University of Strathclyde

Department of Computer and Information Sciences

Developing Recommendation Systems for Movies Using Graph Database Clustering

Author:
Daniel Moorhead

Supervisor:
Dr Clemens Kupke

Co-Supervisor:
Mr William Wallace

A thesis submitted for the degree of

MSc Advanced Computer Science with Big Data

August 18, 2019

Declaration

This dissertation is submitted in part fulfillment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [✓] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 18074.

I confirm that I wish this to be assessed as a Type 1 2 3 4 [✓ 5] Dissertation (please circle)

Signature:

[illegible]

Date: 18/08/2019

Abstract

This paper covered the idea of developing recommendation systems from clustering graph databases. Graph databases are a form of storing data where each piece of information is stored as a node. Each node is connected by an edge that is defined by some relationship in the data between the nodes. This paper investigated two methods of representing movie data in a graph database sing the TMDb 5000 Movie database. The first was by representing each node as either an actor, a movie, a director or a genre. The relationships were between each non-movie node and the director/genre/actor node associated with it. The second graph contained only movie nodes that were connected by shared actors, directors, genres and keywords.

Clustering algorithms were tested on these graphs to see if the clusters found were suitable to be used for a recommendation system where a user is associated a cluster that contains movies they have enjoyed. Other movies in this cluster would then be recommended to the user by the system. The clustering algorithms investigate where Connected Components, Strongly Connected Components, Louvain Modularity and Label Propagation

The graph databases were successfully constructed, however none of the algorithms were able to produce clusters that could be used for a recommendation system.

Acknowledgements

I would like to acknowledge Billy Wallace and Clemence Kupke, whose help and guidance were pivotal to the success of this project. And Esha Khimji, who got me through it all.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation was on Developing Recommendation Systems for Movies Using Graph Database Clustering. The objective of this dissertation was to investigate the use of graph databases as a method of developing a movie recommendation system.

With the rise of online movie viewing, it is useful to develop a system that can provide recommendations to users [Cha et al., 2009]. Here a new approach involving graph databases was used. Graph databases are a method of connecting data using graph theory. Connecting pieces of data, called nodes, with a relationship between them allows for a structure that can be used to store, query and represent data [Huang et al., 2002].

This piece of work investigated the unique properties of graph databases that separate them from different types of databases. The investigation looked into how these properties could be used to develop a system that can recommend movies to a user, based movies that they had previously enjoyed. A system using this kind of database has yet to be implemented into a commercial system, and so this an area of recommendation system development that has the potential for a lot of further research.

This paper looks specifically into the graph database property called clustering. As the data is represented geometrically, it is possible to find pieces of data that have grouped together to form clusters using many different algorithms. It is hypothesised that when clustering movie data will reveal groups of similar movies.

This paper attempts to answer two questions: Can a database containing data about movies be successfully represented as a graph database? Can graph clustering algorithms be used to find sets of movies that can be used as a recommendation system?

The area of mathematics that graph databases fall into is called graph theory. Throughout its history there have been many algorithms developed to cluster graphs. Although the algorithms have been developed to detect clustering, there is currently no research into these algorithms ability to find clusters that can be used on movie data to build a recommendation system. There has, in fact, been no research on the effect of these algorithms on movie data at all. It is not yet known how the clusters found with these algorithms will be structured [Schaeffer, 2007].

Although graph databases have been used for the development of recommendation systems, there has been no published research on using clustering as a technique.

Graph databases have proven to be very useful in the commercial world. The world wide web is a graph database. The development of the internet into this graph has been essential for everyday use, Google's search system is based around a graph algorithm called page rank [Schafer et al., 2001]. The use of graph algorithms here has been revolutionary to how society runs. It is clear that graph databases are a powerful tool in the information world, and so it is very much worthwhile to investigate the use of graph databases, graph theory and graph algorithms with the goal of developing a recommendation system.

There has been a lot of time and money put unto the current recommendation systems that have been developed by large movie streaming services such as Amazon and Netflix. The systems in place work by recommending movies based on the similarities of other users, this is called collaborative filtering [Smith and Linden, 2017]. The main difference with the approach investigated here is that the recommendation is based solely on the users own preferences. Another problem that may be alleviated with the current recommendation system is that as new movies come out, the database can be immediately updated and re clustered, allowing them to be recommended easily.

Collaborative filtering however, must allow time for the new movie to be seen by enough people to make a meaningful recommendation. This is called the cold start problem. As this is a common problem in recommendation systems, it is important that research is done to best account for that [Lam et al., 2008]. Hence, there is a lot of merit in this research.

Chapter 2

Literature Review

2.1 Recommendation Systems

The Recommender System Handbook defines recommender systems as "software tools and techniques providing users with suggestions for items a user may wish to utilize" [Ricci et al., 2011]. This book is considered the core piece of writing on the subject and considers recommendation systems as a tool for commercial use. The book states that it is an incredibly important tool for the average person in the information age.

Jussi Karlgren was the first person to bring up the idea of a recommender system. In hi2 1990 Columbia University technical report, "The Systems Development and Artificial Intelligence Laboratory, An Algebra for Recommendations" 1990 he described the idea of a a recommender system as "digital bookshelf". [Karlgren, 1994] In this report, Karlgren posits that, when searching for a document in a bookshelf a researcher will come across an interesting title and pull it out, regardless of their main objective, before carrying on with their original search. The result of this is that documents of similar interest gather into small clusters. A a result, people using the bookshelf easily come across documents of inters, when looking for specific document. However, equally spaced documents in a hierarchy of computer folders has "no corners where interesting documents could collect".

A very early description of what is now referred to as a recommendation system was developed in Elaine Rich's 1979 paper User Modelling via Stereotypes[Rich, 1979]. The problem being solved her was developing computer systems that treat users as individuals. The idea of Elaine rich to exploit the stereotypes of a user to provide a service unique to that user. An example of the exploitation of stereotypes is a librarian interacting with a customer. If a customer where to ask for a book on China, they must use stereotypes that have been developed in their mind to make the correct recommendation. If the customer was a child, a learning age book on China and its culture would be most suitable. A young adult may prefer a book on tourism etc.

The potential system that was suggested here was a "virtual librarian" named Grundy. A conversation with Grundy would allow Grundy to build a profile of terms describing the user. Each term would has a numerical value (-5 to 5) to a values of the stereotypes of that term. For example, a user with the term "Feminist" in their profile would have +5 added to their "Sex Tolerant" profile term.

While this system performed well on the small group of testers, it was never tested on a wider group of people. It is unlikely that the system would be fully functional as there is evidence of stereotypes being a poor method of predicting traits [Devine, 1989]. However, the building of profiles has become a major part of many modern recommendation systems, such as collaborative filtering [Lam et al., 2008].

The idea of computers being used as librarians has been expanded upon by many companies such as Apple and Amazon, with their computer assistants Siri and Alexa [Assefi et al., 2015][Chung et al., 2017].

2.2 Importance of Search Recommendation Systems

Search recommendation systems have become the backbone of many major services. Amazon, for example, has based its business model on the ability to recommend new items based on customers preferences. In terms of media consumption, Netflix and YouTube similarly gained great popularity using similar algorithms. This was discussed in Smith and Lindens' paper 'Two Decades of Recommender Systems at Amazon" [Smith and Linden, 2017]. This paper is written from the experience working at Amazon, a major company based on recommender systems. It mentions how recommender systems have become such an important technology that there was a prize of 1,000,000 given to the best collaborative filtering method called the Netflix Prize.

Schafer et al. in their 2001 paper E-commerce recommendation applications [Schafer et al., 2001] showed the importance of recommender systems in E-commerce by describing the three ways in which it improves an E-commerce system. The first is Converting Browsers into Buyers where Schafer et al claims that a visitor to a Website will often visit the site without purchasing anything. A recommender systems can help customers find the necessary products with greater ease.

Secondly, they claim that recommendation systems increase cross-sell. These systems can recommend products that a customer would not typically be aware of, but would still be of interest. This can result in a new purchase. Thirdly they claim a recommendation system will build customer loyalty. As these systems rely on building relationships between the site and the user in order to better the recommendations. The more information input from the customer, the better tailor the recommendation. This symbiotic relationship means the more frequently used sites will provide better recommendations. In turn increasing the number of visits in comparison to other sites.

While the E-commerce this paper refers to is focused on online retails, these same concepts can apply to businesses that use movie recommendations. Media streaming services such as Netflix and Amazon Prime do not sell particular products, instead are centered around subscriptions that provide unlimited movies and television shows [Pogue, 2007][?].

With the increase of the number of streaming services, building customer loyalty is essential. In a 2017 report, A.J. Montenary found that seventy percent of people viewing media did so through a streaming service , with forty percent of television viewing done through a streaming service. This is twice as many as five years prior [Montenieri, 2018]. As no single product is being sold, these services rely entirely on customer loyalty which, as Schafer suggests, can be significantly improved by enhancing their recommendation system.

2.2.1 Types of recommendation system

Currently the recommender systems widely used are based on collaborative systems. That is, systems are built based on a collection of data of people's viewing/ buying habits. Items are recommended based on the habits of people who also purchased that item. However, this becomes an issue when dealing with new shows products as there is no purchasing information on it. This is called the cold start problem [Lam et al., 2008].

There are three variations of the cold start problem, New community, new item and new user [Bobadilla et al., 2012] . The new community cold start problem refers to the problem of providing recommendations in a start-up. Although there is a catalogue of items, there are very few users to interact with with it. This lack of user interaction causes problems when trying to create recommendation. This would not effect the development of a recommender system of movies based on graph clustering, as there is a large community of movie fans.

The new item problem largely effects collaboration based systems. When a new item is added, the system may not have enough information needed to recommend it. In terms of collaborative filtering, the missing information is the users who enjoy it. No collaborative filtering system can be built from an item with no collaboration. This issue is not applicable to graph cluster recommendation systems. As long as the full information about the product is provided, the information can be added as nodes and relationships. The clusters can then be adapted to include the new item.

The issue that effects many recommendation systems is the new user is the new user cold start problem. With collaborative filtering, there a new user has no information about their preferences. And so it is difficult t recommend items based on shared preferences. While this would also effect a graph database recommender system based on clustering, in terms of movie recommendation this can be averted by requesting an initial list of preferred movies.

Another approach to filtering based recommendation systems is content based filtering. This is based on the data of an item rather than data of the user. This system compares the information about items the user has already bought/ used with the data in new items, as described in Peter Brusilovsky's 2007 book The Adaptive Web [Brusilovsky and Millán, 2007]. As this is based on item data, it is free from the effect of the new item cold start problem as items already have data needed to describe them. It does, however, succumb to the new user problem, as the system needs to develop information about items that are of the users preference.

There are also multi-media recommendation systems that combine different forms of information into a recommendation. Such as the multiple-criteria decision-analysis (MCDA) field system developed by Kleanthi Lakiotaki et al in their 2011 paper Multicriteria User Modeling in Recommender Systems [Lakiotaki et al., 2011]. This system is based on performing user-grouping before applying collaborative filtering. This is method can be reasonably applied to current technologies, as user data is gathered when creating an account to use these services. The initial stage used k-means clustering as a method of grouping the data of users. This is further study of a concept first discussed in Lyle H. Ungar et al's 1998 paper Clustering Methods for Collaborative Filtering [Ungar and Foster, 1998]. The effect was shown to be an improvement on basic collaborative filtering methods, although further research should be done with larger groups of people.

2.2.2 Graph Databases

A graph database is a NoSQL database that is structured as a graph. It consists of nodes that represent the entities of the data that are interconnected by edges that represent the relationships between said entities [Angles and Gutierrez, 2008]. Graph databases are extremely useful for heavily interconnected data and as such have generated much interest in the fields of biology, computer science and information technologies. Yoon et al showed in the 2017 paper "Use of Graph Database for the Integration of Heterogeneous Biological Data" [Yoon et al., 2017] the success of graph database in complex biological problems. The use of this technology in such a complicated task shows the power that these databases have. This is a heavily cited paper with strong repeatable results that show a true potential for this technology.

An basic graph database is shown in Figure 2.1. This was taken from the official neo4j website; a major company in graph database software. As this company is a major source of graph database technology, it is a useful source of graph database information [Sasaki, 2018]

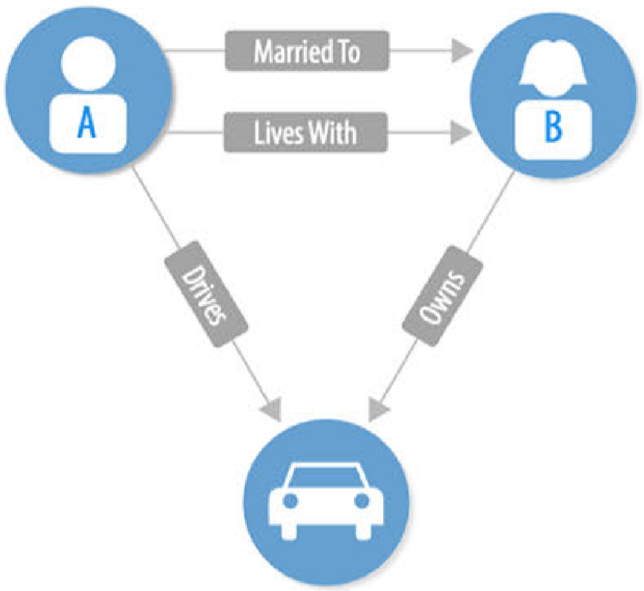

Figure 2.1: Example of a basic graph database

This graph shows how two people, A and B, are related to each other, and how the are both related to a car.

The first proposal of a graph database as a method of describing data was in John Sowa's 1979 paper Conceptual Graphs for a Data Base Interface [Sowa, 1976].This paper considers the concept of a graph database as a means of describing data rather than storing it. The graph was to be an intermediary between the human user and the computer. The concept was conceived as a method of querying a computer system. The system was to convert a question from human language into a conceptual graph. The system could search for other graphs within the data base that are relevant to the original question.

Since then, many graph database systems have been developed commercially. The first commercial graph database was Allegro graph in 2004, a graph database initially designed to store RDF triples [Buerli and Obispo, 2012].

A very notable example of the use of a graph database is with Amazon Neptune, first discussed in 2011 in Chris Brunch et al's paper "Neptune: a domain specific language for deploying hpc software on cloud platforms" [Bunch et al., 2011]. The system was released in 2018. It is an extension of Amazon web services. Amazon claim that this system is a quicker and more efficient way to run web applications that work with incredibly large datasets[GroupLens, 2018, ?]. Although this information comes from Amazon's documentation with little evidence provided. Due to how new the system is, there has been little time to create full reports on its capabilities. A lot of information about Neptune is confidential as this product is a large investment for Amazon.

Many popular open source graph database systems have also been released. One popular example is Neo4j, a graph database query program based on Java, .NET, JavaScript, Python, Ruby that was released in 2007 [Kemper, 2015]. This program stores can be used to store data into graph format.

There have also been querying languages developed to increase the utility of graph databases as methods of storing data. A commonly used language is Cypher that has made it possible for communities to develop graph database systems easily [Holzschuher and Peinl, 2013].

2.2.3 Graph Databases for Recommender Systems

It wasn't until Zan Huang et al's 2002 paper A Graph-based Recommender System for Digital Library until graphs were applied to recommender systems [Huang et al., 2002]. This paper investigated graph representation as a potential method of recommendation for digital libraries. At this time the technology wasn't available for a true NoSQL graph database, instead this paper created a graph network between two SQL databases. The first layer being book contents and the second layer being customer demographics. As with standard graph databases, data points were treated as nodes and relations as edges. However, the connection of nodes could only occur between two layers. Even with this limitation, when evaluated with human subjects a recall of 18.3% and a precision of 38.1% was obtained. Although, the subject evaluation if this research was incredibly limited. The subjects were only two MSci students given six lists. This is not only a very limited number of subjects but also a limited range of subjects.

It wasn't until the mid-late 2000's when true commercial graph databases were established with packages such as neo4j in 2010. The effectiveness of a graph database system as opposed to a standard relational database was shown in Chad Vicknair et al's 2010 paper A Comparison of a Graph Database and a Relational Database which compared neo4j to the common relation database MySQL [Vicknair et al., 2010]. The paper investigated two types of query, structural and data queries. Structural queries and data queries. The data queries were, count the number of nodes whose payload data is equal to some value; Count the number of nodes whose payload data is less than some value; count the number of nodes whose payload data contains some search string (length ranges from 4 to 8). The result from this was that neo4j performed better than MySQL, resolving the queries in quicker time. However, this does not relate specifically to recommender systems.

In the early 2010's, multiple graph database query languages were developed. In 2013, Holzschuher et al. compared the graph database languages Cypher, Gremlin and Neo4j in their 2003 paper Performance of Graph Query Languages [Holzschuher and Peinl, 2013]. This comparison was based on a database consisting of As sources of data we used lists of common first and last names, street names and geographical data from geonames.org7. This was supplemented with randomly generated group names, interests, job titles and organization types, activities and messages. The test data contained 2011 people; 26,982 messages; 25,365 activities; 2000 addresses; 200 groups and 100 organizations. This is a considerable amount of data that should prove as a good test of a graph based language. Particularly as this project will be dealing with large amounts of data. The resulting processing of the relationships in the data showed that the database languages outperformed the comparative SQL language, MySQL, by an order of magnitude, and that Neo4j was the best performing language.

