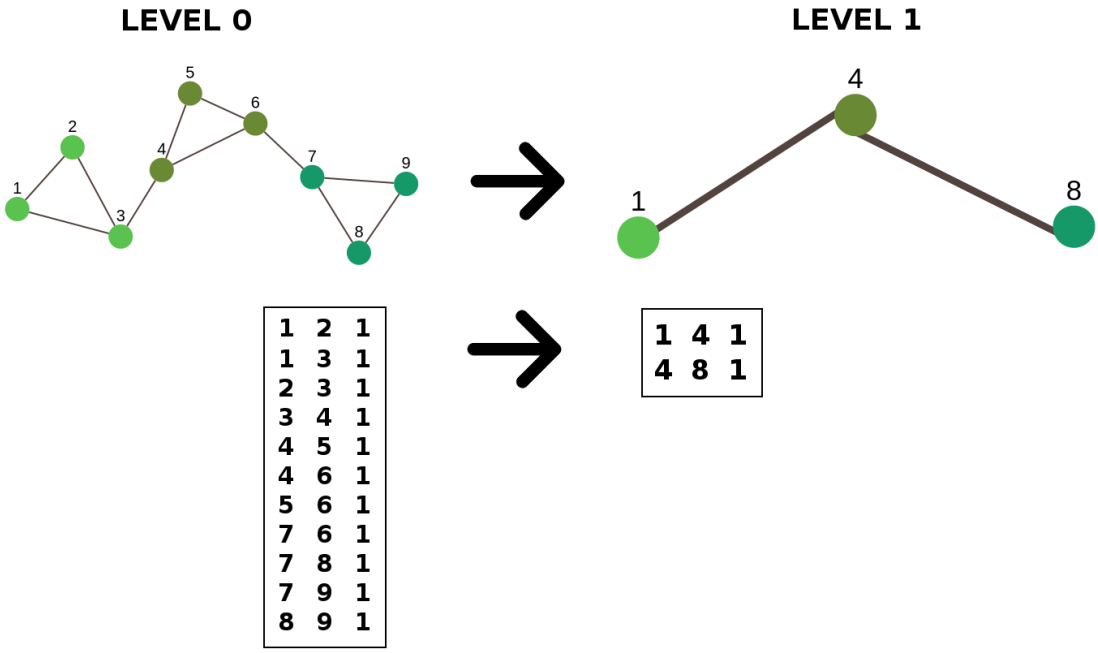


Figure 13: Example of a contracted edge list file after communities have been found. Level 0 is the input graph and level 1 is the contracted graph with a corresponding contracted edge list.

5.1.1 Label Propagation

This implementation did not deviate from the algorithm as described by Raghavan et al. (2007). When the label of a vertex is changed, GraphChi's scheduler is to used to schedule all the neighbours of this vertex to be updated on the next iteration. This prevents vertices unnecessarily being visited on an iteration if none of its neighbours have had their labels changed. This also allows the program to terminate when an iteration is reached with no vertices scheduled as this means that no labels changed on the previous iteration. This happens when a consensus is reached on labels and the algorithm is finished.

5.1.2 Louvain Method

Blondel (2011) provides source code of a C++ implementation of the Louvain Method which we adapted to use in GraphChi. The Louvain Method specifies that if on a single iteration any modularity improvement is made, then every node is checked again on the next iteration. Therefore, we do not use the scheduler provided by GraphChi and instead visit every node on every iteration.

One main difficulty was managing the community hierarchy produced by this algorithm. At the end of each main pass over the graph, all nodes that belong to a particular community are contracted to a single community to be used as input for the next pass. To accomplish this in GraphChi, at the end of each pass we write a new edge list file that represents the contracted graph as described above. If overall modularity is still improving, this new edge list file is then used as input again into the same program and the process is repeated.

5.1.3 ORCA

No source code for any implementation of ORCA was available at the time of developing this application. Given some ambiguities in the description of the algorithm by Delling et al. (2009), a number of assumptions and changes were made in our implementation.

1. They recommend looking for the dense neighbourhood around each node in a graph by visiting nearby nodes within a path distance of 1. However, it is unclear if this distance refers to a number of connections or the total weight of a path. For example, in figure 14 if we consider the number of connections, then only B is with distance 1 of A. However, if we consider the weight of the path, then both B and C are within distance 1 of A. In our implementation, we have interpreted this distance to mean the number of connections, ignoring weight. Tests with both methods found that this option produced results closer to the example results given by Delling et al.

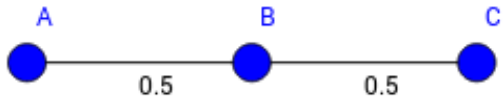


Figure 14: Example graph for ambiguous path length.

2. When contracting dense regions into so called super nodes, Delling et al. state that the edges between these super nodes should be "weight by their average adjacency to the region". No exact calculation for this is given and formula could be determined from the examples provided in the their work. To calculate the new edge weight between two super nodes A and B we used
$$\frac{\sum_{e \epsilon E(A,B)} \omega(e)}{(\mid A \mid + \mid B \mid)}$$
where $E(A, B)$ is the set of all edges connecting nodes within A and B and $\omega(e)$ is the weight of an edge. This produced values close to those in examples given by Delling et al.

3. Delling et al. found that at the end of later iterations there were often many single element nodes remaining and suggested contracting these nodes to the community they were most strongly connected to. However, we found that without doing this at every iteration we were almost always unable to find a meaningful community structure, even with the same example graphs that Delling et al. used. Delling et al. did not explicitly describe how they join these single element nodes, they only state that they join the node "to the cluster it is connected to most strongly". For this, we used local modularity optimisation as is used in the Louvain Method. Each single element vertex joins the neighbouring community such that the structure's modularity is improved the most.

The hierarchical results created by ORCA are dealt with in the same way as the Louvain Method where a new edge list is written at the end of each iteration and used as the input for the next iteration.

5.2 Graph Visualisation

Here we describe a number of steps involved in graph visualisation. First, we explain how results of a community structure are converted to a JSON object to be sent to Cytoscape JS.

We then describe how edge lists are sorted in order of community in order to allow visualisation of individual communities. Finally, we look at how Cytoscape JS is used to physically display and place the nodes and edges of the graph on the web page.

5.2.1 JSON Serialisation

Cytoscape JS is used to display graphs and requires a graph input to be in JSON format, an example of which is shown in appendix C.4. The edge list files that are produced as a result running the GraphChi programs are transformed into JSON format before being sent to the client. To do this, an edge list is read and data about each edge and node is stored in main memory. This data is stored in a single object that is composed of a number of other Java objects in a structure that corresponds to the required JSON format. A JSON serialiser is then used, which is capable of turning Java objects into strings representing JSON objects that can be sent to the client in an AJAX response.

A number of open source JSON serialisers are available in Java including Google's Gson, Jackon's JSON Processor and Boon JSON. Each of these three options were tested in the application. Jackson and Boon were found to be considerably quicker than Gson. Jackson is used in the final application as it has more extensive support and reliability than Boon which is a relatively new library with just a single developer.

Most of this process is identical for both visualising an entire level of a community hierarchy and visualising a single community within a level of a community hierarchy. For an entire level, every edge in the corresponding edge list is loaded into memory and processed. For a single community, only the edges that are edges within that community are loaded into memory. This is the motivation for sorting edge lists by community as explained in the next section.

5.2.2 Sorting Edge Lists

Sorting is done to allow efficient extraction of individual communities as explained in section 4.5. An edge list with edges in no specific order is sorted so that edges within the same community are located together in the edge list file. The application makes use of Java's `TreeSet` implementation which is a set that maintains order whenever an element is inserted. This insertion takes $O(log\, n)$ time where n is the size of the set so far. To insert all n edges in an edge list it therefore takes $O(nlog\, n)$ time overall to sort an edge list. Once all edges have been inserted into the `TreeSet`, the `TreeSet` is traversed in order and each edge is written to a new file which becomes the sorted edge list file.

This current implementation leaves a restriction on how big the edge list file to be sorted can be as all edges must fit in main memory in the `TreeSet` object. To make this solution scalable for a greater number of edges, an external merge sort can be implemented. This involves splitting the unsorted edge list file into p partitions that each fit into main memory. Each of these p partitions is loaded in turn into main memory and sorted independently of the other partitions. Once all partitions are sorted, the first part of each sorted partition is loaded into main memory and the first value from each partition is compared. The lowest value is then written to the final sorted edge list file. This process is repeated and when all edges from the first part of a sorted partition are taken, the next part of that partition is loaded into memory. This is repeated until all sorted partitions have had all their edges loaded into memory and written into the final sorted edge list file.

5.2.3 Cytoscape Graph Layout

Once a graph has been loaded into Cytoscape JS from a JSON file, Cytoscape can the lay the graph out on the web page. There are two options for this, either every node is given an exact coordinate on the page or a dynamic layout can be used where each node is placed randomly on the page and a simulation is run and nodes are moved about on the page until some condition is met. For the visualisation to be aesthetically pleasing, it is necessary for as few edges in the graph crossing as possible and for nodes of the same community to be close to each other on the page. Setting the exact coordinates of each node such that these conditions are met for any given graph is a difficult task to complete algorithmically. Instead, a dynamic layout is used. Cytoscape has a built in layout option that uses a force-directed simulation which is used in our implementation.

5.3 Testing

The application has automated testing on two different levels with unit testing at a module level and functional testing at a system level. The overall output of the application, the community structures themselves, are difficult to test in an automated fashion as there is no clearly defined, expected result for non trivial inputs. It is even legitimate and common for the exact same input and parameters to produce two different results that could be deemed correct. For this reason, the validity of the output and validity of the algorithms themselves are covered in section 7.

5.3.1 Unit Testing

Each package within the Java application on the server has associated JUnit tests. These unit tests test the basic functionality of the public methods of each class. The aim of these unit tests is to test only the class under test and should be independent of any other classes in the application. In cases where one class was dependent on another, a stub version of the dependency was created to return an expected behaviour. This stub class was either created manually or created using the open source library Mockito (2014). When code is changed in a particular module, the unit test for that module is run to check for any errors the change might have introduced.

5.3.2 Functional Testing

The Java application on the server also has a number of functional tests covering the whole server side application from the client making a request to receiving a response. In the real application, every request from the client creates an `httpRequest` object on the server which contains request parameters and an output stream to write a response to. To test this automatically without the need for a real client, Mockito (2014) is used to create a mock `httpRequest` object which can be used in JUnit. Given this setup, the system is tested with very basic graphs. These graphs are trivial graph in terms of community detection, instead the purpose of the tests are to check that the server as a whole takes an expected input and returns an expected output without error. When changes are made to the code anywhere on the server, these tests are run and if errors are found, the unit tests can help identify the location of the error.

6 User Interface

In this section we describe the web application's user interface and show how communities found by the detection algorithms are displayed to the user.

6.1 Navigating the Detection Results

Firstly we use the example of Zachary's karate club network (Zachary, 1977), a university karate club with 34 members whose members split into two separate groups after an internal dispute. Figure 15 shows views where individual nodes are grouped together into communities. The visual size of each community is related to the number of members in each community, while the width of each edge relates to the number of connections between two communities. Wider edges indicate a stronger connection between two communities, relative to the connections between other communities. Figure 16 shows the bottom level of the community structure where every node from the input graph is displayed. These levels are navigable by drop down menus at the top of the web page.

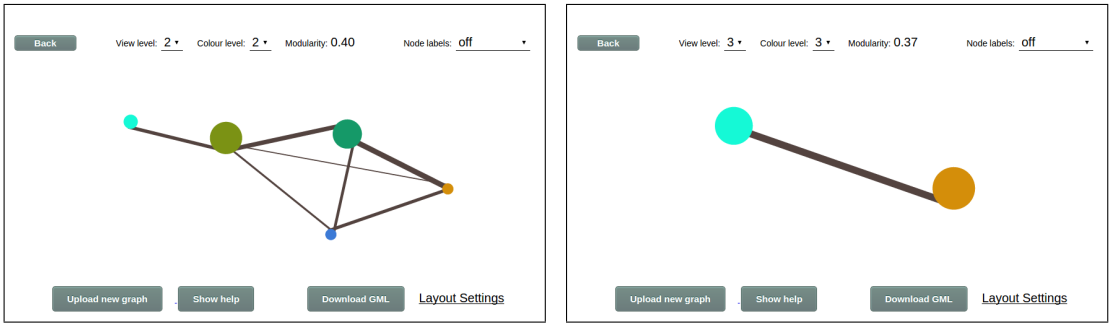


Figure 15: Example of interface for results of community detection on the karate club. The algorithm returned 3 different levels in the community hierarchy. The left view shows level 2 where the nodes are grouped into five communities. The right view shows level 3, the top level, where all 34 nodes are grouped into two communities.