With respect to recommender systems, Franųis Fouss et al developed a graph based collaborative recommendation system in their 2006 paper An Experimental Investigation of Graph Kernels on a Collaborative Recommendation Task [Fouss et al., 2006]. This paper describes the creation of a graph database based on the MovieLens database. This database consisted of people, movie and movie category . This graph's nodes were defined by people in the database and the edges were the relations in the database. Namely has_watched, between people and movie, and belongs_to, between movie and movie_category. This should cause clustering of nodes within the graph. People within a cluster will be recommended films that were enjoyed by others in the cluster. Clusters were determined by calculating the Euclidean distance between nodes. The result of this paper showed " *that three similarity measures provide good and stable performance*".

Past this there still is no research in using a movie database to construct a graph with movies/TV shows as nodes that are connected with edges that are defined by properties of the movies themselves. A clear benefit of such a graph is that as the graph database recommender is not based on the consumers information, it is not affected by the cold start problem.

2.3 Clustering in Graph Theory

While there is no single definition of a cluster in a graph dataset [Duda and Hart, 2001], S.E. Schaeffer defines a cluster as "data such that the elements assigned to a particular cluster are similar or connected in some pre-defined sense" [Schaeffer, 2007]. Bar-Ilan et al. define a set of requirements that a collection of data must meet to be accepted as a cluster defined by paths in the graph [Bar-Ilan and Peleg, 1991]. In graph theory a path is a sequence of edges starting a node N_0 and ending at node N_k. Bar-Ilan's definition is as follows [Bar-Ilan and Peleg, 1991]:

- Each cluster should be intuitively connected.
- There should be at least one, preferably several, paths connecting each pair of vertices with the cluster.
- Paths should be connected internal to the cluster.
- Two nodes in the cluster must not only by a path that passes through them, but also by a path that only visits nodes on the cluster.

J.M. Kleinberg defines a cluster in terms of its density; the ratio of present edges to the maximum number of possible edges. Kleinber considers a "good" cluster to be one where the subgraph that makes up the cluster is dense but there are relatively few connections from the nodes in the cluster to the rest of the graph. This definition of this graph, however, relies on rather vague terminology [Kleinberg, 2002]. R Kannan et al. considser a good cluster to be a 'maximal clique', a subgraph into which a node could be added without losing the clique property. A clique being an undirected graph such that every two distinct vertices in the clique are adjacent [Kannan et al., 2004]. This definition is only applicable to undirected graphs.

The issue with these theoretical definitions of clusters is that they are inapplicable to real world problems due to being so high level. There are, however, algorithmic ways to evaluate a cluster. The five common approaches to detecting clusters, as discussed in M. Needham and A.E. Hodler's book [Mark Needham, 2019], are triangle count and density coefficient; strongly connected components; label propagation and Louvain modularity. They define the triangle count measurement how many nodes form triangles and the degree to which nodes tend to cluster together; the strongly connected components as an algorithm that finds groups where each node is reachable from every other node in that same group following the direction of relationship; label propagation as an algorithm that infers clusters by spreading labels based on neighborhood majorities, and Louvain modularity as an algorithm that maximizes the presumed accuracy of groupings by comparing relationship weights and densities to a defined estimate or average. The book then suggests that the strongly connected components algorithm is ideal for recommender systems. However,the algorithm definitions are extremely high level and there is no experimental evidence to back up this claim.

Triangle Count and Density Coefficient

T. Schankand D. Wagner performed a thorough analysis of triangle count and density coefficient in their 2005 paper Finding, Counting and Listing all Triangles in Large Graphs, An Experimental

Study [Schank and Wagner, 2005]. They defined a triangle as a three node subgraph and investigated several methods of counting triangles. The success of the triangle counting algorithms were measured by the time executed and the number of triangle operations. The definition of triangle operations varied between algorithms but in essence was the asymptotic running time.

The algorithms were run on an undirected graph network, road network Germany, and a directed graph, IMDB movie database. For the bith networks, the node iteration method was the quickest to execute, it was the most asymptotically intensive. This method relied on iterating across each node and counting surrounding edges to count each triangle. The forward-hashed algorithm was the worst performing. This algorithm worked by iterating along dynamic data instead.

Strongly Connected Components

The strongly connected component algorithm was one of the first cluter detection algorithms. It was invented by Robert Tarjan in 1970[Tarjan, 1972]. The algorithm works by finding sets of nodes where all nodes can be reached by all other nodes, in both directions but not necessarily directly [Fleischer et al., 2000]. Esko Nuutila and Eljas Soisalon-Soininen improved on this algorithm in 1990, making it able to handle sparse graphs and trivial components[Nuutila and Soisalon-Soininen, 1994].

Louvain Modularity

Louvain Modularity is a graph clustering algorithm that was developed in 2008 by Vincent D Bondel et al based on the development of modulatrity in a system [Blondel et al., 2008]. Modularity is a method of quantifying the strength a cluster in a graph system. Modularity is value between -1 and 1 that measures the density of edges inside a graph cluster compared to the density of edges outside that cluster. The mathematical definition of modularity is shown below in Equation 2.1.

$$Q = \frac{1}{2m}\sum_{ij}\left[A_{ij} - \frac{k_i k_j}{2m}\right]\delta(c_i, c_j) \tag{2.1}$$

Where A_{ij} is the weight between nodes i and j, k_i and k_j are the sum of the weights of the nodes attached to i and j respectively. $2m$ is the sum of all of the weights of edges in the graph, c_i and c_j are communities of the node and δ is a simple delta function. [Newman, 2006].

Modularity has been an important component in many areas that can be represent in a graph network. This includes , the World Wide Web, metabolic networks, social patterns and so a clustering algorithm based on modularity was an important milestone to achieve [Clauset et al., 2004].

The Louvain modularity algorithm is a two step iterative process, outlines in Equation 2.2 below.

$$\Delta Q = \left[\frac{\sum_{in} + 2k_{i,in}}{2m} - \frac{\sum_{tot + k_i^2}}{2m}\right] - \left[\frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \tag{2.2}$$

Where $\sum_{in}$ is sum of all the weights of the links inside the community i s moving into $\sum_{tot}$ is the sum of all the weights of the links to nodes in the community i.

In terms of recommendation systems, the Louvain modularity has been used to build movie recommender systems from social data. Deepika Lalwani et al published a paper in 2015 using Louvain modularity to find clusters within a social media graph database find recommendations for movies [Lalwani et al., 2015]. In another study in 2013 by Maryam Fatemi et al used the IMDb data to build a graph data base that described common movies between people [Fatemi and Tokarchuk, 2013]. While both experiments resulted in accurate recommendations from within the cluster, both are reliant on user information. There has yet to be an investigation into using Louvain Modularity on a graph database that soley contains information about the movies.

2.3.1 Label Propagation

Label propagation is a system that works by labelling and relabeling nodes in a graph until satisfactory clusters of labels have been created. This technique has been used in the medical sector. Rolf A.Heckemann et al used label propagation on MRI scans of brains in order to achieve greater accuracy when combining results in their Automatic anatomical brain MRI segmentation combining label propagation and decision fusion [Heckemann et al., 2006] with positive results.

Within mathematics, concepts were developed to solve complex problems inhomogeneous biharmonic equation with dirichlet boundary conditions. This was discussed in Jingdong Wang et al's 2008 paper Linear neighborhood propagation and its applications[Wang et al., 2008]. However the actual application of this algorithm to complex mathematical problems has yet to be fully realised.

ZH Wu et al proved the use of label propagation in social networks for finding overlapping communities in their 2012 paper "Balanced Multi-Label Propagation for Overlapping Community Detection in Social Networks" [Wu et al., 2012]. Label propagation has also been shown to be useful in graph crompression, as was shown in 2011 by Paolo Boldi et al in thier paper Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks.

However, in terms of recommendation systems, Hou Qiang et al proved the potential of its use by combining it with collaborative filtering. This was discussed in their 2012 paper "A method of personalized recommendation based on multi-label propagation for overlapping community detection" [Shao et al., 2009]. This paper investigated the Movie Lens dataset and found scarcely improved results to the baseline collaborative filtering comaprison.

Within music recommendation, Bo Shao et al used label propagation to develop recommendation system based on a graph database describing the user access patterns of the NewWisdom music network, as well as the acoustic features of the songs the user has listened to. This was written up in their 2009 paper "Music Recommendation Based on Acoustic Features and User Access Patterns " [Shao et al., 2009].

There is still yet to have been research done into recommender systems using clustering on a graph movie database that contains only information about the movie.

Chapter 3

Methodology

The main goals of this project were to build a graph database describing movies; analyse graph clustering methods when applied to said database and investigate the clustering techniques as a method of recommending movies to people based on their movie preferences. As such the methodology was split into three sections, building and investigating different graph databases; performing clustering methods on the databases, and using the clusters found to make a recommendation system.

Different graph databases where to be tested in order to determine the optimal graph. While it is possible to see if the graphs are constructed with clear nodes and relationships, the functionality of the graphs can only be investigated by analysing the clustering algorithm results. Likewise the optimal clustering method found could only be found by comparing results of the recommendation system.

3.1 Building the Graph Database

Two approaches to building the graph database were investigated. The first involved creating nodes for each movie and each property. That is, there is a node labeled "MOVIE" for each movie, and a node labeled "ACTOR", "GENRE",... for each actor, genre etc in the database. The edges between these nodes will be directed relationships from a property node to a movie node. The basis of this graph is that property nodes will cluster with movie nodes, the movies can then be identified from these clusters when recommending.

The second approach was to build a graph that contained only movie nodes. The movie nodes can then be connected bidirectionally with edges that represent shared properties. For example, if a movie has any of the same actors there will be a bidirectional relationship labeled 'Shared_actor'. The same was done for the other properties considered. With graphs designed in this method the clusters were expected to form based on relationships between each movie, rather being based on the property nodes shared between movies. As the clusters would be entirely movies, the entire cluster could be used for recommendation

3.1.1 The data used

The data used to construct the graph database was the TMDb 5000 Movie database. TMDb (The Movie Database) is an online, community built television and movie database. The website has been collecting data since 2008 from a community of 1,428,725 [(tm, 2019]. The site has users collecting information about 474,652 movies, pairing that with its public access makes this an excellent source for analysing movie properties. While publicly contributed and moderated data is not generally as reliable as published data, the scope of the website suggests it is usable for analysis. the TMDb 5000 Movie Database is a sample of 4083 movies with a selection of the information about the movies.

The data came in the form of two .csv files; tmdb_5000_movies.csv and tmdb_5000_credits.csv. The former contains detailed information about each movie, including cast and crew members, while the latter gives a deeper description of the cast and crew.

tmdb_5000_movies.csv

The tmdb_5000_movies data' columns detailed each movie's Budget (USD), Genres, Homepage,ID (a number unique to each movie), Keywords, Original Language, Original Title, Overview, Popularity, Production Companies, Production Countries, Release Date (dd/mm/yyyy), revenue (USD), runtime (minutes), Spoken Languages, Status (released/ not released), Tagline, Title, Vote Average, Vote Count.

The data was taken from Kaggle [?], an online community of data scientists owned by Google.

Continuous data is inappropriate for a graph database as they cannot be used to create distinct, usable nodes. Hence, the Budget, Vote Count, Voting Average, Runtime, Revenue, Release Date and Popularity were discarded. Overviews, Homepage and Taglines cannot create usable nodes as they are distinct to each movie and will not add anything of worth to the graph's architecture. The release status of a film would not affect anything that can relate to a person's preference and so that was ignored. The current title was used instead of the original so as to keep consistency with other movie data.

This csv had a single row for each movie. Genre, Production Companies, Production Countries and Spoken Languages have multiple attributes and so they were represented by JSON strings. A sample of the data kept can be seen in Table 3.1

genres	id	original language	title	...
[{"id": 28, "name": "Action" }, {"id": 12, "name": "Adventure"}...]	19995	en	Avatar	... ...

Table 3.1: Sample of the columns kept from the tmdb_5000_movies data

tmdb_5000_credits.csv

The tmdb_5000_credits csv had a similar layout though with just four columns; Movie Id, Title, Cast, Crew. Movie Id and Title correspond to their equivalent in tmdb_5000_movies while Cast and Crew where JSON strings describing the cast or crew member. The Cast JSON string contained an cast ID identifying the cast member within the cast list; the character name; a credit ID identifying them in the credits; their gender (numeric: 1=female, 0=male); their name and their credit order (order of 0 suggests the main actor). An example string is shown below:

[
{ "cast id": 242,
"character": "Jake Sully",
"credit id": "5602a8a7c3a3685532001c9a",
"gender": 0,
"id": 65731,
"name": "Sam Worthington", "order": 0}...
]

The Crew string contains the credit ID, the Department the crew member worked in, the Gender (0/1), an unique ID, their job and their name. A example string is shown below:

[
{"credit_id": "52fe48009251416c750aca23",
"department": "Editing",
"gender": 0,
"id": 1721,
"job": "Editor",
"name": "Stephen E. Rivkin"}...
]

In both of these strings the only useful information is the staff's name and their ID to distinguish them.

3.1.2 The programs used

Two programs were used for the graph creation; Python and Neo4j. Python was used to prepare the TMDb 5000 data into a format that could be used to turn the wanted data into a graph database. Neo4j was the software used to create the graph once the data was in a usable format.

Python and pandas

Within Python,the library pandas was used to manage the csv's. This library can import csv's and store them as a dataframe object. This allows for complicated manipulation of data treated as a standard table. The library is capable of reading in and storing JSON strings as dataframe objects. This allowed the properties described as JSON strings to be accessed. The library can also be used to join tables, making it useful for dealing with data that is split across multiple files [McKinney, 2012]. As was the case here.

Neo4j

Neo4j is the world's most widely used graph database management system. The system works of graph theory, creating nodes that contain the data connected by edges that represent some data connection. Neo4j allows for the free construction of graphs, including the creation of new nodes and edges. When a node is created it must be assigned a label. This establishes a distinction between nodes and the ability to query the graph. A node with a label,n, is defines as (n). Individual nodes can be distinguished from each other by assigning properties to them when they are created. This allows for the querying of specific nodes in the graph. The number of properties per node is limitless. Properties may be integer, float, string, Boolean or an array of any of the above. The syntax of creating a node labelled "MOVIE", with the properties "title: The Matrix" and "id: 423" is show below:

CREATE (m:MOVIE {title: "The Matrix", id: 423})

In Neo4j, the edges in the graph are called relationships. These relationships can be non-directional, mono-directional. Bi-directional relationships are formed by two mono-directional relationships in reverse from each other. As with nodes, relationships must be assigned a label. A relationship r between two nodes n and m is represented in Neo4j as:

$(n) - [r] - > (m)$

The arrow is optional and indicates the direction of the relationship. Relationships can be assigned weights in a similar way as node properties are assigned. This is used to build weighted graphs that place importance on specific relationships. When weights are assigned they must be labeled for proper querying. The creation of a relationship called "acted_in" between a node (n) and a node (m), with a with a weight wt=0.5 us shown below:

MATCH (n), (m)
CREATE $(n) - [: acted\_in : \{wt : 0.5\}] - (m)$

MATCH is an important command in Neo4 as that is how nodes are accessed.

Querying in Neo4j

A benefit of Neo4j is its ability to query databases. The program uses a querying language called Cypher to investigate properties of the graph. It can be used to search for specific data, construct specific sub graphs and apply operations such as the count or the mean. It also allows for the manipulation of node properties through numeric and string operations.

Querying in Cypher is oriented around two commands; MATCH and RETURN. MATCH whether a node or relationship in the graph required for the query, RETURN gives the final result of the query.

Between this, specific nodes can be accessed and manipulated by one of their properties. For example; finding a MOVIE node with the title "The Matrix" can be done as follows:

MATCH (m:MOVIEtitle: "The Matrix") RETURN m

Within this, Boolean, string and numeric operations can be performed, in the same manner as SQL. For more advanced statistical and mathematical operation, packages in Neo4j are needed.

Algorithms in Neo4j

There are packages available to allow for more complicated analysis. The two packages used in this project were APOC (Awesome Procedures on Cypher)and Graph Algorithms. APOC contains many useful functions, and is particularly useful for performing various statistics equations. Graph Algorithms is used to apply algorithms specific to graph theory. These include centrality, path finding, link prediction and community detection. Within the community detection graphs are the strongly connected clusters, connected clusters, Lovain modularity and label propagation discussed in the literature review. It is these inbuilt algorithms that where used to investigate communities in the TMDb 5000 movie database.

3.1.3 Multi-label graph

The first graph investigated consisted of nodes with multiple labels. Each movie was labeled with its title and its id. Every property of that movie was also created as a node with its respective label. For example GENRE and ACTOR nodes.

Properties Investigated

[Why the properties that were investigated were chosen] The director was the only crew investigate as they are the main influence on the movie.

Preparing the Data

In order to constrict a graph from imported data in Neo4j, it must come in one csv. When the csv is read in the table's rows can be iterated through and the values used in node, relationship and property construction. As the data was in the form of two csv's, they needed to be combined into one in such a way that the graph wanted can be built through the iteration.

For the multi-label graph, the table needed to have a column for each movie and property investigated. With the data in this format, each column can be considered a node label; each unique entry a node, and relations can be constructed from each row.

For this to be the case there had to be multiple entries for movie to match each of the properties. In order to do this, a pandas dataframe for each of the csv's was built. The rows of the csvs were iterated along, allowing the reading of the JSON strings. The individual properties from each string was built into a new dataframe, a column for the value and a column for the movie id. The dataframes were then outer joined on the movie id.

The final columns were Movie ID, Title, Original Language, Release Date, Director Name, Director Id, Genre, Actor Name, Actor ID.

Different tables were made containing different numbers of actors so that the impact of the number of actors can be investigated. Seven tables were created containing one to seven actors per movie.

Creating the Graphs

The graph was then built using Neo4j. From the previous table four types on node were to be created; (ACTOR), (MOVIE), (DIRECTOR), (GENRE). These contained the respective name and ID. The relationships built were:

- (:ACTOR)-[:ACTED_IN]->(:MOVIE)
- (:DIRECTOR)-[:DIRECTED]->(:MOVIE)
- (:MOVIE)-[IS_GENRE]->(:GENRE)

The process for creating the graph with this software can be broken down into several steps. Firstly, constraints were made on the creation of nodes such that ACTOR nodes have a unique actor ID, MOVIE nodes have a unique movie ID, DIRECTOR nodes have a unique director ID and that genres name were unique. The constraints were made on IDs rather than names as it distinguishes shared names.

Secondly, the table rows where run through with a new ACTOR node for each actor ID, this node was given the name and ID information. This node was then merged into the graph. The same thing was run for each node type.

To establish relationships, the the table's rows were run again. This time the Neo4j function MATCH was used to find the node that matched the actor ID and the node that matched the movie ID and creates the relationship ACTED_IN between them. This was repeated for DIRECTED and IS_GENRE.

Multiple graphs were made using this method for all the different number of actor tables and a version with and a version without the genres. This allowed for testing on the parameters effect.

A schematic of a multi-label graph representing movie data can be seen below in Figure 3.1.

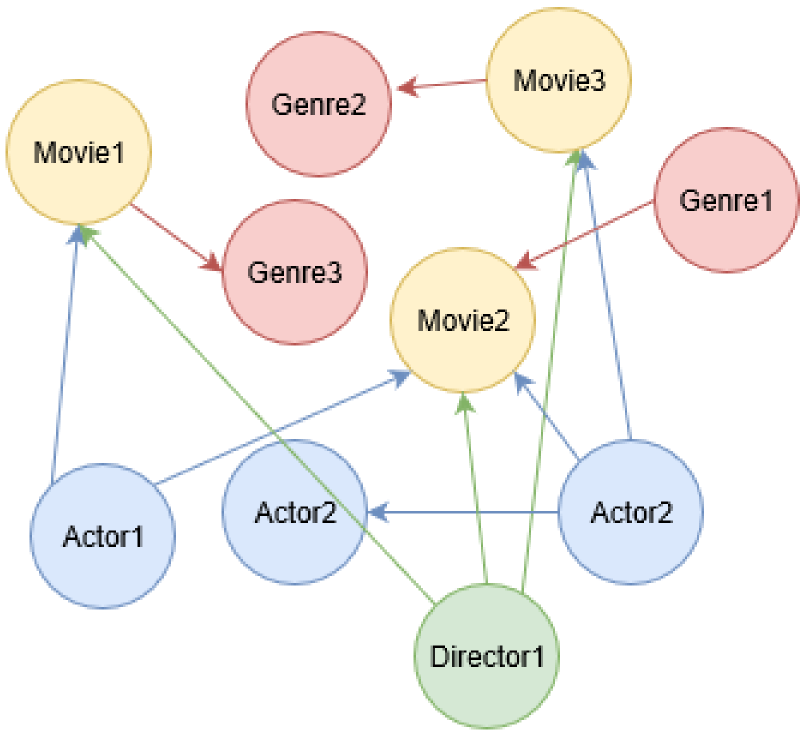

Figure 3.1: Schematic of a multi-label graph representing movie data

The image shows the graph's different labels of node, red being GENRE; blue being ACTOR; green being DIRECTOR and yellow being MOVIE. The green arrows represent the :DIRECTED relationship; the blue represented the :ACTED_IN relationship and the red arrows represented the :HAS_GENRE relationship.

3.1.4 One node Label

The second graph, also built in Neo4j, consisted only of nodes labeled (MOVIE). The details of each movie was stored as properties to its unique node.

Properties investigated

This method allowed the inclusion of more properties without the fear of properties filling clusters at the expense of movies. The properties included, actors, directors, genres, keywords, production companies, original language production country and languages spoken. For the actor properties, the first five listed actors were selected. This was done because smaller actors tend to have many minor parts in films, regardless of style of film. It was assumed that when it comes to film preferences, only the top listed actors are often noticed. All the directors and genres were used a this was rather specific. Keywords highlight distinctive properties and so are more useful. Production companies tend to have unique styles that can be preferred, such as Disney.

Preparing the Data

The one label graph was built by combining the two original csv's into one whole csv's. All of the JSON data was converted into an array containing just the id values. Only the id values were used as that ensures the values being dealt with are unique. The final csv contained one row per movie, with metrics that have multiple values being represented as an array of their id's. In Neo4j the csv was iterated and a node was created for each movie. As the nodes were created, each was given a property for each metric in the csv. As the properties can be stings, numeric, Boolean or arrays it was possible to describe all the genres and actors this was.

Graph creation

When the csv was imported into Neo4j; the nodes where created by a simple CREATE command, using the row's values for the properties of the node.

When a csv is read into Neo4j, every entry is read in as a string. Hence, it is not possible to simply read in an array to be a node property. To counter this, each array value in the csv was converted into a comma separated string. When the value is being read in the Neo4j command 'split' was used to separate the values into neo4j array. This array was then assigned as the property.

Relationships were created between nodes by using the querying language, Cypher. Cypher was used to match nodes that have the same property value , this was done for non-array properties such as the original language property. For relationships between array properties, matches were created based on the intersection of the properties being non-empty. The final graph was a node for each movie and a relationship between those with similar properties

The query can be seen below:

```
MATCH (n:MOVIE), (n2:MOVIE)
WITH n, n2, apoc.coll.intersection(n.actor, n2.actor) AS common
WHERE NOT common = []
AND NOT n = n2
CREATE (n)-[:shared_actor]->(n2)
```

The union was found using the apoc addition to neo4j.

A schematic of the graph is shown below in Figure 3.2.

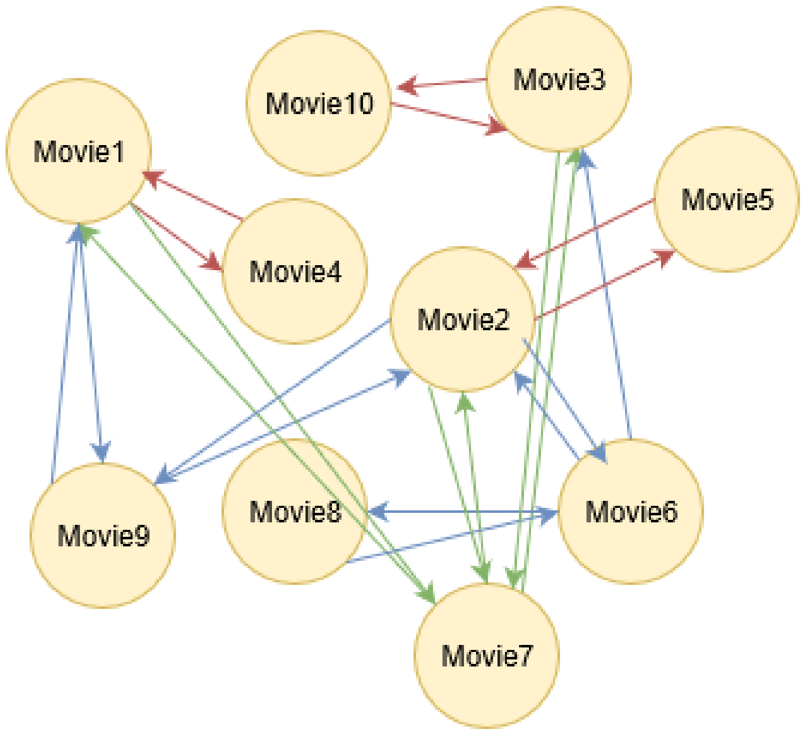

Figure 3.2: Schematic of a single-label graph representing movie data

This image shows different movie nodes and how they connect with each other. Red arrows represent shared genre; green arrows represent shared movie and blue arrows represent shared director.

Adding weights to the graph

Another property of graph databases that can impact the behaviour is weighted relationships. Adding weighs to a graph can help to identify the importance of a relationship. Doing this can can counter a graph's density becoming troublesome. Another reason to investigate weights is that establishing importance between relationships can further identify communities. Rather than all relationships being treated equally, a focus is on nodes with stronger relationships can improve the results of clustering algorithms. This is particularly useful for label propagation and Louvain modularity.

The weightings for this graph where created by finding the ratio of the number of values in the source node property with the number of values in the source and target nodes' intersection. This returns a value between zero and one for all relationships. This gives a way to distinguish importance of relationships, while giving all relationship types a comparable range. It is important to have a similar range of values for each relationship type so that no one relationship type is given more priority.

This was specifically done in Neo4j by altering the relationship creation Cypher query in the previous code. This is shown below:

```
MATCH (n:MOVIE), (n2:MOVIE)
WITH n, n2, apoc.coll.intersection(n.actor, n2.actor) AS common
WHERE NOT common = [ ]
AND NOT n = n2
CREATE (n)-[:shared_actor{weight: length(n.actor)/length(n.common)]->(n2)}->(n2)
```

Although the maximum memory was allocated to the program, the task of assigning relationships based on production country was too great.

Fixing the weights

One issue that can arise from weighting the relationships in this was is that the number of distinct keywords far outnumber the number of genres per movie. this combined with many commonly repeated keywords meant that the intersection was much greater, leaving a much smaller weight for the keywords. It is also difficult to establish importance of individual properties within the context of the data.

There are many values within these nodes that are very often repeated. Such as the keyword actor occurring in X number of films. This is not useful when it comes to recommendation as it doesn't focus on the distinct properties of people's preferred films. Putting more emphasis on less frequent words that are shared is far more useful.

To counter this, the frequency of each element was found within the data. The inverse of each element's frequency was assigned as that elements weight. Assigning a weight to each element of each property gives a better indication of the truly important values. In terms of the keywords, the less commonly used the word, the higher the element's weight.

Making the final weight of the relationship the sum of the element weights in the property intersection gives a better indication of the effect the different property elements have on the graph.

With the relationships being developed this way, there was no need to limit the number of actors. As this weighting approach focused more on unique values that are shared compared to common ones.

In order to do this practically in neo4j, the properties of the node had to contain information both about the property and their frequency. There also had to be a way of relating the two. Unfortunately in Neo4j, properties can only be basic objects, meaning that an array of arrays could not be used to compare indexes. There was also no way of relating each property to a separate table with a pottery id as a key and the frequency as the values as only information about the graph could be stored.

This was resolved by altering the arrays in the property columns of the tables in this section. The arrays that contained the properties were changed from ["id1", "id2", "id3" ... "idn"] to ["id1|frequency1", "id2|frequency2", "id3|frequency3" ... "idn|frequency"].

When building the relationships between nodes, after the intersection was found, the values within the array of properties could then be split on "|" and the index [1] (containing the frequency) for each value could be retrieved and summed. An array can have each element in an array worked upon through a process called list comprehension. This process constructs a new list from a defined function.

In this case the be list created was a list of the frequencies of the intersection array. The code for this can be seen below:

```
WITH [x IN common | toFloat(split(x,'|')[1])] AS result
```

Here each element in the intersection array is split, converted to a float and then the second element (containing the frequency) is selected. This returns the result as a variable "result". This array could then be summed using the apoc function apoc.coll.sum() and set as the weight for the relationship.

However, in Cypher, when using list comprehension, references to the initially matched nodes (n) and (n2) are lost. Hence, when trying to combine the functions in a query such as below:

```
MATCH (n:MOVIE), (n2:MOVIE)
WITH n, n2, apoc.coll.intersection(n.keywords, n2.keywords) AS common
WHERE NOT common = []
AND NOT n = n2
WITH [x IN common | toFloat(split(x,'|')[1])] AS result
CREATE (n)-[:shared_actor{wt: apoc.coll.sum(result)}]->(n2)
```

The result is the creation of two new blank nodes, connected by the "shared_actor" relationship for every combination of (n) and (n2).

To overcome this, the query was to produce the intersection was altered to return n.title, n2.title, common_keyword, common_actor... A "common" for each intersection between (n) and (n2). This results in a table with all combinations of (n) and (n2) that are allowed by the query. This table was then saved as a csv.

A blank instance of Neo4j was then created. In this instance the csv was loaded row by row as usual. This time the row's values for common was run through the list comprehension and summed.The sum was then returned and saved to a csv. This csv contained the weights for every relationship type, for each (n), (n2) combination in the same order as the previous csv. These csv's were combined using pandas.

After which, in another blank instance of Neo4j, a constraint to make each nodes title unique was placed. After this the final csv was read in row by row, merging the nodes and relationships.

Combining the relationships

Another step in confronting the density of the graph was combining all relationships between nodes into one single relationship. Reducing the number of relationships helps with the graph density while simplifying the relationship labels may improve the performance of the clustering algorithms.

As well as the reducing the density of the graph, combining the relationships can also make it clearer how each node is related. One label of relationship may also improve the quality of the community detection.

It is possible to build a query that uses aggregation functions to determine a combined weight of all of the relationships between two nodes. Here for each node (n), (n2), the weights of the existing relationships were aggregated, this was the used as the weight for a new relationship between the nodes. The other relationships between the nodes were the deleted.

There are three main aggregation methods that could be used to combine the relationships in a meaningful manner when focusing on numerical weights. They are Count, Average and Sum. Count returns the number of relationships between the nodes. This is a good option for creating a weight from unweighted relationships. Rather than the algorithm interpreting multiple relationships, the count can be used to give a clear relationship that clearly highlights the strength of the relationship.

However, this is not beneficial to aggregating relationships with weights as it ignores the importance of the existing relationships, giving each of them equal value. There are three ways of averaging weights; mean, median and mode. The median was not useful for this instance as float values are being aggregated, meaning there will be very few common weights. Using the median would ignore the impact of the strongest relationships.

The mean could be a beneficial way of aggregating, however it ignores the number of relationships between the two nodes. Two nodes with relationship weights [0.5, 0.3, 0.4] have the same mean as a two nodes with relationship weights [0.5, 0.3]. While the values between the two are similar, the extra relationship of weight 0.4 clearer suggests a stronger relationship. Hence, the sum of the weights was chosen as the aggregation function to use.

While Neo4j has inbuilt aggregation function, these functions aggregated the weights of all the relationships in the graph. It is not possible to use them to aggregate only relationships in specific nodes. To do this required matching and assigning variable to all of the values. Basic matching will not work here as this query discards null values. This results in only connecting weights for relationships that exist between all nodes.

```
OPTIONAL match (n)-[s:shared_keywords]->(m)
OPTIONAL match (n)-[r:shared_production_country]->(m)
OPTIONAL match (n)-[t:shared_director]->(m)
OPTIONAL match (n)-[u:shared_actor]->(m)
```

This collects all of the stated relationships between n and m as a set of objects. This is saved for each (n), (m) combination. A list containing the title for each (n) and (m) were combined into a list containing another list with all the weights, [n.title, m.title, [r.wt ,s.wt, t.wt, u.wt]]. The last element was summed using the apoc.coll.sum() function. However, as optional matching was used, this list contained null values that cause an error when run. To counter this, the Neo4j filter function was used. This function tested for each of the elements where IS NULL returns true. This value was then replace with zero. The full function i shown below:

```
WITH [n.title, m.title, apoc.coll.sum(FILTER(x IN [s.wt, r.wt, u.wt, t.wt]
WHERE x IS NOT NULL ))] AS combination.
```

The list of n.title, m.title and their collective relationship weight was assigned to a variable, combination, in order for it to be accessed when creating the new relationship.

In the same query, another MATCH clause was used. The specific nodes were identified by indexing the combination list. The new relationship was created with the weight found by indexing combination. This part of the query is shown below:

```
MATCH (n:MOVIE title: combination[0])-[w]->(m:MOVIE title: combination[1])
MERGE (n)-[:combinewt: combination[2]]->(m)
```

Here w is just a place holder for the relationship. MERGE was used instead of create to avoid duplicating relationships. After this the original relationships were deleted, leaving only the combined weight relationships.

3.2 Clustering

Investigating clustering was the second main objective of the project. The goal here was to determine if a movie database graph could perform successful clustering with the goal of recommending movies. There were two key points of analysis; finding the optimal graph and pairing it with its optimal algorithm. The quality of the graph and the quality of the algorithm were tested simultaneously as the algorithm was applied. If clusters were successfully detected then that would accomplish proving that it is possible to construct a graph with the intention of clustering and it is possible to use algorithms to detect communities. The clusters built here could then be taken on to building a recommendation system

3.2.1 Cluster comparison metrics

As the numeric methods of determining connections are not applicable, the metrics on which the algorithms where judged were the size of each cluster and the number of nodes in each cluster. This is the best metric for the use of recommendation systems as each movie must be within a cluster, and each cluster must contain a suitable number of clusters in order to build recommendations.