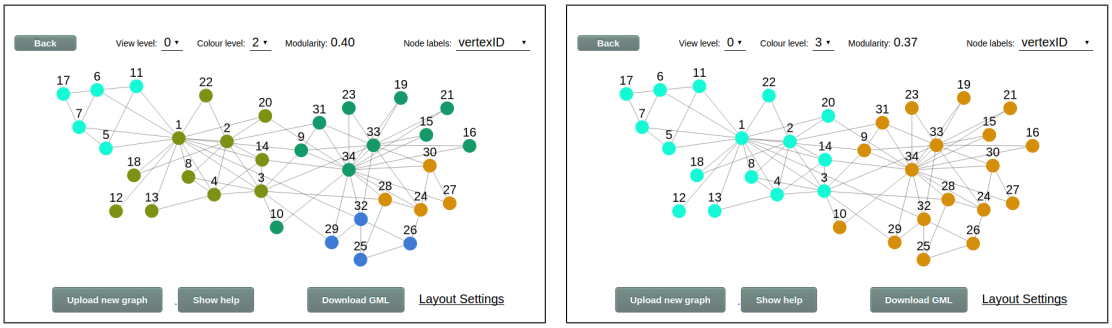


Figure 16: This view displays level 0, where every node in the original graph is shown as a separate node in the display. The left view shows each node in the graph coloured at level 2, where there are 5 different communities. The right view shows each node in the graph coloured at the next level, level 3, where the previous 5 communities now grouped into 2 larger communities.

6.2 Navigating Individual Communities

Here we use the example of a network representing the topology of the Western States Power Grid of the United States (Watts and Strogatz, 1998). This is a graph containing approximately 5000 nodes and 7000 edges.

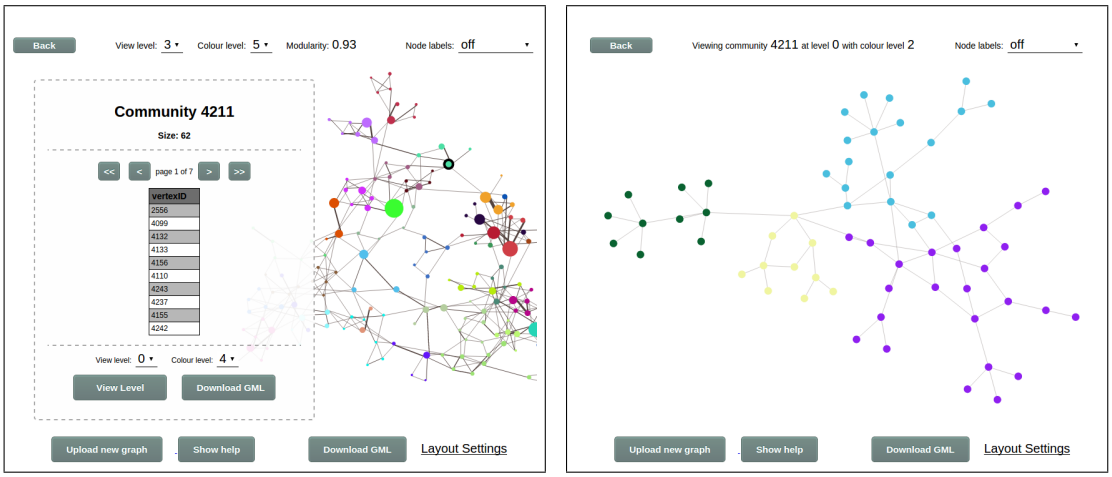


Figure 17: Example of interface for results of community detection on a power grid network. Left view shows the list of nodes displayed when a particular community is selected. Right view shows the result of viewing this particular community alone at the bottom level, where every individual node from the input graph is displayed.

A community structure with 5 levels found with the Louvain Method detection algorithm is used here. Figure 17 shows a user having selected a particular community in the graph. Clicking on a community brings up a display with a table listing data about all members of that community and gives options to navigate into this community and see its internal structure. If a chosen community has smaller communities within it, these can also be visualised with the use of colours as is shown in this example.

6.3 User Help

To help users understand how to navigate through the community structure, instructions are available for each drop down option and button on the page as shown in figure 18. Experienced users can choose to hide these instructions.

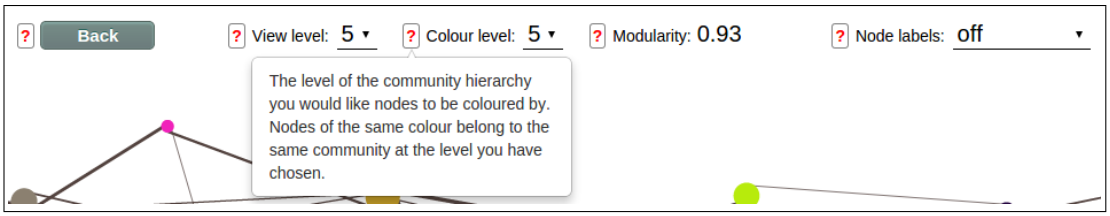


Figure 18: Help options for the top menu bar. Each question mark displays information when hovered over.

6.4 Progress Status

To give users an indication of how long the processing of a graph will take a status bar is displayed while they wait for the processing to finish. However, this bar is an estimation based purely on the size of the input graph and the community detection algorithm chosen rather than an accurate depiction of how much progress has been made.

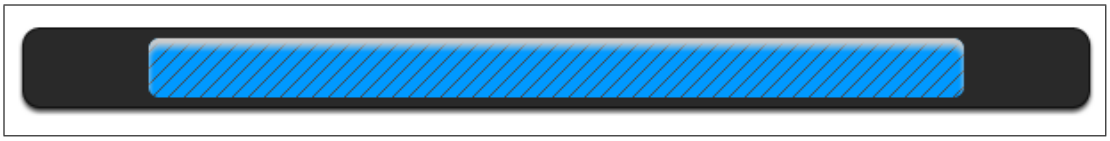


Figure 19: Status bar for graph processing progress.

7 Results

In this section we look at the speed performance of the application and the quality of communities detected. Further to this, we briefly investigate the notion of ground-truth communities and their relationship to the topological community structures found by the application.

7.1 Performance - Speed and Quality

In order to test the performance of the implementations of the detection algorithms, we input graphs from the from real world networks and generated benchmark graphs.

7.1.1 Real world networks

The time and modularity performance of the three algorithms are tested here on a number of real world networks. As a benchmark, the graph analysis software package igraph described in section 4.4.2 is used. This package is written in C++ and provides an implementation of the Louvain Method. The application Gephi described in section 2.3 is also used here as it too provides an implementation of the Louvain Method. The graphs considered include three online social networks and M. musculus, a dense biological network which is not completely connected. The same amount of main memory was allocated to each algorithm and to each application.

Network (nodes/edges)	Facebook (4k/88k)	Twitter (81k/1.7m)	M.Musculus (20k/5.3m)	Google+ (107k/13.6m)
Label Propagation	0.76	0.78	0.27	0.39
Louvain Method	0.83	0.73	0.35	0.44
ORCA	0.71	0.65	fail	fail
Louvain (Igraph)	0.83	0.79	0.32	0.44
Louvain (Gephi)	0.83	fail	fail	fail

Table 1: Summary of modularity scores on example real world networks.