An ideal recommendation system would provide a method of recommending each film. This would require minimising the number of single node cluster and clusters containing just one film node. In contrast, if the graph is clustered in way that the majority of movies formed one large cluster, the method would not provide a narrow enough recommendation. It would be preferable for clusters to form into many, evenly sized clusters.

These metrics were used to compare the different algorithms and different graph structures when trying to determine the best graph structure and the best algorithm to perform recommendation testing.

3.2.2 Neo4j clustering algorithms

These clustering was done with the Neo4j package Graph Algorithms

Louvain Algorithm

The implementation of the Louvain algorithm in Neo4j is shown below:

```
CALL algo.louvain.stream(label:String, relationship:String,
weightProperty:'weight', defaultValue:1.0)
YIELD nodeId, community
```

Within the algo.louvain.stream clause there are five arguments, label, relationship, weightProperty, default value and concurrency. The first two arguments define which label node and relationship that is wanted analysed. To use all labels and relationships, this is left blank. The weight property and the default values give the opportunity to include weights in the algorithm. weightProperty can be used to set a define a weight from a variable 'weight. If this is not used, the weight for all relationships is "defaultValue" which is set to 1.0

Yield returns results form a Neo4j Graph Algorithm. "nodeId" is the ID number of a node. This can be used to determine the movie and its properties. "community" is the integer community label.

The weights in these systems varied between relationships. To include this into the information the weight had to be specified. This was done by creating a match clause returning the relationship variable in the algorithm's node label argument. The relationship argument is then defined by the another match clause. This time using SOURCE and TARGET. Using the previously matched nodes as the source and target of the statement, the weight of the relationship can be returned as 'weight'. This can then be picked up by the algorithm. This is shown below:

```
CALL algo.louvain.stream(('MATCH (n) RETURN id(n) as id',
'MATCH (n1)-[r:combine]->(n2)
RETURN id(n1) as source,id(n2) as target, r.wt as weight',
graph:'CYPHER', write:true))
```

Defining the variable "graph" makes this a Cypher projection. Meaning that the clustering is built from projecting the subgraphs queried in the label and relationship arguments.

This alteration to the algorithm is the same method to include the weight in all Neo4j 'Graph Algorithm' algorithms

Label Propagation

The Neo4j label propagation algorithm that was used is shown below:

```
CALL algo.labelPropagation.stream(label:String, relationship:String,
iterations:1,
weightProperty:'weight', writeProperty:'partition',
direction:'OUTGOING')
YIELD nodeId, label
```

The algorithm works in the same way as the Louvain algorithm. However, there are three new arguments; "iterations", "writeProperty" and "direction". "iterations" sets the maximum amount of iterations the algorithm will run through. To maintain a small enough runtime a value of 5 was chosen for every run. "writeProperty" is the property the algorithm is written back to. The default 'partition' was used.

As with Louvain, an initial matching was needed to define the weights. This was done in the same way.

Neo4j Connected Components

The Neo4j function for performing the Connected Component is shown below:

```
CALL algo.unionFind.stream(label:String, relationship:String,
weightProperty:'weight', threshold:0.42, defaultValue:1.04)
YIELD nodeId, setId
```

This algorithm works much the same as before. The only difference is the inclusion of "threshold" which is a float defining the threshold of a relationship's weight. Below this the relationship is discarded. "setId" gives a float values defining the cluster, as before

Neo4j Strongly Connected Components

The stringly connected components algorithm is much simpler in comparison, as can be seen in the syntax below:

```
CALL algo.scc.stream(label:String, relationship:String)
YIELD nodeId, partition
```

There is no method of including weights into this algorithm. The only arguments that can be altered are the node labels and relationships investigated. Here all the labels and relationships were used. The "partition" yield is the integer corresponding to the cluster, as with the other algorithms.

3.2.3 Multi-label graph

Choosing the algorithm

Each graph contained a different combination of movie features and so the algorithms would perform differently. However, the one commonality with these graphs is their lack of triangles. The only relationships that can form from this data is mono directional from a non-movie node to a movie node. Hence, there is no possibility of three edges within three nodes. The triangle count and density function rely on the forming of triangles in the database [Schank and Wagner, 2005] making these methods unusable in this context.

In the same way both Strongly Connected Components and Connected Components are unusable as the definition of a cluster in these methods is an area where nodes can be reached from each other. The different types of node will never be directly reachable from each other making those also unsuitable.

Hence, the methods that that were investigated were Label Propagation and Louvain Modularity.

Applying the algorithms

The algorithms where run using the inbuilt Neo4j functions. This allowed it to be directly applied to the graphs that where built. The first graphs investigated consisted of :MOVIE nodes and :ACTOR nodes with the :ACTED_IN relationship. Graphs were investigated between one actor per movie and seven actors per movie. The number of actors was limited due to the computation intensity of creating more relationships and nodes.

Once the results were taken for these, the :GENRE nodes and :HAS_GENRE relationships were added. All twenty genres were added and the full list of genres was given for each film. This was followed by adding the original language.

As both of the algorithms investigated are non-deterministic, each algorithm was five times and the results were averaged.

3.2.4 One-label graph

As the One-label graph contains only bi-directional relationships it was possible to investigate all four of the discussed algorithms, strongly connected components, triangle count, Louvain modularity and Label propagation. However, the Neo4j strongly connected components algorithm ha no weight argument, making it unsuitable when investigating weighted graphs.

As the Louvain modularity and Label propagation are non-deterministic, they had to be run multiple times to account for the different results. These algorithms where run run five times each per graph. Running the algorithms resulted in a table consisting of each movie and its corresponding cluster, represented by an integer. To account for the non-determinism, the the mode of each movies cluster.

The other methods were deterministic, meaning that the result will be the same every time it is run. Hence, there was no need to run the graph multiple times.

3.2.5 Comparing results

The results for both approaches to the problem where compared with each other against the metrics established in Section 3.2.1. The combination of graph and algorithm that produced the best results were run again, this time the tables defining the clusters were produced and collected. If the optimal results were achieved using a deterministic-approach then only one set of results need to be taken. If the optimal method was non-deterministic then five sets of results were taken. This was to ensure reliability moving forward.

Chapter 4

Analysis

This investigation seeks to solve three main goals: building a graph database from movie data; investigating clustering in movie databases and investigating how well these clusters can be used to recommend movies. The analysis was broken into three aspects; investigating how well the graphs were built, investigating how well the clustering algorithms worked, investigating how the clusters worked as a method of recommendation.

As there were two structures of graph investigated, the two were analysed separately. The single labeled graph was looked into how well all the different clustering methods works, and the multi labeled graph investigated how the change in nodes included affected the results of the applicable algorithms.

4.1 Single node graph

4.1.1 Building the graph

The first part of the analysis was investigating how the graph was built. This involved the looking at the number of nodes and relationships. An initial indication of a successful graph is if there is a node created for each movie, containing the required properties. Another thing to initially consider is if the relations are successfully connecting, and if there are a sensible amount of connections. For example, it would not make sense for a node to connect to every other node for any of the properties. It is also important to see if there are sensible number of relationships per node.

The single node graph was built by initially establishing all the nodes in one query, then querying the creation of each relationship. Each relationship was based on the intersection of the node's properties. These where added one query at a time. Table 4.1 below shows the results of the queries as they were input.

Relationship	Number	Number per node	Time taken (ms)
Shared actors	165,496	34.4567	471,996
Shared directors	15,902	3.3108	159,199
Shared keywords	847,246	207.505756	811,661
Shared genres	8,831,308	2162.94587	369139
Shared company	536,698	111.742244	98354

Table 4.1: Number of relations added for each property.

When trying to compare countries it ran out of RAM, even though it was max for the computer at 7GB. This is because almost all where produced in the US. When trying to build recommendation systems, losing information about the movie can cause negative effect. However, in this instance, including the country of origin does not offer much for a recommendation system as such a common result cannot offer distinction between movies. Also building a common relationship between each node can cause confusion in graph algorithms.

On average, an actor will have five top credited roles in a movies ,according to research done by the movie data and education website Stephen Follows[Follows, 2013]. As only the top five listed actors were kept in the array 25 relations per node should be expected.

The number of actor relationships built is 34.4567, 1.37 times expected. This difference is reasonable based on the the assumptions made. A higher value also makes sense because more popular films have smaller selection of actors.

It can be usually expected that there is between one and three directors per movie. A second research piece by Stephen Follows suggests that the average director will make only two movies. Hence, the expected number of shared director relationships per node would be between two and six. The value reached here of 3.3108 fits well into this prediction.

The definition of genres depend on TMDb itself, so it is difficult to compare these numbers to movie statistics. However, TMDb offer a selection of just twenty genre labels. These results imply that 45% of movies share a genres. When comparing the number of movies in the data, 4803, with the number of genres it is clear that ther would be a large cross-over of genres.

TMDb keywords cover a range of properties of the film. This includes general descriptions such as "Action" or more specific terms such as "Jason Vorhees", referring to a character in the Friday the 13^{th} series. As such there should have a large number of shared keywords per node.

80% of the movie market share is owned by five production, according to Box Office Mojo, a website owned by IMDb that tracks box office revenue [Mojo, 2019]. In this graph there are 111.74 shared production company relation ships per node. This is 2.3% of the total number of node. Based of the statistics this may be a rather low number. However, it must be considered that many production companies publish movies under asset company names. For example Disney publishes Star Wars through LucasFilm [Businessweek, 2013]. When considering this, it account for the lower than expected result.

When considering the above, it can be assumed that this method of building a graph database to store movie information works well both for storing the TMDb5000 movie database, as well as representing the general movie industry. Although a more powerful computer could be used to include the country of origin.

4.1.2 Basic weighting

When build the system with the weights each relationship was added separately. The same issue of processing power came about when considering the production country. However, in this instance, the number of genres was also affected. This extra loss of information is not ideal when trying to represent the movie data. However, this came with the added benefit of graph weights. The loss of specific details about the movie results in a better representation of the how strongly the nodes are connected.

The actors directors and keywords were all able to form relationships. The number of relationships is shown in below in Table 4.2.

Relationship	Number	Number per node	Time taken (ms)
Shared actors	165,496	34.4567	10,362,35 0
Shared directors	15,902	3.3108	4,262,39
Shared keywords	847,246	207.505756	17,524,651
Shared company	536,698	111.742244	194,304
Mean	387,399.755	80.657	7126886

Table 4.2: Number of relations added for each property, with ration weights.

The resulting number of relationships was the same as the previous graph. This is expected as the only change desired was the added weights. This shows that the added process didn't affect the alter structure of the graph. However, the running time of this took, on average,11.6n times longer to run. Making it a lot more computationally demanding.

A sample of the graph can be seen below in Figure 4.1

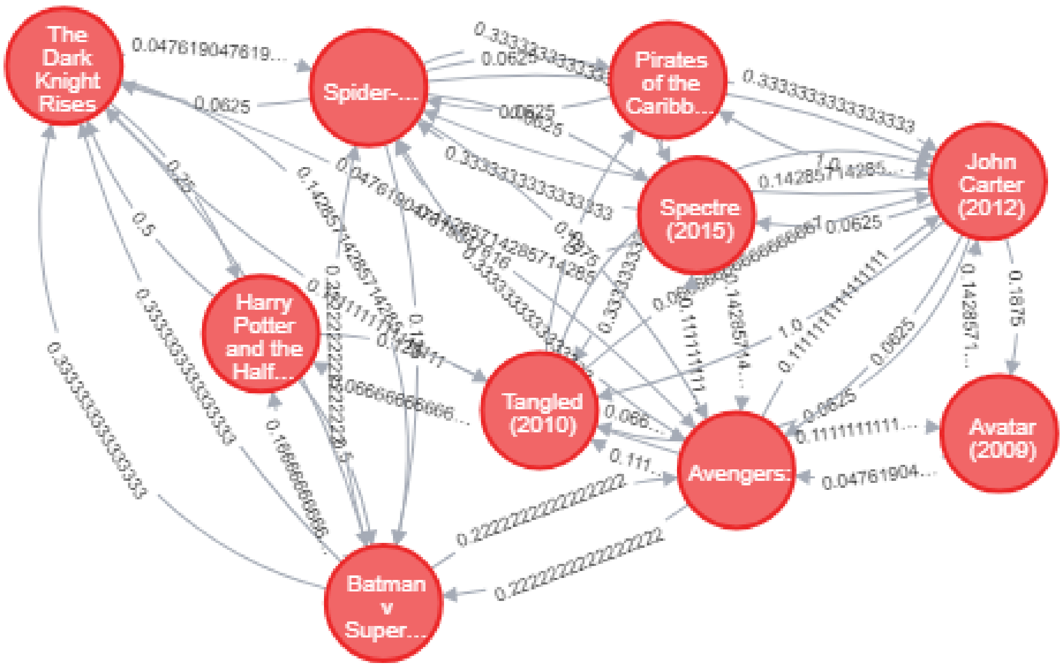

Figure 4.1: Sample of graph with intersection ratio as the weight

This method was successful in building a graph database connecting movie nodes with weighted relationships defined by shared movie properties.

Figure 4.2 below is boxplot comparison of the weights of the different relationships.

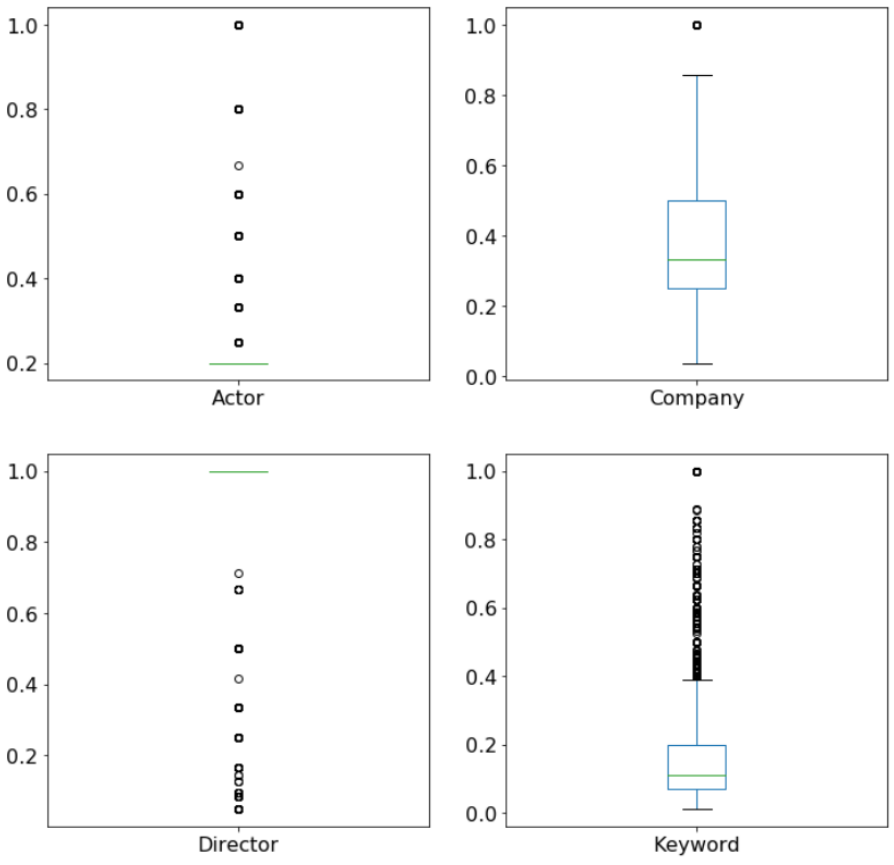

Figure 4.2: Boxplots of weights from the basic weighting

The mean shared actor weighting was 0.2048, meaning that on average there was only one actor in in the intersection, as the number of actors for each movie was set to five. The distribution around this result is very small, as can be seen from Figure 4.2. Although there are movies that share much more. The few stronger results help create a stronger relationship that can distinguish from the basic number of actors.

Looking at Figure 4.2, it can be seen that the mean weight of the shared director relationship is 0.972, with very little range. This is due to there only being around one director per movie, meaning that any intersection will most likely be the same as the union of the properties, giving a ratio of one.

Shared keywords have a very small mean of 0.165, although there are some very strong outliers and more of a distribution than the others. The company shows a much more even distribution with a mean of 0.416.

The issue with the contrast in weights shown is that almost all director relationships will outweigh keyword relationships, potentially rendering the keyword data as redundant. However, as the number of shared director relationship per node is so much lower than the number of shared keyword relationships per node, implying that the shared director relationships are a more useful identifier of relationships when clustering.

4.1.3 Fixed weighting

When creating the graph based on the document frequency of the properties, the frequencies were successfully created into the desired form as described. To create the graph , the nodes with the properties were read in fine. However, issues occurred when trying to create the relationships. As the process is based on producing a table that contains the nodes to connect and their intersection. Due to the number of possible nodes and connections, the production of this table was too demanding computationally. Although this was attempted with all of the properties investigated, none of them were able to produce a table without crashing.

To investigate proof of concept, the technique was tried with a graph of twenty movies. The process was the same but a copy of the csv containing only the first twenty entries was read in to Neo4j. A small version of the desired graph was created. A visualisation of this graph displaying the weights for shared_keywords relationships and their weights can be seen in Figure 4.3

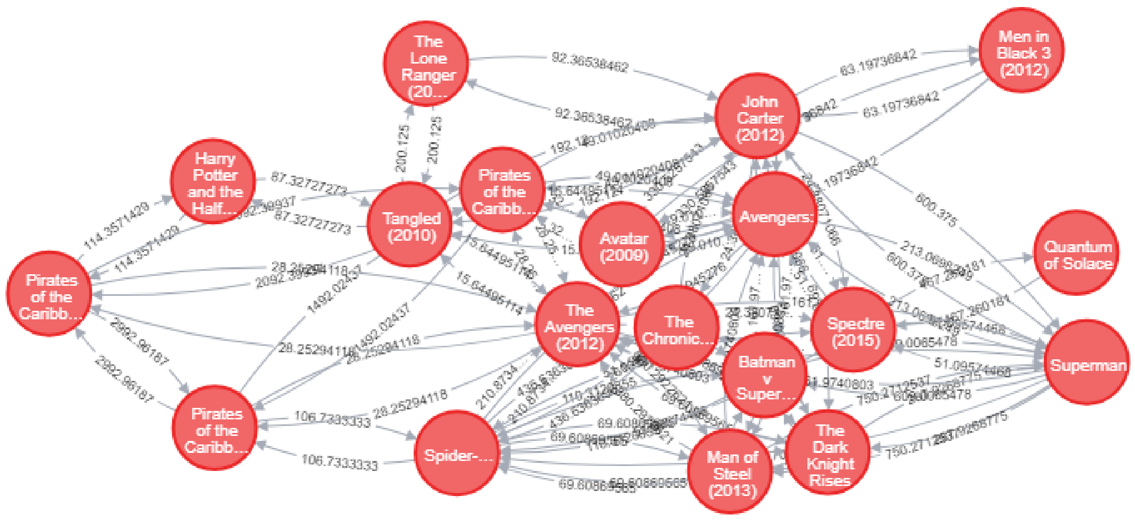

Figure 4.3: Graph showing the frequency weights of shared keywords

While it is unfortunate that the graph could not be investigated further, the proof of concept shows its ability to use the document frequency as weights for relationships. It would be worth investigating this further.

4.1.4 Combining Relationships

The code outlined in Section 3.1.4 ran successfully. This set 847,246 properties, created 847,246 relationships and was completed after 5,439,016 ms. The code successfully set one property per new relationship, as required. This is 176.4 relationships per node. 2.18 times the average relationships per node from the basic weighting result. Considering that averages are skewed by extreme values, this suggests that all the relationships were successfully merged.

Below in Figure 4.4 is a sample of the combined relationship graph database

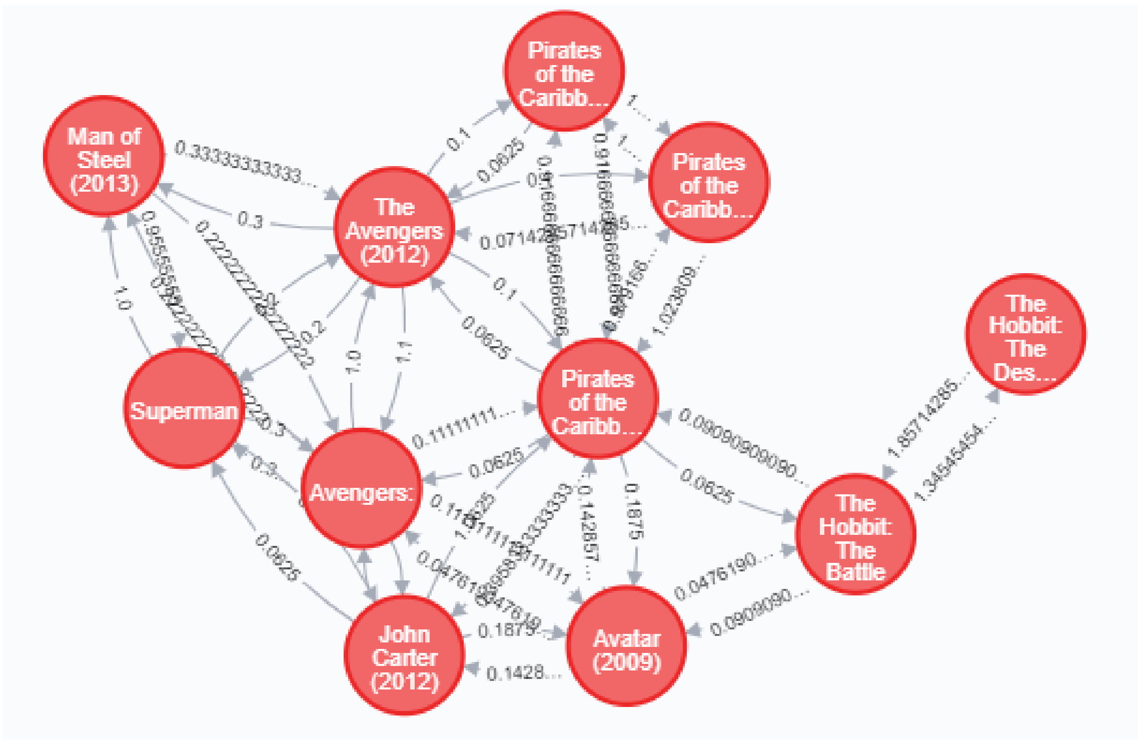

Figure 4.4: Sample of the aggregated relationship graph

From this sample it can be seen that there are many relationships per node, however there are only two relationships between any two nodes, as desired.

A boxplot of the weights of the aggregated relationships can be seen below in Figure 4.5.

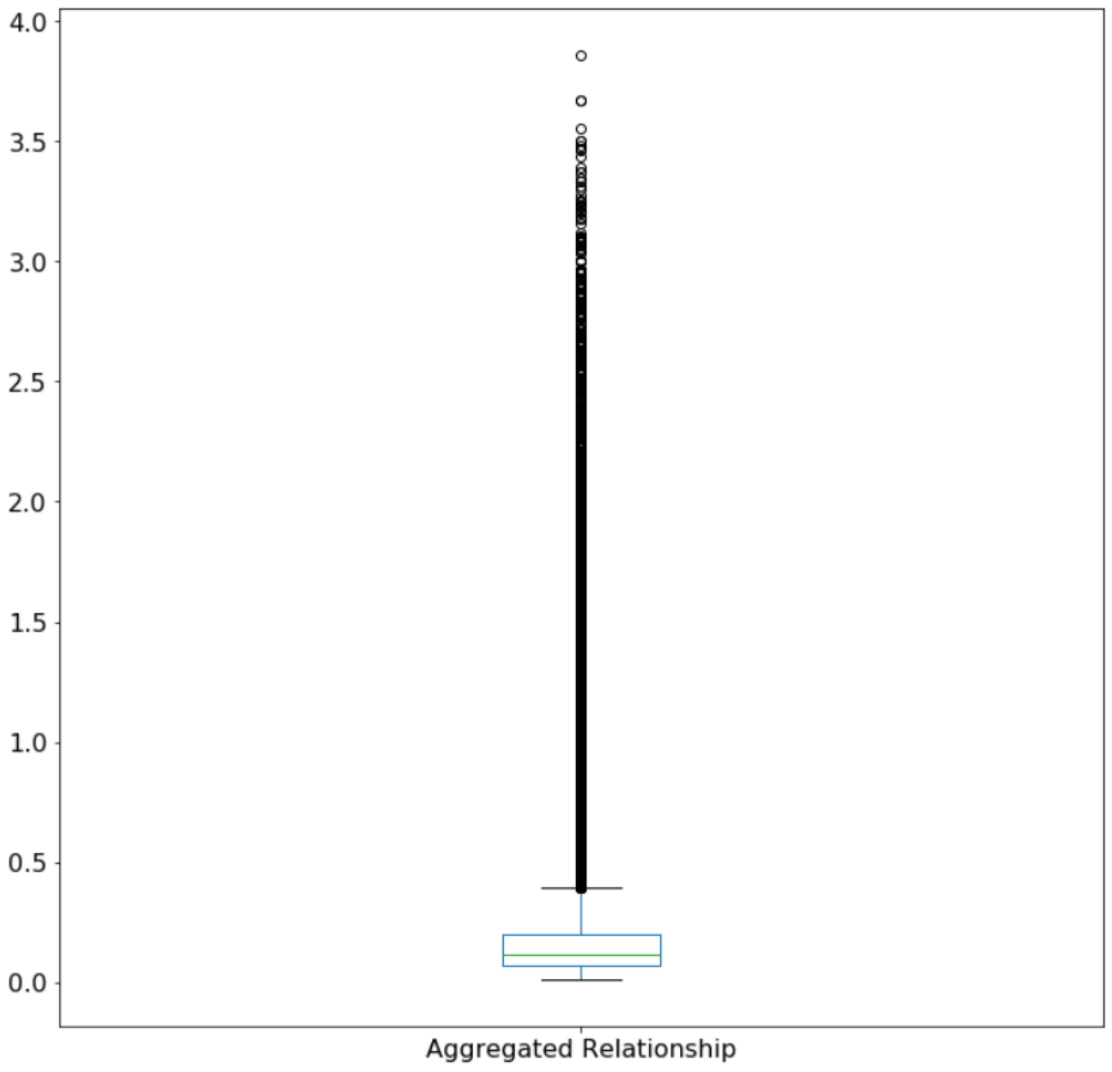

Figure 4.5: Boxplot of the weights of the aggregated relationship graph

It can be seen that the mean and inter-quatile range are both very low. However there are some very large anomalous values. This may suggest effective clustering potential as the few incredibly strong weights will make clear connections over the many weak relationships.

4.1.5 Clustering

Pure graph

The first graph analysed was the pure single node graph, with no weightings added. The first measure was the triangle count and clustering coefficient. A boxplot was made to show the different results, as can be seen in Figure 4.6

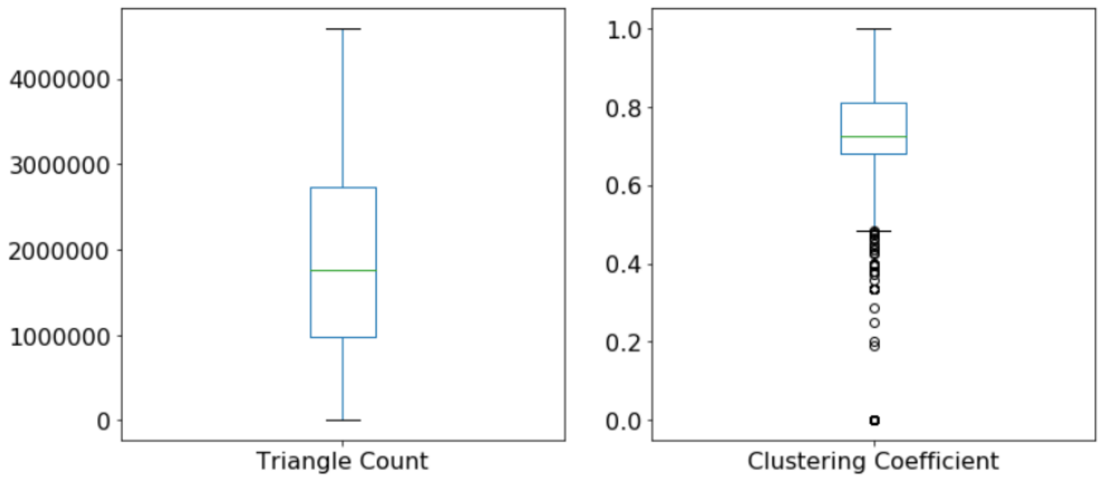

Figure 4.6: Boxplot of the Single node graph

Looking at the triangle count, it can be seen that there is a huge range from zero to almost 5000000. This is not a good sign as it would be preferable for all of the nodes to contain many triangle. Similarly, although clustering coefficient the average of 0.786 is rather height, the spread of the results implies that many of the nodes are not within a suitable cluster. For a recommendation system, it is important that all of the movies are into well defined clusters so that thorough recommendations can be made.

The second algorithm run on this graph was the Strongly Connected Component algorithm. Figure 4.7 below shows the size of the clusters found by the algorithm.

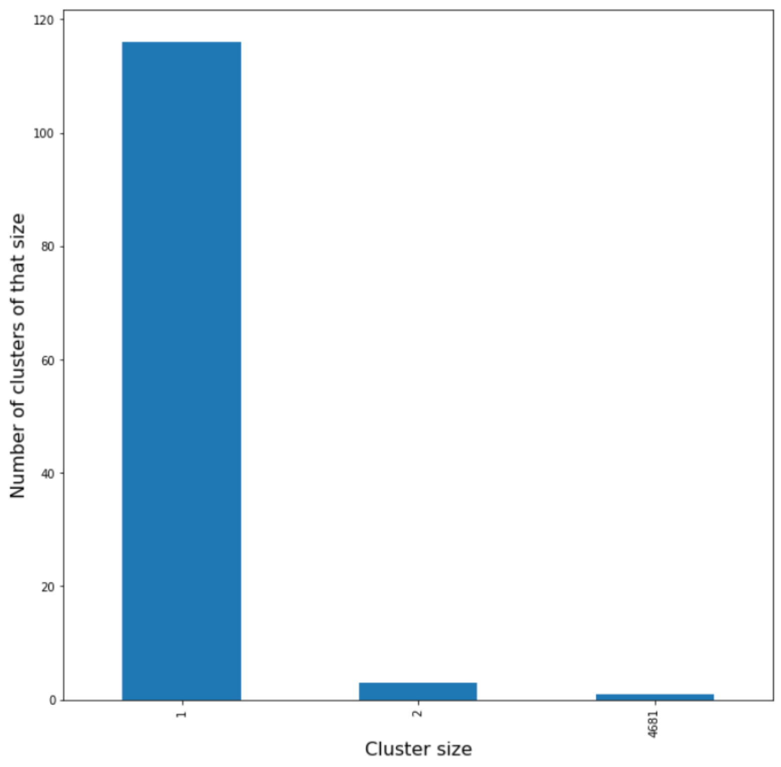

Figure 4.7: Graph showing the sizes of the different clusters, as well as the different number of clusters in that size using the Strongly Connected Component Algorithm

This figure shows that the clustering is very poorly distributed. There is one cluster that contains 4681 of the 4803 movies, the rest fitting into clusters of size 1 or 2. This is very much in contrast to the comparison metrics set out in Section 3.2.1.

This is likely due to the density if the graph. As there are relationships for each aspect of a movie, there are 10,396,650 total relationship. This is what creates such high number of triangles. Since there are so many triangles formed, all of the nodes are too connected for the algorithm to differentiate.

Running the connected component algorithm resulted in the same values as the Strongly Connected Components. As can be seen below in Figure 4.8.

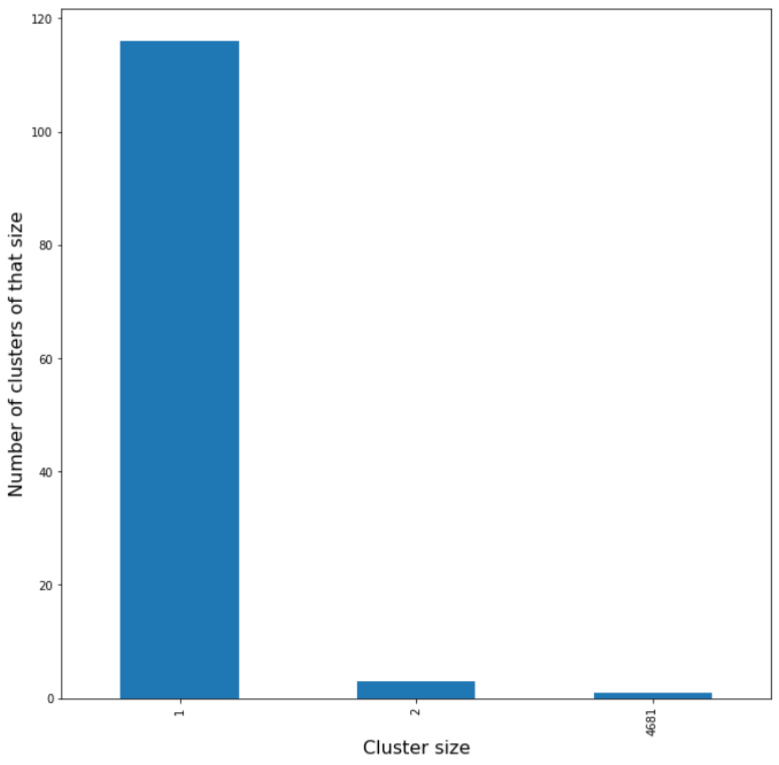

Figure 4.8: Graph showing the sizes of the different clusters, as well as the different number of clusters in that size using the Connected Component Algorithm

This is due to the only difference in the algorithm being that strongly connected components finds groups of nodes that are reachable by following the direction of the relationship. The connected components algorithm, however, ignores the direction. This difference has no effect on graphs of this structure because every directional node between each relationship has another relationship in the opposite direction. So nodes will be reachable regardless of the direction of the relationships.