Table 1 shows the modularity scores of each algorithm and table 2 shows the processing times of each algorithm for both community detection and further processing completed to make graph visualisation efficient. The modularity found in each graph by our application is comparable to the igraph implementation. We find that the detection time of our implementation of the Louvain Method takes 2-5x longer than igraph's implementation. This is expected as our implementation is written in Java as opposed to C++ and GraphChi also has to load data in and out of external memory many times whereas igraph loads all the graph data into main memory only once. This loading of data from external memory is not the most efficient method for graphs that do fit in main memory, but would allow graphs larger than main memory to be processed whereas igraph would not be capable of processing these graphs.

Gephi failed on all but the Facebook network. Gephi was able to load the Twitter network, but then ran out of memory when performing the Louvain Method. Gephi then ran out of memory when loading both the biological network and the Google plus network and therefore the Louvain Method could not even be started. This memory issue appears to be a result of Gephi always trying to visualise each individual node and edge of an input graph when the input graph is initially loaded.

Network (nodes/edges)	Facebook (4k/88k)		Twitter (81k/1.7m)		M.Musculus (20k/5.3m)		Google+ (107k/13.6m)	
	Detection	Post	Detection	Post	Detection	Post	Detection	Post
Label Propagation	<1s	<1s	11s	12s	26s	62s	93s	349s
Louvain Method	1s	<1s	16s	15s	63s	45s	112s	243s
ORCA	4s	<1s	58s	26s	fail		fail	
Louvain (Igraph)	<1s	-	3s	-	18s	-	78s	-
Louvain (Gephi)	1s	-	fail		fail		fail	

Table 2: Summary of processing times for example real world networks. Post processing includes sorting edges as described in section 5.2.2.

Label propagation was always the quickest of the three algorithms whilst ORCA was always the slowest. Our implementation of ORCA was also unable to process disconnected graphs, so failed on the biological network. ORCA ran out of memory on the largest graph, Google plus. On four out of five occasions, the Louvain Method found the structure with the best modularity score.

As the size of the graph increased, the time taken for visualisation processing increased at a greater rate than the time for detection. This time was dominated by the process of sorting edges so that individual communities could be visualised efficiently. Further investigation is needed into why the sorting took so much longer on the large instances. Furthermore, when graphs of much greater size than those used in table 2 were tested, label propagation and the Louvain Method successfully found communities but we ran out of main memory when sorting edge lists and found sorting time to be much greater than the detection time.

7.1.2 Generated benchmark graphs

We used the LFR benchmark graphs described in section 2.1 to test the detection speed of each of the three detection algorithms. We used the software provided by Lancichinetti et al. (2008) to generate graphs of varying sizes. The software allows you to specify several parameters including the number of nodes in the graph, the average degree of each node and a mixing parameter, μ. Roughly speaking, the mixing parameter defines how strong the community structure in the graph is. Specifically, each node will "share a fraction $1-\mu$ of its links with the other nodes of its community and a fraction μ with the other nodes of the network" (Lancichinetti et al., 2008). This results in a threshold at $\mu = 0.5$ where values above this result in graphs that do not have a strong community structure and nodes within a community will have more neighbours outside its own community than within it. Therefore we choose a value of $\mu = 0.4$ where the structures are still strong but sufficiently challenging for the algorithms to identify.

We generated graphs ranging from 100,000 nodes in size to 1,000,000 million nodes. In each case the average degree was 10, resulting in graphs with 10 times as many edges as nodes. To check that the algorithms find the community structure correctly, we calculate the **normalized mutual information** (NMI) between the actual community structure and the structure found by the algorithm. We use a variant of normalized mutual information introduced by Danon et al. (2005). This produces a score between zero and one. 1 indicates that the two partitions are identical while 0 indicates no mutual information between the two partitions.

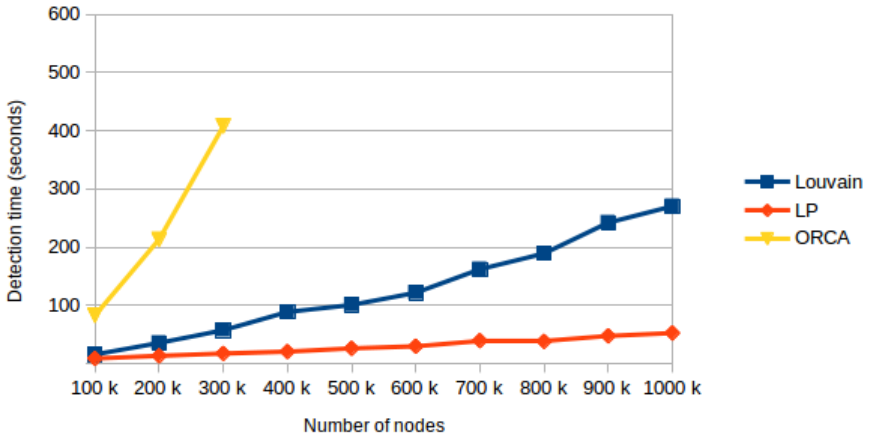


Figure 20: Time taken for detection algorithms to find communities. Each graph had ten times as many edges as nodes.

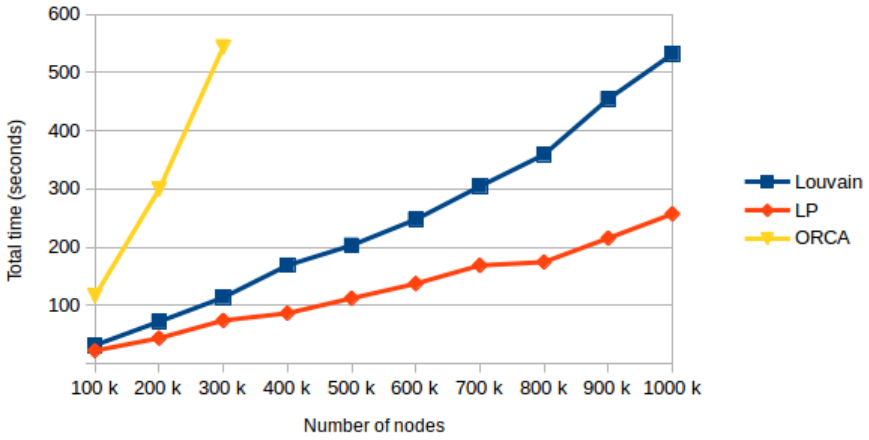


Figure 21: Time taken for detection algorithms to find communities and prepare the results for visualisation. Each graph had ten times as many edges as nodes.