After running the Louvain algorithm on this graph, the results were plotted in a bar chart as shown below in Figure 4.9

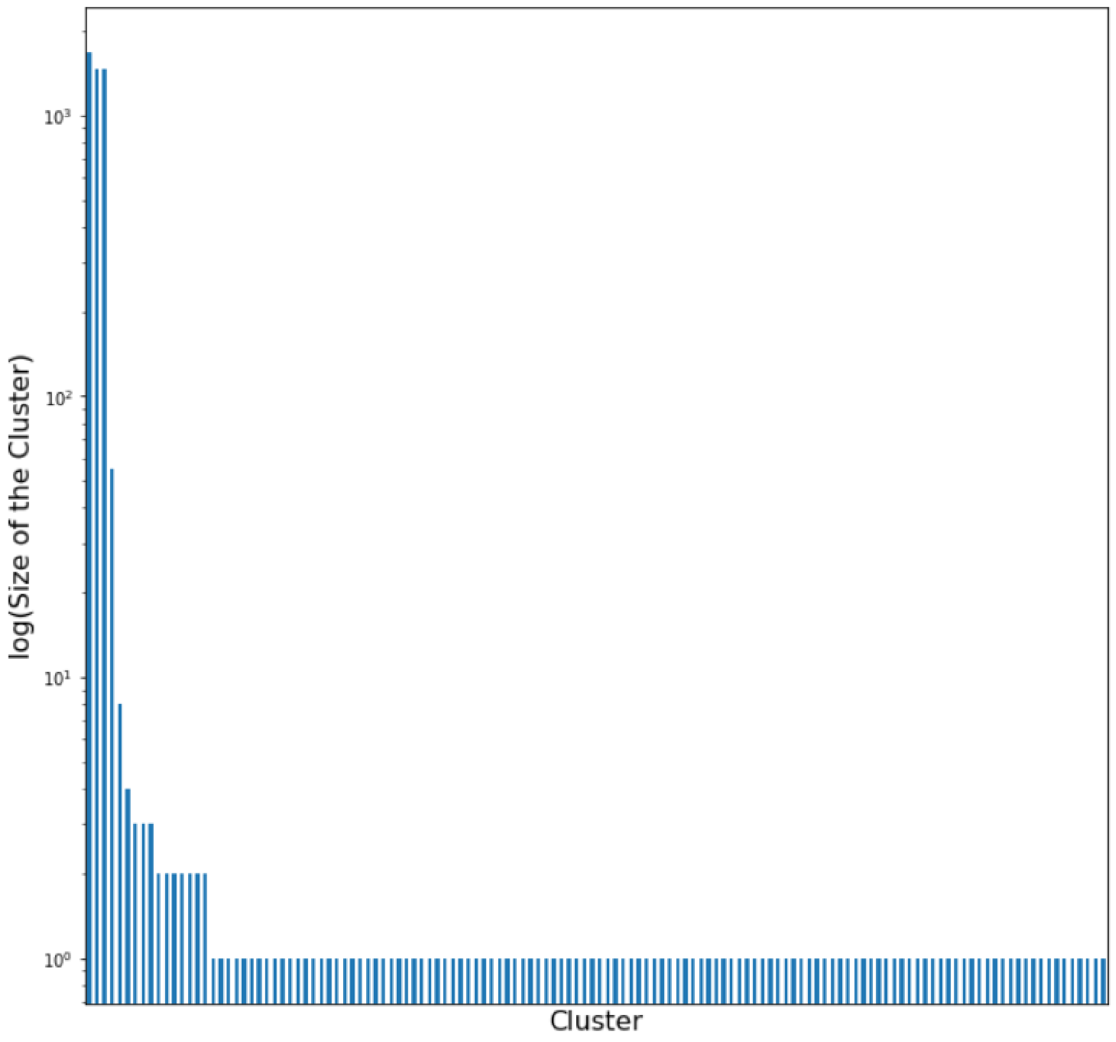

Figure 4.9: Bar chart showing the base ten log of the size of each cluster detected using the Louvain clustering algorithm

[]

In this graph each bar represents a cluster and the y axis is the base ten log of the size. This was done to better compare the size of each cluster, considering the large differences.

In comparison with the strongly connected component and connected component results, there are far more clusters. 129 clusters were detected. Most of the movies cluster into three large clusters, averaging at 1535. 4605 movies were contained into these three clusters. 116 the clusters contained only one movie. Seven clusters contained two movies and three clusters contained three movies. Finally there were three clusters containing 4, 8 and 56 movies. Although more clusters were detected, 89% of the contained only one node. This means that 89% of the clusters can not possibly be used to recommend movies. Although there were more clusters with higher amounts of movies, it is still not enough to be considered for a recommendation system.

The reason for this most likely lies in the number of relationships creating a graph that is too dense. Another aspect is that the Louvain algorithm is largely based around relationship weights. The all the weight here were set to the default of 1.0 reducing the effect of the algorithm.

Lastly, the label propagation algorithm was run. To avoid making the process too computationally demanding, the iterations were set to 10. Another bar chart was made showing the size of the clusters, as can be seen in Figure 4.10

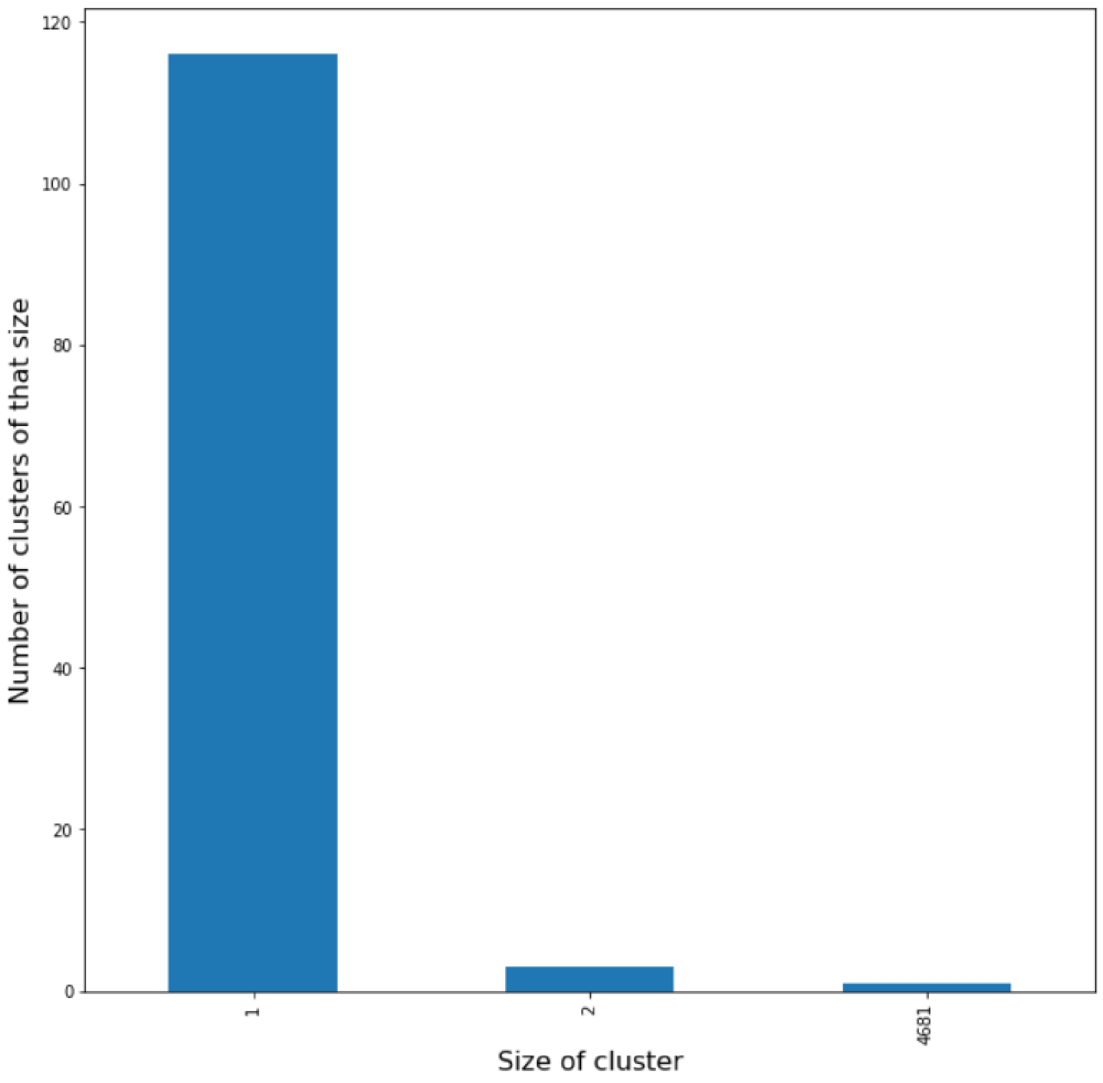

Figure 4.10: Bar chart showing the size of the clusters found by the label propagation algorithm and the number of clusters of that size

Interestingly, the label propagation function found clusters of the same size as the connected component and strongly connected component algorithms. This is likely due to the density of the graph, as well as the lack of weights. This likely resulted in the label propagation algorithm spreading labels in such a concentrated manner. Only the movies that contained very distinct properties could be separate from the main cluster.

4.1.6 Added weights

The clustering algorithms were run on the weighted version of the one label graph. When investigating the weighted version of this graph, there was no need to look into the strongly connected component algorithm. This is because the Neo4j algorithm does not take into account weights. As such, the result would be the same.

The results of the triangle count and the correlation coefficient would also be the same as there was no change to the structure of the graph.

For each of the algorithms run, 4803 clusters were formed. That is every movie formed into its own cluster. The addition of weights into this structure of graph vastly changed how each of the algorithms ran. When there was no weight applied, the density of the graph caused the algorithms to group the movies into few, very large clusters.

The introduction of the weights provided the needed distinction. However, the density of the graph still prevented true patterns from showing. The number of weights between the graph could also have an issue when running the algorithm. When looking at Figure 4.2 previously, the range of weightings between relationships was very large. This most likely resulted in more division.

With regards to the Louvain modularity, the algorithm maximises its presumed accuracy by comparing the density with the weights of relationships. The density of the relationships was very high and the most common relationship, shared keywords, had very low weightings. The algorithm could not build clusters when comparing these two.

Aggregated graph

Due to the restructuring of the graph, the number of relationships were decreased. Hence this reduced the number of triangles formed, and therefore effected the clustering coefficient. Box plots of the new triangle counts and clustering coefficients can be seen below in Figure 4.11

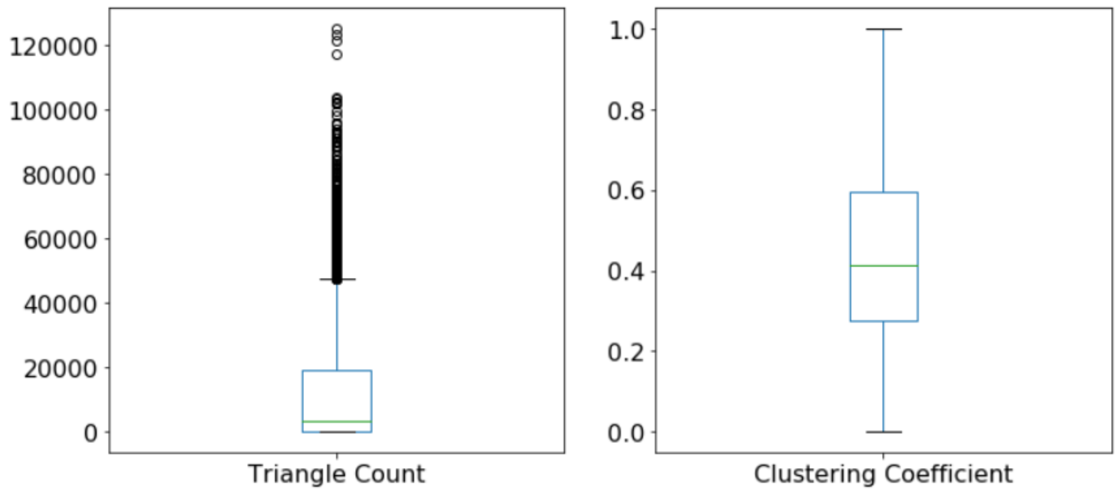

Figure 4.11: Box plots of the triangle count and clustering coefficients of the nodes in the Aggregated One Label Graph

As expected, the number of triangles was greatly reduced, the mean triangle count now being 13472.84. As a result of this, the mean of the clustering coefficient was 0.51 times that of the pre-aggregated graph. This is a notable reduction many movies' ability to cluster.

The first algorithm run here was the connected component graph. The different structure of the reduced the number of connection and should result in more divisions. The new weights of the graph were also taken into account.

Using this algorithm, 678 clusters were detected, however 4126 of these were within one cluster, the rest of the movies were contained in individually in one node clusters. This is far from a positive result

4.2 Multi-label graph

The analysis of the multi-label graphs developed was divided into two parts to correspond with the first two main objectives. The first part was to investigate the building of the graphs, then the application of community based algorithms.

4.2.1 Building the graph

The Cypher queries successfully built a graph containing ACTOR nodes, DIRECTOR nodes, MOVIE nodes and GENRE nodes, as well as the corresponding relationships. A sample of the graph can be seen below in Figure 4.12

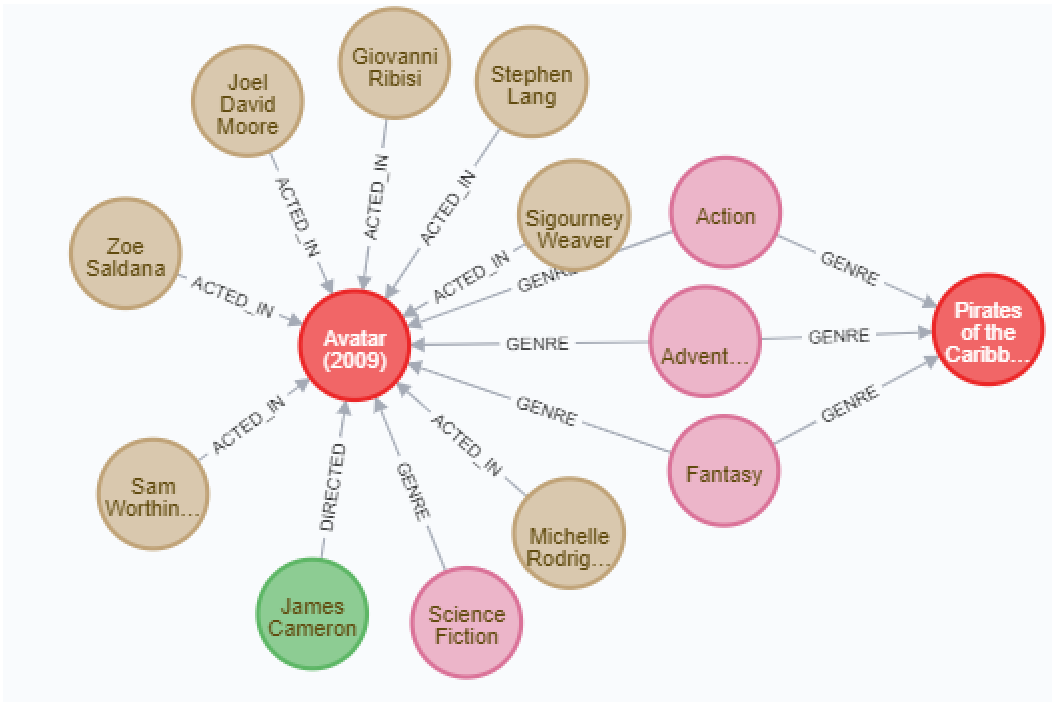

Figure 4.12: A sample of the multi-label graph database

Here brown represents actors, pink represents genres, red represents movies and green represents directors. It cab be seen looking at this figure that the nodes that were wanted were successfully created. It can also be seen that the correct relationships connect the node.

The number of each type of node can be seen below in Table 4.3

Node Type	Number
ACTOR	13,025
DIRECTOR	2417
MOVIE	4803
GENRE	20

Table 4.3: Table showing the number of each type of node

It can be seen here that all of the 4803 movies in the database where added to the graph, as well as all 20 genres. There were 13,025 actors added. This is 3.20 actors per node. This is a reasonable value as seven actors from each movie where collected, but the nodes are distinct for each actor. Hence, the value of actors should be less than 7 to account for crossover in cast. The result here shows that, on average, movies will share 45.7% of their top listed actors.

The number of each type of relationship can be seen below in Table 4.4

Relationship Type Type	Number
ACTED_IN	32791
DIRECTED	4775
HAS_GENRE	12160

Table 4.4: Table showing the number each type of relationship

There are 4803 MOVIE nodes and ACTED_IN relationships were based on the top seven listed actors in that movie. This would imply that there should be 33621 shared actor relationships. However, there are only 32791 ACTED_IN relationships per MOVIE. This is an average of 6.827 actors per movie. Although this is very close to the predicted, the slight discrepancy shows that something had occurred that was unexpected. There were 650 instances of less than seven actors being assigned to a movie. This is likely not due to the process of importing, rather that some movies in the TMdB5000 movie database had fewer than seven actors listed.

Similarly, there were 4775 DIRECTED_BY relationship, meaning there was 0.994 directors per movie. It would be expected that there would be slightly more than 1 director per movie, as all but a few films only have one in the data. The difference in relationships is again very slight and most likely due to a lack of director data in some movies in the TMDB 5000 movie database.