In every case where a result was found, the NMI score was between 0.99 and 1, indicating that the correct community structures were always found. As with the real world networks, label propagation was always the fastest to find the community structure. Label propagation ranged from taking 8 seconds on the smallest graph to 52 seconds on the largest, only a fourfold increase in time despite the graph size increasing by a factor of 10. The Louvain method took between 15 and 270 seconds to perform detection. ORCA performed very poorly overall, taking significantly longer than both label propagation and the Louvain Method on the smallest instances. An out of memory error occurred on the graphs with more than 300,000 nodes, by which point the

community detection was already taking longer than both of the other other algorithms took on the largest graph.

In almost all cases, the total processing time, including graph pre and post processing described in section 4.5, took roughly twice as long as the detection time alone. This is in contrast with the results found on real world networks suggesting there are a number of factors that affect the relationship between these times other than just total graph size.

These results give an initial idea on the comparative speeds of each of our algorithms on graphs of varying sizes up to 1,000,000 nodes and and 10,000,000 edges. However, in future work we would like to investigate how other properties of the graph and its community structure affect the speed of the detection. Firstly, how the density (the ratio of nodes to edges) of the graph affects the speed. Secondly, how the size and number of communities in the graph affects the speed. Lancichinetti et al. (2008) found that these properties had a significant impact on the ability of the algorithms to correctly identify community structures. It would be useful to know how these properties affect speed performance as well, particularly the density of the graph as this property is known before any community detection is carried out and could help inform which algorithm is the most efficient to use on particularly dense or sparse graphs.

7.2 Detected Communities Versus Ground Truth

7.2.1 Background

One primary motivation for identifying topological community structures within networks is to group nodes with similar (non-topological) properties or functions. Hric et al. (2014) recently argued that most scholars have assumed structural communities found with commonly used detection algorithms represent groups with such similar properties or functions. Using large networks with established ground-truth communities provided by Yang and Leskovec (2012), Hric et al. found that traditional community detection algorithms often failed to detect these ground-truth communities.

Ground-truth Yang and Leskovec (2012) distinguish between groups and structures in a network. Nodes in the networks in the network can share some common function, where this can be a common role, affiliation or attribute. For example, communities in social networks may share work places, common hobbies, interests and affiliations. These communities with shared common functions are what Yang and Leskovec consider to be groups and this is the basis on which they identify ground-truth communities.

Here we test the implementations of our detection algorithms against a small subset of these networks with ground-truth communities. In doing so, we aim to test the conclusions of Hric et al. and to also further test the validity of the application.

7.2.2 Method

There are a number of traditional networks with ground truth communities that have commonly been used to evaluation detection algorithms and found structures. Here we look at three of them. A single karate club whose members split into two factions after a dispute between key members (Zachary, 1977). A network of American football games in a single season, with groups identified by the conference within the league that the teams belong to (Girvan and Newman, 2002). Finally, a network of hyperlinks between political blogs in the United States, with groups identified by political alignment.

Name	#Nodes	#Edges	#Communities	Description of community nature
karate	34	78	2	membership after group split
football	115	615	12	team scheduling groups
polbooks	105	441	2	political alignment
dblp	317080	1049866	13472	publication venues
amazon	366997	1231439	29432	product categories

Table 3: Basic properties of data sets used in this analysis. All information taken from Hric et al. (2014).

In addition to these smaller networks, we considered two much larger networks found by Yang and Leskovec (2012). These are a network of co-authorships between computer scientists, with groups identified by publication venue and an Amazon product co-purchasing network. These two networks contain overlapping communities, where some nodes are considered to be part of several different ground truth communities, and some nodes are not assigned to any ground truth communities. As the community detection algorithms we used do not identify overlapping communities and assign every single node to a single community we have to perform some preprocessing on the community structures before comparing their mutual information.

Similar to the methods of Hric et al. (2014), only nodes which belong to a community in both structures are considered in the comparison. Where a node belongs to multiple ground truth communities, we assign it only to a single one of those communities and remove it from any others. The ground truth communities are ranked by how significant Yang and Leskovec (2012) considered them to be, so a node is assigned to the group it belongs to with the highest rank. With the remaining non-overlapping ground truth communities and the communities found by a detection algorithm we calculate the normalized mutual information as described in section 7.1.2.

7.2.3 Results

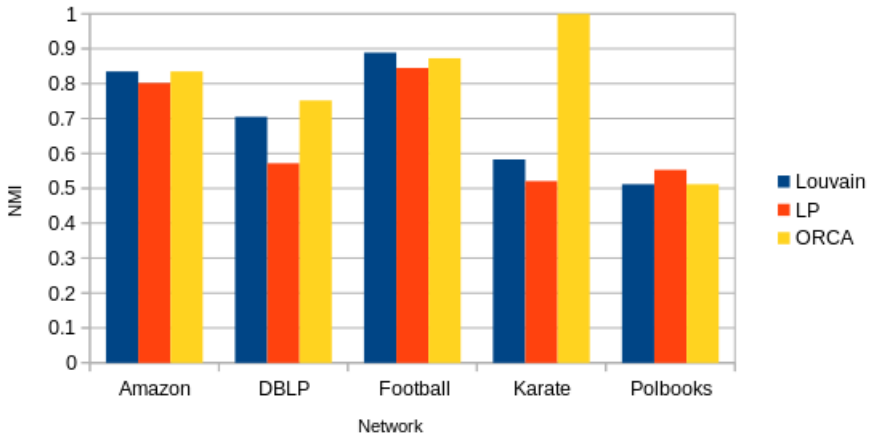


Figure 22: NMI analysis results. For Louvain and ORCA which return hierarchical community structures, the level with best NMI score is used.

The values for NMI were in all but one case very close to those found by Hric et al. (2014) on the same graphs. The exception was the co-authorship network, dblp. Hric et al. were able only to find an NMI score as high as 0.3 compared to scores of 0.57 to 0.75 found here. Given that Hric et al. also used the Louvain Method and a variation of label propagation, this large difference may be due to different methods of assigning nodes that were part of overlapping ground truth communities.

The Louvain Method and ORCA had very similar results in most cases and generally performed better than label propagation. This is largely due to label propagation returning larger and fewer communities than the Louvain Method and ORCA are capable of finding at the lowest level of their community hierarchy.