With the HAS_GENRE relationships there were 2.532 relationships per movie node. This sort of value is expected as the trend of movies suggested between 2 and 3 directors per movie.

Comparing the number of relationships with the number of nodes there are 2.48 ACTED_IN relationships per ACTOR node. This means that the average actor is top credited in 2.48 movies. This is half of the amount of the number of movies an average actor is in that was found through research. This implies that half of the roles of the TMDb 5000 movie database actors were top seven credited. This also reveals that each director directs, on average, 2.00 movies. This agrees with the research.

The results from analysing this graph imply that the TMDb 5000 movie database was successfully adapted into a graph database format, confirming the first objective of this investigation.

4.2.2 Investigating Clustering

As the structure of this graph prevented triangles from forming the triangle based algorithms, triangle count, connectivity coefficient, connected component and strongly connected component, were unusable. Hence, clustering analysis was done with Label propagation and Lovain modularity. This graph also had no weights in its relationships and so that aspect of clustering algorithms could also not be investigated.

Louvain Propagation

The number of movies in each cluster, after the mode was taken, can be seen below in Figure 4.13

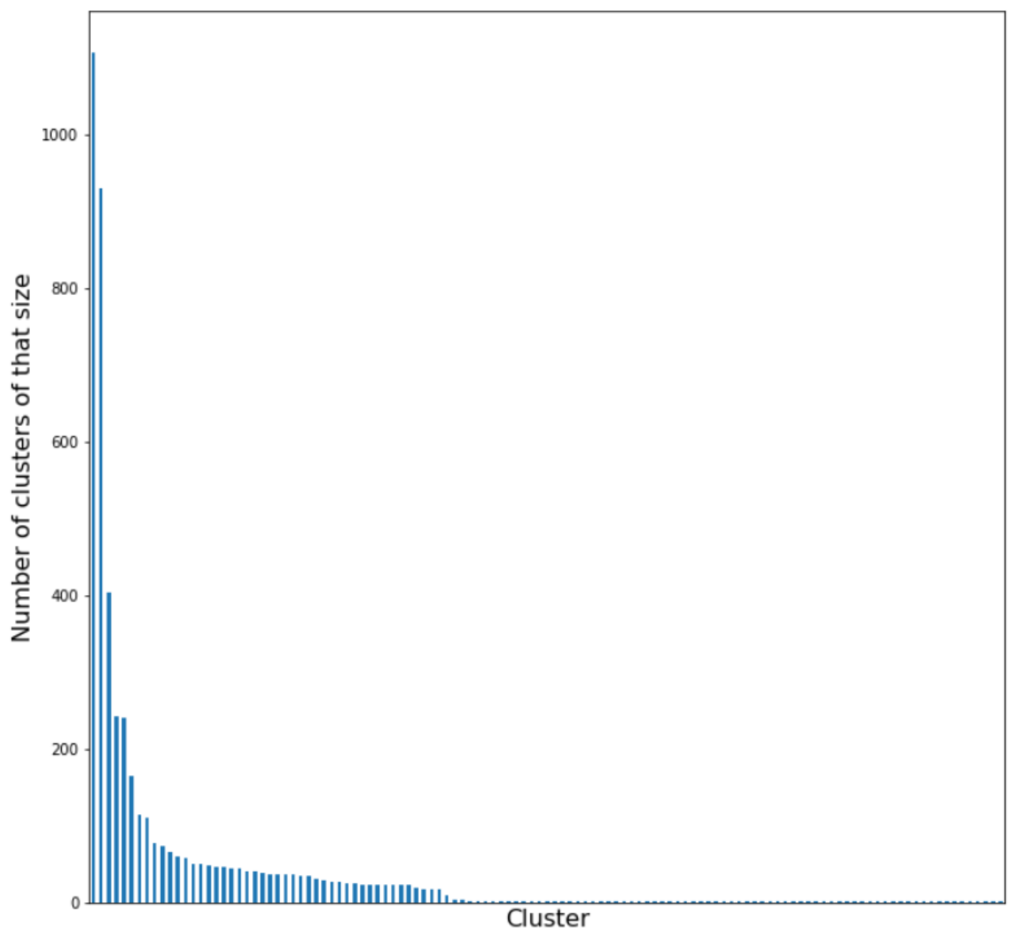

Figure 4.13: Bar chart showing the the number of movies per cluster using Louvain modularity

This algorithm produced 678 clusters with an average of 40.36. As can be seen in this image the nodes are dominated mostly by a few columns, as was much the same with the results form the single label clustering. Although there is a lot more range in the values than seen previously. The largest cluster contained 1106 movies. This means that 0.23 movies exist in one cluster. There were 58 clusters that contained only one movie. When comparing these results with the clustering metrics outlined. The Louvain method does not work well with this structure of graph as a recommendation system. The clusters do not form an even distribution of clusters containing all movies.

There are too many movies that cannot be recommended due to the fact they exist in their own cluster. It is also difficult to make meaningful recommendations when a quarter of all movies fit into the same cluster.

Label Propagation

The same method was applied to the graph gut with label propagation. The results can be seen below in Figure **??**

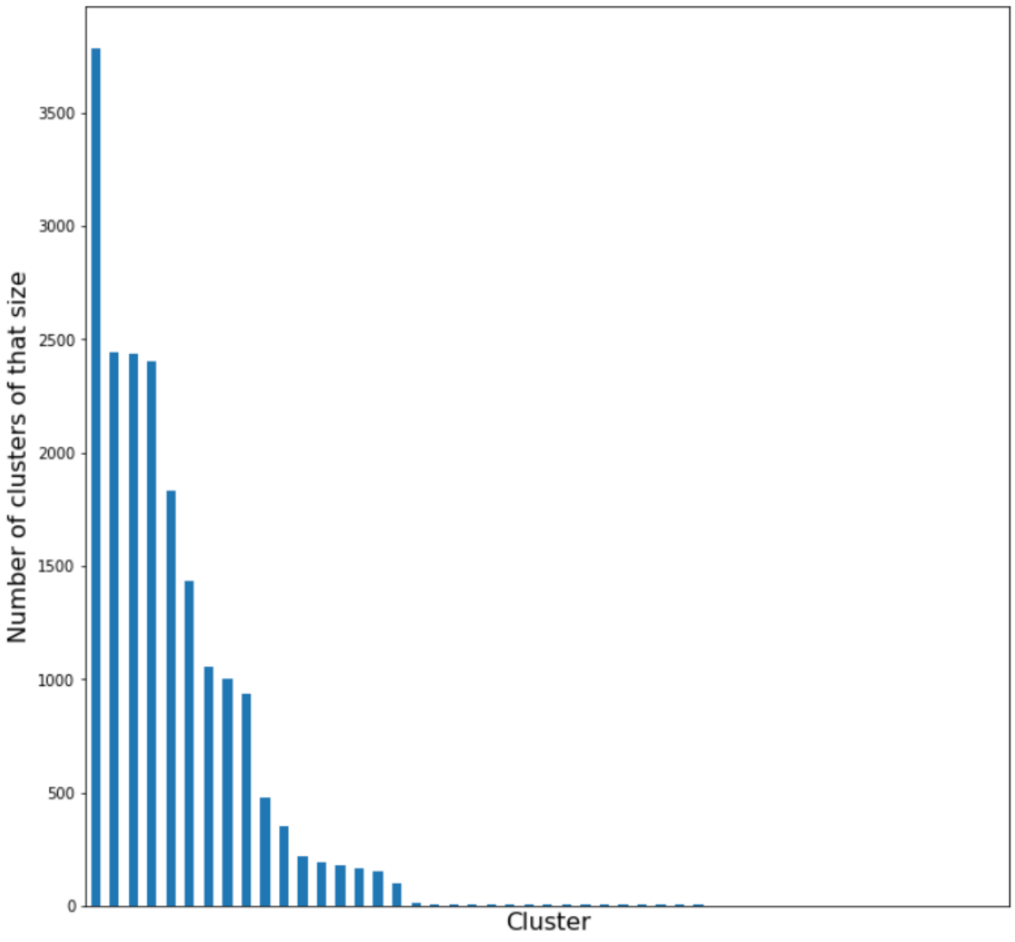

Figure 4.14: Bar chart showing the the number of movies per cluster using label propagation

The most remarkable thing when looking at this bar chart is that the size of the clusters add up to be far more that the number of movies. This is expected as there were many more clusters than the movie labelled ones. It would be worth further investigation to determine if the movies become distributed evenly. The goal of this experiment was to investigated separately clustered movies. Although it may be worth further investigating this form of clustering as a method of movie recommendation.

Even with this in mind, the distribution of the size of the clusters is too great to build a successful recommendation system, as outlined previously.

4.3 Comparing Graphs

When comparing the clustering of the two graph it is clear that non of the approaches used were successfully in producing distinct clusters. However, it was the Lovain modulation on the Multi-Label graph that proved to be the closest to effective. Although it was not possible to fully investigate how the document frequency weighted would have fared against the clustering metrics as there was not enough computation power to fully investigate.

Chapter 5

Conclusion

The two main objectives of this investigation were developing a graph database to represent movie data, and to apply clustering algorithms to it to find clusters that could then be used for a recommendation system. The first objective was successful as there were many varieties of graphs that were constructed. The second objective was not so successful. Although many there were many different approaches, a method of clustering a graph database such that it could be used for a recommendation system was found. After applying multiple clustering algorithms, no result managed to satisfy the metrics that were established to consider the method suitable for a recommendation system.

5.1 Building a Graph Database that contains movie information

Neo4j has proved to be an effective tool for building graphs and applying clustering algorithms. The was used to successfully develop a graph database describing movies. Most of the information on the TMDb 5000 movie database was successfully transferred into a graph database. However much of the data that was described was not suitable for a graph database. There was no method found that could include a movie's budget, any information about the tag-line, overview, budget, revenue or run-time. Any of this information could prove to be useful when using it to develop a recommendation system.

One issue that occurred was the lack of processing power that prevented more complicated weights from being developed. Further investigation into the different methods of term frequency weighting would be a worthwhile endeavour if there is access to higher performance computers.

5.1.1 Unused data

There is also the fact that the TMDb 5000 Movie Database data that was used did not contain all possible pieces of information about the movies. An important example of extra data is the age rating of a movie. For a movie recommendation system to be truly effective it must take into account the age rating of the movie in order to avoid recommending movies with violent or sexual moments to a young user. As movie ratings are discrete labels; Uc, U, PG, 12a, 12, 15, 18, it would be simple to create a node labelled "RATING" and combining it with the multi label graph, connecting it with a "HAS_RATING" relationship to the movie. In terms of the single label graph, a relationship of "shared_rating" could be linked between movies.

Some of the continuous data surrounding movies could be developed into a set of discrete data. For example, the release date in the TMDb database was continuous, however this could have been broken into several discrete values. One method of incorporating this data would be to establish the decade in which the movie was released. Another method of using release date date would be to make nodes or relationships defined by the month in which it was released. There may be unseen patterns in the difference between summer-released movies and winter-released movies.

Also further information about the crew was ignored. It could be possible that recurring editors could make an impact on a movies appeal. One possible reason could be that certain editors edit in an idiosyncratic manner that is notable and preferable for much of an audience. The types of movies that members of the crew affiliate themselves with could also be notably distinct. All of this could have increased made the clusters formed more distinct.

Although, when deciding whether to put more information into the graph, care must be taken to be aware of cluttering the graph, making it too dense to fully analyse. This is why aggregation techniques were included in the single label graph.

5.1.2 Adding weights

For the single label graph, the were many ways in which weights were successfully implemented. Unfortunately it was not possible to full developed the graph based on the frequency of the data. Putting emphasis on the less commonly occurring data could have helped with the clustering, as there would be fewer high weighted relationships throughout the graph, reducing the density. In terms of recommending, emphasis on specific details would also have been useful.

There were no weights included in the multi-label graph. One method off adding weights could be defining the weight by the order in which that value was given in the TMDb 5000 Movie Database. As the data that contained multiple values for each move came in the form of JSON string, it would be possible to use the order of this string as the weight relationship between the information node, for example ACTOR, and the MOVIE node. This is under the assumption that there was priority listing in all JSON strings in the TMDb 5000 Movie Database.

Another way of adding weight to the multi label graph would be to find the shortest path distance between two movie nodes. Although the MOVIE nodes are not directly connected with a relationship, they are connected through other nodes and relationships. This form of connection is called a path [Gubichev et al., 2010].

The path between two movies from the multi label graph can be seen below in Figure 5.1

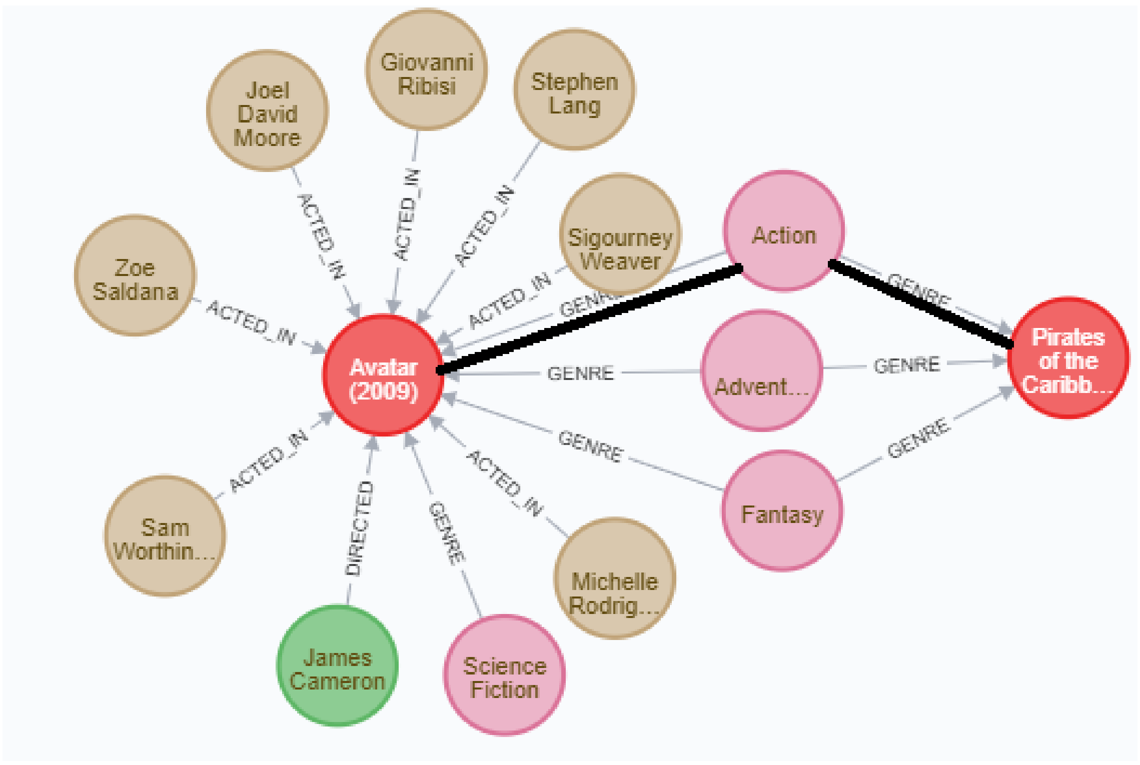

Figure 5.1: A sample of the Multi Label Graph with the path between the two MOVIE nodes highlighted as a black line

The thick black line shows that there the two relationships that connect the Avatar (2009) MOVIE node and the Pirates of the Caribbean, Cruse of the Black Pearl (2003) MOVIE nodes. This is a path distance of 2. If the shortest path between two MOVIE nodes can be found, then a single relationship, weighted by the shortest path length can be used in place of the many unweighted relationships.

The shortest path algorithm is included in Neo4j Graph Algorithm package and so can be used on the systems that have already been developed for this project. The code for this is outlined below:

```
MATCH (source:Place id: "Avatar (2009)"),
(destination:Place id: "Pirates of the Caribbean, Curse of the Black Pearl (2003)")
CALL algo.shortestPath.stream(source, destination)
YIELD nodeIds, cost
```

This code matches the source node, "source" and the destination node "destination" and returns the ids of the nodes, "nodeIds" and the length of the path, "cost".

The algorithm is based around the idea of "tentative distance", giving a connection between two nodes an estimated length and slowly improving it [Dial, 1969]. The algorithm works in six steps [Fredman and Tarjan, 1987]:

1. Define every node in the graph as "unvisited", form a set from these called the "unvisited set"
2. Choose a current node, define it current and assign a tentative value of 0 to it and a tentative value of infinity to all other nodes.
3. Consider all of the current nodes unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
4. Define the current node "visited" and remove from the "unvisited set"
5. If the destination node has been defined as "visited then stop the algorithm.
6. Otherwise select the unvisited node with the smallest tentative distance

Performing this would remove fears of creating an overly dense graph. There was also the fear that clustering with this many different labelled nodes would result in clusters that do not contain MOVIE nodes. Condensing the information in this graph to one label of nodes and a relationship based on path differences could help resolve that.

One issue with applying weights that are calculated from different pieces of information is the range of the weights are different for each type of relationship. This also affects the aggregation of weights. Further work done investigated these structures of graphs should look into the effect of normalising the weights, or fond some other way of making the different types of weight comparable

5.2 Investigating Clustering

Clustering metrics were established to define whether the algorithm successfully divided the data into clusters that were suitable for recommending movies. Overall none of the algorithms performed well enough on any of the variations of the graph databases.