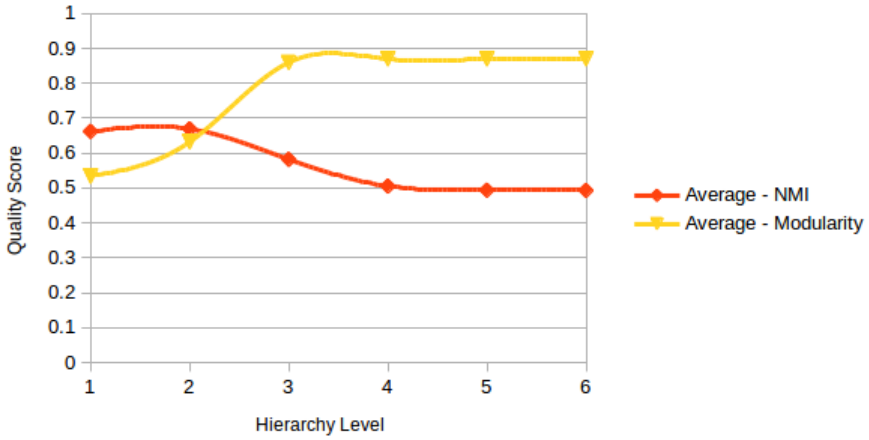


Figure 23: Comparison of modularity score with NMI across different levels of the hierarchy for the Louvain Method.

Given that we use modularity as a quality function it is worth looking at the relationship between modularity and NMI in our results. As the level of a community hierarchy increases, modularity increases as expected but NMI scores decrease. This is primarily due to the size of the ground truth communities being smaller than those found by the detection algorithms. The found communities are at their smallest at the lowest level of the hierarchy, and then are grouped together as the level increases. The found communities are already larger than the ground truth communities at the lowest level, then get even larger in comparison to the ground truth communities so the NMI score decreases. This gives further evidence to the resolution limit of modularity described in section 2.1.

7.3 Discussion

Testing these algorithms on real world networks with and without ground-truth communities, and on generated benchmark graphs has given an insight into the advantages and limitations of each algorithm and the application as a whole. Label propagation always performed community detection in the least amount of time, while overall the Louvain Method found the best results in terms of modularity and identifying ground-truth communities. Although ORCA was able to match the Louvain Method when identifying ground-truth communities it failed on large graphs

due to out of memory errors. This suggests a problem with our implementation as Delling et al. (2009) were able to handle much larger graphs with the same algorithm.

Efforts to make visualisation of the community structures more efficient appear to have been relatively unsuccessful, causing the application to fail on graphs with upwards of approximately 15 million edges. To solve this problem, either the sorting algorithm would need to be made more memory and time efficient or an alternative approach used to tackle the problem of navigating efficiently through different levels and communities within a community structure.

8 Project Management

In this section we describe the software process model used in the development of the web application and list the key tasks completed throughout the project.

8.1 Software Process Model

An agile development methodology was used when developing this application using roughly weekly iterations. As many new technologies were being learned throughout the process, it would have been very difficult to follow a waterfall development process where all analysis, specification and design was done before any implementation. Instead, a preliminary list of user requirements was set out at the start of the development process. These requirements were then ordered and prioritised by importance.

On a weekly basis, using the priority of the requirements, a number of tasks were derived such that this collection of tasks would take approximately one week to complete. At the end of each week, the result of work on these tasks would be used as feedback to amend the requirements and prioritise future tasks for the next week. Information gathered whilst learning new software packages and technologies had a significant effect on the overall design and structure of the architecture. This agile process was a key factor in being able to quickly adapt to these changes.

The web-based project management application Trello was used to manage and organise the tasks in this project.

8.2 Project Timeline

Week 1

- Preliminary reading on graph theory and community detection algorithms.
- Implementation of label propagation algorithm in GraphChi Java.

Week 2

- Creation of prototype HTML page for graph visualisation with Cytoscape JS.
- Allowed user to upload edge list file through the web page ready for processing by GraphChi.
- Initial implementation of JSON serialisation in Java for results of label propagation algorithm to be used by Cytoscape JS.

Week 3

- Implementation of navigating into individual communities of single nodes from parent communities in Cytoscape JS.
- Implementation of Louvain Method in GraphChi.

Week 4

- Added storage of community detection results in Java servlet sessions to allow for future requests to see different levels of a hierarchical community structure.
- Implementation of algorithm for sorting edge list files to allow for efficient building of JSON results for individual communities.

Week 5

- Added support to allow user to upload graph as a GML file to then be converted to an edge list file ready for processing with GraphChi.
- Implementation of GML file creation to allow user to download results as a GML file.
- Begun implementation of ORCA algorithm in GraphChi.

Week 6

- Continued implementation of ORCA algorithm in GraphChi including own modifications.

Week 7

- Added support for text vertex ids and decimal edge weight values where previously only integer ids and weights were allowed.
- Redesigned user interface.

Week 8

- Allowed user to upload custom edge lists in a character-separated value format with a choice of column separator and column numbers for edge source, target and weight.
- Added unit tests for individual modules and functional tests for server classes in Java.

Week 9

- Conducted research into mutual information between communities found by implemented detection algorithms and ground truth communities.
- Tested performance and validity of algorithms with real world and benchmark graphs.

Weeks 10 and 11

- Preparation of project presentation and dissertation.

9 Evaluation

In this section we evaluate the software development process outlined in section 8 and the overall approach to the project. In addition to this, we evaluate the final web application based on the results found in section 7 and look at the strengths and limitations of the final product.

9.1 Process

A number of new technologies and libraries were learnt throughout the process of developing this application. These include Java servlet technology and the graph processing library GraphChi. I also had no prior experience with JavaScript, HTML or CSS which were all heavily used on the client side of the application. Once the basics of these languages were understood, I learned how to use the graph visualisation library Cytoscape JS. The agile methodology used in the development process allowed this learning to take place whilst still working towards the requirements of the final product.

One limitation of the process was the relationship between the development of unit test code and production code. Production code was almost always written before test code. This resulted in difficulties when writing tests as classes were often tightly coupled which made testing each class individually quite difficult. Had a more test driven development approach been taken, this problem would have been highlighted earlier and potentially avoided. Instead, a significant amount of time was devoted to refactoring code to make it more testable and loosely coupled.

9.2 Product

Each of the user requirements set out in section 4 are satisfied by the final product. However, with regards to the scalability of the final product for very large graphs the results shown in section 7 are mixed. On graphs with up to 10 million edges the product performs well in terms of both speed and quality of communities found for two of the three community detection algorithms. The speed was comparable to other previous implementations of community detection algorithms and initial investigations into ground truth communities showed that each of the algorithms were capable of finding labeled groups in real world networks with a good degree of accuracy.

Scalability Through testing the product on large instances in section 7 we found that although it performed well on graphs up to 1 million nodes and 10 million edges in size, it failed on graphs substantially larger than this. This limitation was caused by the graph pre and post processing done to make navigation of the resulting community structure efficient. Not only did this processing take substantially longer than detection on large graphs, it also caused out of memory errors on very large graphs causing the program to crash. In future work, either an alternative approach to processing the community structure results is needed to make navigation efficient or the algorithms currently used need to be made more efficient.

ORCA Our implementation of ORCA has been shown to not be very memory efficient. Each edge in the graph is stored in main memory several times which is the cause of the out of memory errors found in section 7. ORCA and the Louvain method were shown to have comparative performance in terms of the quality of communities detected, but ORCA always took significantly longer. Although these two algorithms were published at around the same time, only the Louvain Method has been frequently used in other work. While the Louvain Method has been

implemented many times in many languages and is used in multiple different graph processing libraries, no implementation of ORCA is publicly available or used in any graph processing library that we could find. This contrast in support for the two algorithms combined with our own results suggest that it may not be worth pursuing ORCA any further.

Visualisation In some cases, the dynamic force-directed simulation used to display community structures found by the application does not produce very aesthetically pleasing results. For example, multiple communities will often overlap all in one place on the screen. A number of settings are available to change in the simulation such as how much each node attempts to push away from other nodes and fixing the physical length of edges in the visualisation. Unfortunately, we could not identify algorithmically when these situations would occur. Our solution was then to always use default settings for the simulation to start with and allow the user to amend these settings if the appearance of the community structure was not clear to them. This results in a trial and error process for the user to try and get the results that are aesthetically pleasing. In future, we would like to identify the relationship between a community structure and the optimal simulation settings for that structure so that we can take this responsibility away from the user.

Progress status Another limitation of the web application is the progress bar shown in section 6.4. The progress bar gives the user an indication as to how long they will have to wait before processing of a graph will be complete. However, this is currently just a time estimation based on the size of the input graph and the algorithm chosen for community detection. Therefore this progress bar could be misleading and very inaccurate for some input graphs. Giving the user an accurate remaining time based on the actual progress of a community detection algorithm is a difficult task. A more suitable approach in the future may be to give the user an indication of what work has taken place so far rather than giving them an estimation of what time is left. For example, the status could be updated after the initial graph pre-processing is complete and after each iteration in an algorithm. This would give a more honest indication to the user about the progress of the application.

10 Summary

In this work we set out to provide three main contributions: a web application for visualising community structures, implementations of three community detection algorithms, and an investigation into the performance of community detection algorithms.

A client-server web application We built a web application that allows users to visit a web page and upload graph files of a number of different types and visualise the community structure of these graphs. The software performs community detection on a server and returns the result to be visualised interactively on the client's web page and informs the user of the resulting modularity, a quality measure used to quantify community structure quality. We found that this application performed well on graphs with up to 1 million nodes and 10 million edges with regards to finding community structures but often struggled when visualising such large graphs.

In future work, we would like to extend the application to allow further input types and provide more quantitative details about the community structures to the users such as quality measures other than modularity. We also wish to look further at the visualisation of the graphs on the web page. Currently, all graphs are visualised using a dynamic force-directed simulated layout already implemented in our chosen visualisation library, Cytoscape JS. Further work is needed to identify how the settings used with this simulation can be tailored to each individual graph result so that results are always shown in an aesthetically pleasing way.

Implementations of detection algorithms We used the graph processing library GraphChi to implement three community detection algorithms: label propagation, the Louvain Method and ORCA Reduction and ContrAction. We found that GraphChi was a suitable library for label propagation and the Louvain Method but we struggled to efficiently implement ORCA with GraphChi. In future work, we would like to implement these algorithms in GraphLab, a distributed framework for graph processing provided by the authors of GraphChi that is based on the same model of processing graphs. This would aim to improve both the scalability and speed of the algorithms and could easily be integrated into the current web application.

Investigation into algorithm performance We tested the algorithms on real world networks and artificially generated networks to investigate speed performance and community structure quality. We found that label propagation was consistently the fastest at finding a community structure, while the Louvain Method and ORCA generally found more accurate and higher quality results. Although ORCA found good quality community structures, it failed on graphs with more than 3 million edges and always took significantly longer than the other two algorithms on smaller graphs. We also looked at both the time taken for the detection algorithms alone to run and the time taken for the graph pre and post processing required for efficient visualisation of the algorithm results. In doing so, we found that the graph pre and post processing time was the limiting factor when using graphs much larger than 10 million edges.

In future work we would like look at alternative methods of processing graph results to make visualisation efficient so that we process and investigate graphs with many more than 10 million edges. We also aim to do a more thorough investigation of the speed performance of these algorithms. In particular, using artificially generated graphs, we wish to investigate the effect graph density has on speed performance in addition to just the overall graph size.

References

Adamic, L. A. and Glance, N. (2005). The political blogosphere and the 2004 us election: divided they blog. In *Proceedings of the 3rd international workshop on Link discovery*, pages 36–43. ACM.

Blondel, V. (2011). Louvain method for community detection. `http://perso.uclouvain.be/vincent.blondel/research/louvain.html`. Accessed: 2014-08-21.

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008.

Cytoscape (2014). Cytoscape. `http://www.cytoscape.org/`. Accessed: 2014-09-06.

CytoscapeJS (2014). Cytoscape.js. `http://cytoscape.github.io/cytoscape.js/`. Accessed: 2014-08-21.

Danon, L., Diaz-Guilera, A., Duch, J., and Arenas, A. (2005). Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(09):P09008.

Della Rossa, F., Dercole, F., and Piccardi, C. (2013). Profiling core-periphery network structure by random walkers. *Scientific reports*, 3.

Delling, D., Görke, R., Schulz, C., and Wagner, D. (2009). Orca reduction and contraction graph clustering. In *Algorithmic Aspects in Information and Management*, pages 152–165. Springer.

Feld, S. L. (1981). The focused organization of social ties. *American journal of sociology*, pages 1015–1035.

Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3):75–174.

Fortunato, S. and Barthelemy, M. (2007). Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41.

Gephi (2014). Gephi - the open graph viz platform. `https://gephi.github.io/`. Accessed: 2014-08-26.

Girvan, M. and Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826.

Google (2014). google-gson. `https://code.google.com/p/google-gson/`. Accessed: 2014-08-26.

GraphLab (2014). Graphlab.org | graphlab open source. `http://graphlab.org/projects/index.html`. Accessed: 2014-08-20.

Hightower, R. (2014). Boon json. `https://github.com/RichardHightower/boon`. Accessed: 2014-08-26.

Hric, D., Darst, R. K., and Fortunato, S. (2014). Community detection in networks: structural clusters versus ground truth. *arXiv preprint arXiv:1406.0146*.

igraph (2014). igraph | the network analysis package. `http://igraph.org/redirect.html`. Accessed: 2014-08-20.

Jackson (2014). Jackson project home @github. `https://github.com/FasterXML/jackson`. Accessed: 2014-08-26.

Kyrola, A., Blelloch, G. E., and Guestrin, C. (2012). Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46.

Lancichinetti, A. and Fortunato, S. (2009). Community detection algorithms: a comparative analysis. *Physical review E*, 80(5):056117.

Lancichinetti, A., Fortunato, S., and Radicchi, F. (2008). Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4):046110.

Leskovec, J. and Mcauley, J. J. (2012). Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547.

Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. (2012). Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727.

Lusseau, D., Schneider, K., Boisseau, O. J., Haase, P., Slooten, E., and Dawson, S. M. (2003). The bottlenose dolphin community of doubtful sound features a large proportion of long-lasting associations. *Behavioral Ecology and Sociobiology*, 54(4):396–405.

MapEquation (2014). mapequation.org - network navigator. `http://www.mapequation.org/apps/NetworkNavigator.html`. Accessed: 2014-08-21.

Mockito (2014). Mockito. `https://github.com/mockito/mockito`. Accessed: 2014-08-24.

NetworkX (2014). Networkx. `https://networkx.github.io/`. Accessed: 2014-08-20.

Newman, M. E. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113.

Raghavan, U. N., Albert, R., and Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106.

Rosvall, M., Axelsson, D., and Bergstrom, C. T. (2009). The map equation. *The European Physical Journal-Special Topics*, 178(1):13–23.

Stanford University (2009). Stanford large network dataset collection. `http://snap.stanford.edu/data/`.

Trello (2014). Trello. `https://trello.com/`. Accessed: 2014-08-26.

Watts, D. J. and Strogatz, S. H. (1998). Collective dynamics of small-world networks. *nature*, 393(6684):440–442.

Yang, J. and Leskovec, J. (2012). Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM.

Zachary, W. W. (1977). An information flow model for conflict and fission in small groups. *Journal of anthropological research*, pages 452–473.

A CD Instructions

A.1 File structure

```
/
├── andrewWalker-report.pdf
├── example
│   └── //sample data files
└── webapp
    ├── css
    │   └── //web page style sheets
    ├── js
    │   └── //web page JavaScript files
    ├── src
    │   └── //Java source files
    ├── test
    │   └── //Java unit test source files
    ├── WEB-INF
    │   ├── classes
    │   │   └── //compiled Java classes
    │   ├── lib
    │   │   └── //external Java libraries used by application
    │   └── tmp
    │       └── //folder used for user uploaded graphs in application
    ├── commDetection.war
    └── index.html
```

A.2 Running the software

To run this software, Apache Tomcat (http://tomcat.apache.org/index.html) version 6 or later is required on your machine. Once Tomcat is installed, our application's `war` file needs to be put into the `webapps` directory of Tomcat. There are two options for retrieving this war file:

1. Use compiled `commDetection.war` file shown in the file structure above.
2. In Linux, compile the `war` file from source files by navigating to the `webapps` folder and running the command `jar -cvf commDetection.war` to create a new compiled `war` file.

Once this `war` file is deployed by placing it in the `webapps` folder, the application should be available in your browser at `http://localhost:8080/commDetection`. Example input files are available in the `data` folder on the CD and help instructions for using the software are available within the application itself.

Note GraphChi, the graph processing library used by this application, requires Java's heap size to be set to a minimum of 256mb. Default Tomcat configurations may set the heap size to be smaller than this. Exact instructions for increasing this limit depend on the version of Tomcat and the operating system it is installed on. For example, for Tomcat7 on Linux, edit the `/etc/default/tomcat7` file, find the line beginning with `JAVA_OPTS="-Djava.awt.headless=true` and change `-Xmx128m` to `-Xms256m -Xmx512m`.

B ORCA Reduction and ContrAction Pseudo-code

Algorithm 5 ORCA: find and contract dense regions in the graph

```
for each node in graph:
 | identify dense neighbourhood

for each neighbourhood:
 | rank by average internal edge weight

for each node in graph:
 | if node not already assigned:
 | | assign node to highest ranked neighbourhood it is a member of

for each community:
 | contract all members into single node
 | for each neighbouring community:
 | | set external edge = average edge weight between these commmunities
```

C Graph Input Types

This is a simple graph example with the corresponding input for a default edge list, a custom edge list using character separated variables and the Graph Modeling Language format.

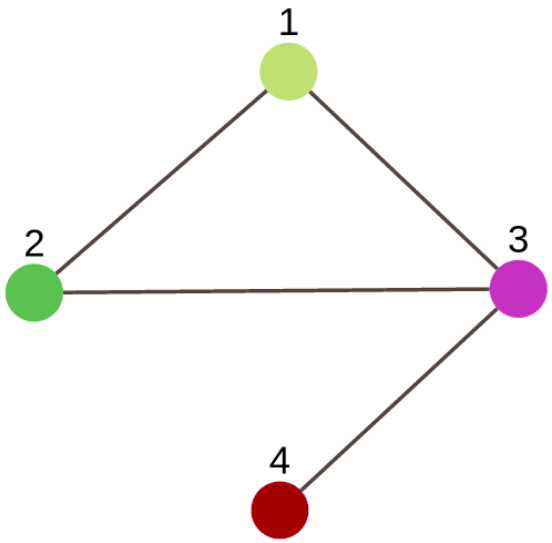


Figure 24: Example input graph

C.1 Edge List

Example in the format 'source target weight'.

```
1 2 1
1 3 1
2 3 1
3 4 1
```

C.2 Custom Edge List - Character Separated Values

Example in the format 'weight,source,target'.

```
1,1,2
1,1,3
1,2,3
1,3,4
```

C.3 Graph Modeling Language

```
graph [
  node [
    id 1
  ]
  node [
    id 2
  ]
  node [
    id 3
  ]
  node [
    id 4
  ]
  edge [
    source 1
    target 2
    value 1.0
  ]
  edge [
    source 1
    target 3
    value 1.0
  ]
  edge [
    source 2
    target 3
    value 1.0
  ]
  edge [
    source 3
    target 4
    value 1.0
  ]
]
```

C.4 Cytoscape JSON Format

```
{ nodes:  [
  { data:  { id:  '1'} },
  { data:  { id:  '2'} },
  { data:  { id:  '3'} },
  { data:  { id:  '4'} }
],
edges:  [
  { data:  { source:  '1', target:  '2' } },
  { data:  { source:  '1', target:  '3' } },
  { data:  { source:  '2', target:  '3' } },
  { data:  { source:  '3', target:  '4' } }
]}
```