5.2.1 Triangle count and Clustering coefficient

The triangle count and clustering coefficient were used initially on the single node graph to investigate each nodes ability to form into a cluster. The result of the indicated that the single node graph with multiple relationships produced a very high triangle count and clustering coefficient. Although further investigation showed that this was a poor indication of the graphs clustering ability. This was due to the high density of the graph. The algorithm worked by counting the number of triangles formed by connecting three nodes with relationships. As there were so many relationships between nodes there where a very high number of triangles. This did not reflect the actual clustering potential between nodes, as there were too many relationships between them to find a sensible result.

For the aggregated graph, the triangle count was much lower with some very high anomalous values. While this may appear to be deterioration of result, fewer triangles is an indication of the reduction in tensity. Fewer triangles were wanted as the previous had far too many. The very high outliers could still imply that very large clusters will form. Further investigation into triangle based clustering algorithms could reveal a more positive result with this structure.

The multi label graph could not use triangle count or the clustering coefficient due to the structure of the graph not allowing triangles to form. Combining the information in the multi label node into a single label graph, as mentioned previously, would result in triangles. However, this would not reveal much about the original structure. It would be useful to be able to link nodes with the same label. This can be done with graph algorithms called link prediction.

Link prediction

Link prediction is used to determine the closeness between two nodes. Closeness is a numeric value where the range is unique to each algorithm [Liben-Nowell and Kleinberg, 2007]. This closeness can be used as the bases for a relationship between two same-labeled nodes. After calculating the closeness, a relationship can be made between two nodes with the result being used as the weight for the relationship. An issue that could arise is blindly applying this technique would result in a link between all nodes, rendering all community detection algorithms useless. To combat this, a cut-off point would have to made on the weights, only the nodes whose link prediction is above this value should have the relationship added.

This would be simply done in Neo4j based on the graph that were already constructed. There are seven link prediction algorithms already contained in the Neo4j Graph Adamic Adar, Common Neighbors, Preferential Attachment, Resource Allocation, Same Community and Total Neighbors.

With regards to movie recommendation, K. Yu et al found successful results using stochastic relational models (STM's) in movie recommendation in their 2003 paper Stochastic Relational Models for Discriminative Link Prediction [Yu et al., 2007]. This approach could be used in combination with the graph to develop the new relationships. However, there are no packages in Neo4j that apply STM's. Investigating this further would require developing an algorithm with Cypher, most likely in combination with programming language such as Python.

5.2.2 Connected Component and Strongly Connected Components

The only graph structures that could implement these were the single node graphs. The previously mentioned link prediction would make it possible to apply this to the multi label graph in further work. The effect of the weight of the predicted link would be useful investigation in connected component

Both the weighted and unweighted single label graphs produced the same result.for the weighted connected component algorithm. And so it most likely is not worth further investigation into this structure with the connected component algorithm.

The Strongly connected component provided the same poor results. However, the inclusion of weight in some manner could potentially improve the system. This is not possible with the built in Neo4j and so it would have to be developed with a combination of Cypher and Python.

5.2.3 Label Propagation

Both the weighted and un-weightd single label graphs had a Label Propagation result that merges all but a few movies into one very large node making it clearly unacceptable for recommendation system.

The weighted graph only produced 4803 individual clusters. This is not usable for recommendation in any form. Although only the non aggregated relationship graph was investigated. It is worth a further look into all variants of the graph to see if the results are more positive. It is also worth further investigation into normalising the weights when applying this as that may cause an issue. Differences in the number of iterations was also not investigated and so that is also worth further investigation.

The multi label graph had much more varied results, indicating that this structure of graph is more suitable for Label Propagation. Although there were still far too many single node clusters that could contain MOVIE nodes. This prevents any form of recommendation for those movies.

Because all of the nodes where clustered, it was not possible to tell how well distributed the MOVIE nodes where. It is possible that by removing the non-MOVIE nodes from the set, the resulting clusters would be more evenly distributed . All the non-MOVIE nodes were centered around MOVIE nodes, meaning that the MOVIE nodes had the most relationships. This could potentially mean that its these nodes that responded best to clustering, and hence are less likely to be in the single-node clusters.

5.2.4 Lovain Modularity

Louvain modularity provided very poor results for all graph structures. When looking at the single node graph, the weighted result produced a greater number of clusters than the non-weighted. It also produced better results than any of the label propagation algorithms run on it. Therefore, when investigating a single node graph database representing movies, Louvain Modularity is the best choice for further investigation. It was not possible to investigate its effect on the term frequency graph due to limited processing power, and there was no time to apply this to the aggregated graph. Therefore there is a lot of room for further investigation focusing on this combination.

Louvain modularity on the multi label graph gave a similar distribution to the label propagation clustering algorithm. There was again a large distribution of cluster sizes. There were more single node clusters in this result however. Indicating that the Label propagation is the preferred method for this structure when clustering with the intent to develop clusters that can be used to recommend movies.

There is the repeated issue of the clusters containing both MOVIE and non MOVIE nodes. A proper comparison of the result of label propagation and Louvain modularity when applied to the multi label graph can only be done once the clusters have been removed of the no MOVIE nodes.

5.3 Comparing Graphs

When looking at the construction of the graphs, both the basic, weightless single node graph and the multi label graphs were successfully constructed with Neo4j. Both of these graphs where able to be queried using Neo4j's querying language Cypher. Although the single node graph displayed more information about each movie in its nodes and relationships.

The single node graph was able to be adapted to add weights to the relationships. This was not able to be done with the multi label graph. However, potential developments were suggested to further study this. Because of this, there where no further developments of the multi label graph. However, for the single nodes graph, there where three graphs that were built that successfully in Neo4j that contained all of the TMBd 5000 Movie Database information, and one that successfully built a sample. From these results, the single label graph was better for developing graph databases.

The single label graph was able to have all of the discussed algorithms performed on it, unlike the multi label graph, for all but the term frequency graph due to its size. Although the potential alterations suggested would allow for this to be done. However, it was the result of both the label propagation and the Louvain modularity that performed better than any result form the single label graph when using equal distribution as a metric. Although, this cannot yet be truly confirmed as the clusters do not just contain nodes that represent the movies. It is only the clustering of these nodes that can be used to investigate movie recommendation.

5.4 Combining techniques with machine learning

Another possible route of further study is to combine these clustering algorithms with other techniques such as machine learning. One example of this is to initially run a convolutional neural network on the graph to predict missing links. This process can be used to build new, computationally, relationships between nodes that may prove useful when clustering. Al Hasan et al showed the strength link prediction using this technique in their 2006 paper [Al Hasan et al., 2006] where they proved the technique able to find missing links in social media.

Using this on the multi label graph could also add relationships between same labelled nodes, allowing for the use of triangle based clustering algorithms. It would also be worth using this technique before the shortest path algorithm to see if the inter-movie relationship weights discussed are altered by this in a way that improves clustering results for movie recommendation.

5.5 Movie Recommendation

This paper investigated the clustering of movie graphs with the ultimate goal of developing a system that can be used as a recommendation system. However, due to the clustering techniques used not producing results that satisfy the established metrics, there was no opportunity to experimentally determine if this recommending movies within the same cluster would work as a functional recommendation system.

This could be done by analysing a second set of data that contains user ratings for movies. This was done her using data collected of peoples movie reviews, using movies rated positively by a person and determining their ideal cluster, an ideal dataset is the MovieLens data [GroupLens, 2018]. The data contains a user id, a movie title, a score 1-5 in increments of 0.5 that describes that users enjoyment of the movie. A rating of 5 suggests maximum enjoyment, while a rating of 1 suggests minimum enjoyment. Positive movies defined as a rating of 3+ could be the indicator for an enjoyed movie. The cluster with the highest amount of users positive ratings would be used as the recommendation cluster. A test-train split should be used to investigate this experimentally.

While the results of investigating clustering for movie recommendation was inconclusive, there is still a lot of potential work that could be done in this area.

Bibliography

[(tm, 2019] (2019). The movie database.

[Al Hasan et al., 2006] Al Hasan, M., Chaoji, V., Salem, S., and Zaki, M. (2006). Link prediction using supervised learning. In *SDM06: workshop on link analysis, counter-terrorism and security*.

[Angles and Gutierrez, 2008] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39.

[Assefi et al., 2015] Assefi, M., Liu, G., Wittie, M. P., and Izurieta, C. (2015). An experimental evaluation of apple siri and google speech recognition. *Proccedings of the 2015 ISCA SEDE*, pages 1–6.

[Bar-Ilan and Peleg, 1991] Bar-Ilan, J. and Peleg, D. (1991). Approximation algorithms for selecting network centers. In *Workshop on Algorithms and Data Structures*, pages 343–354. Springer.

[Blondel et al., 2008] 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.

[Bobadilla et al., 2012] Bobadilla, J., Ortega, F., Hernando, A., and Bernal, J. (2012). A collaborative filtering approach to mitigate the new user cold start problem. *Knowledge-Based Systems*, 26:225–238.

[Brusilovsky and Millán, 2007] Brusilovsky, P. and Millán, E. (2007). User models for adaptive hypermedia and adaptive educational systems. In *The adaptive web*, page 352. Springer.

[Buerli and Obispo, 2012] Buerli, M. and Obispo, C. (2012). The current state of graph databases. *Department of Computer Science, Cal Poly San Luis Obispo, mbuerli@ calpoly. edu*, 32(3):67–83.

[Bunch et al., 2011] Bunch, C., Chohan, N., Krintz, C., and Shams, K. (2011). Neptune: a domain specific language for deploying hpc software on cloud platforms. In *Proceedings of the 2nd international workshop on Scientific cloud computing*, pages 59–68. ACM.

[Businessweek, 2013] Businessweek, B. (2013). How disney bought lucasfilm—and its plans for 'star wars'.

[Cha et al., 2009] Cha, M., Kwak, H., Rodriguez, P., Ahn, Y.-Y., and Moon, S. (2009). Analyzing the video popularity characteristics of large-scale user generated content systems. *IEEE/ACM Transactions on networking*, 17(5):1357–1370.

[Chung et al., 2017] Chung, H., Park, J., and Lee, S. (2017). Digital forensic approaches for amazon alexa ecosystem. *Digital Investigation*, 22:S15–S25.

[Clauset et al., 2004] Clauset, A., Newman, M. E., and Moore, C. (2004). Finding community structure in very large networks. *Physical review E*, 70(6):066111.

[Devine, 1989] Devine, P. G. (1989). Stereotypes and prejudice: Their automatic and controlled components. *Journal of personality and social psychology*, 56(1):5.

[Dial, 1969] Dial, R. B. (1969). Algorithm 360: Shortest-path forest with topological ordering [h]. *Communications of the ACM*, 12(11):632–633.

[Duda and Hart, 2001] Duda, R. O. and Hart, P. E. (2001). Dg stork pattern classification. *John Wiely and Sons.*

[Fatemi and Tokarchuk, 2013] Fatemi, M. and Tokarchuk, L. (2013). A community based social recommender system for individuals & groups. In *2013 International Conference on Social Computing*, pages 351–356. IEEE.

[Fleischer et al., 2000] Fleischer, L. K., Hendrickson, B., and Pınar, A. (2000). On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium*, pages 505–511. Springer.

[Follows, 2013] Follows, S. (2013). How many films in an average film career.

[Fouss et al., 2006] Fouss, F., Yen, L., Pirotte, A., and Saerens, M. (2006). An experimental investigation of graph kernels on a collaborative recommendation task. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 863–868. IEEE.

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615.

[GroupLens, 2018] GroupLens (2018). Movielens 20m dataset.

[Gubichev et al., 2010] Gubichev, A., Bedathur, S., Seufert, S., and Weikum, G. (2010). Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 499–508. ACM.

[Heckemann et al., 2006] Heckemann, R. A., Hajnal, J. V., Aljabar, P., Rueckert, D., and Hammers, A. (2006). Automatic anatomical brain mri segmentation combining label propagation and decision fusion. *NeuroImage*, 33(1):115–126.

[Holzschuher and Peinl, 2013] Holzschuher, F. and Peinl, R. (2013). Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM.

[Huang et al., 2002] Huang, Z., Chung, W., Ong, T.-H., and Chen, H. (2002). A graph-based recommender system for digital library. In *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 65–73. ACM.

[Kannan et al., 2004] Kannan, R., Vempala, S., and Vetta, A. (2004). On clusterings: Good, bad and spectral. *Journal of the ACM (JACM)*, 51(3):497–515.

[Karlgren, 1994] Karlgren, J. (1994). Newsgroup clustering based on user behavior-a recommendation algebra. *SICS Research Report.*

[Kemper, 2015] Kemper, C. (2015). *Beginning Neo4j.* Springer.

[Kleinberg, 2002] Kleinberg, J. M. (2002). Small-world phenomena and the dynamics of information. In *Advances in neural information processing systems*, pages 431–438.

[Lakiotaki et al., 2011] Lakiotaki, K., Matsatsinis, N. F., and Tsoukias, A. (2011). Multicriteria user modeling in recommender systems. *IEEE Intelligent Systems*, 26(2):64–76.

[Lalwani et al., 2015] Lalwani, D., Somayajulu, D. V., and Krishna, P. R. (2015). A community driven social recommendation system. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 821–826. IEEE.

[Lam et al., 2008] Lam, X. N., Vu, T., Le, T. D., and Duong, A. D. (2008). Addressing cold-start problem in recommendation systems. In *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 208–211. ACM.

[Liben-Nowell and Kleinberg, 2007] Liben-Nowell, D. and Kleinberg, J. (2007). The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7):1019–1031.

[Mark Needham, 2019] Mark Needham, A. E. H. (2019). *Graph Algorithms: Practical Examples in Apache Spark Neo4*, volume 1 of *1*. O'Reilly, 1 edition.

[McKinney, 2012] McKinney, W. (2012). *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.".

[Mojo, 2019] Mojo, B. O. (2019). 2019 studio market share.

[Montenieri, 2018] Montenieri, A. J. (2018). Digital streaming: Technology advancing access and engagement in performing arts organizations.

[Newman, 2006] Newman, M. E. (2006). Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582.

[Nuutila and Soisalon-Soininen, 1994] Nuutila, E. and Soisalon-Soininen, E. (1994). On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14.

[Pogue, 2007] Pogue, D. (2007). A stream of movies, sort of free. *The New York Times. Available at: https://www. nytimes. com/2007/01/25/technology/25pogue. html [Accessed 31st December 2017]*.

[Ricci et al., 2011] Ricci, F., Rokach, L., and Shapira, B. (2011). Introduction to recommender systems handbook. In *Recommender systems handbook*, pages 1–35. Springer.

[Rich, 1979] Rich, E. (1979). User modeling via stereotypes. *Cognitive science*, 3(4):329–354.

[Sasaki, 2018] Sasaki, B. M. (2018). 'why graph databases are the future'.

[Schaeffer, 2007] Schaeffer, S. E. (2007). Graph clustering. *Computer science review*, 1(1):27–64.

[Schafer et al., 2001] Schafer, J. B., Konstan, J. A., and Riedl, J. (2001). E-commerce recommendation applications. *Data mining and knowledge discovery*, 5(1-2):115–153.

[Schank and Wagner, 2005] Schank, T. and Wagner, D. (2005). Finding, counting and listing all triangles in large graphs, an experimental study. In *International workshop on experimental and efficient algorithms*, pages 606–609. Springer.

[Shao et al., 2009] Shao, B., Wang, D., Li, T., and Ogihara, M. (2009). Music recommendation based on acoustic features and user access patterns. *IEEE Transactions on Audio, Speech, and Language Processing*, 17(8):1602–1611.

[Smith and Linden, 2017] Smith, B. and Linden, G. (2017). Two decades of recommender systems at amazon. com. *Ieee internet computing*, 21(3):12–18.

[Sowa, 1976] Sowa, J. F. (1976). Conceptual graphs for a data base interface. *IBM Journal of Research and Development*, 20(4):336–357.

[Tarjan, 1972] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160.

[Ungar and Foster, 1998] Ungar, L. H. and Foster, D. P. (1998). Clustering methods for collaborative filtering. In *AAAI workshop on recommendation systems*, volume 1, pages 114–129.

[Vicknair et al., 2010] Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. (2010). A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM.

[Wang et al., 2008] Wang, J., Wang, F., Zhang, C., Shen, H. C., and Quan, L. (2008). Linear neighborhood propagation and its applications. *IEEE transactions on pattern analysis and machine intelligence*, 31(9):1600–1615.

[Wu et al., 2012] Wu, Z.-H., Lin, Y.-F., Gregory, S., Wan, H.-Y., and Tian, S.-F. (2012). Balanced multi-label propagation for overlapping community detection in social networks. *Journal of Computer Science and Technology*, 27(3):468–479.

[Yoon et al., 2017] Yoon, B.-H., Kim, S.-K., and Kim, S.-Y. (2017). Use of graph database for the integration of heterogeneous biological data. *Genomics & informatics*, 15(1):19.

[Yu et al., 2007] Yu, K., Chu, W., Yu, S., Tresp, V., and Xu, Z. (2007). Stochastic relational models for discriminative link prediction. In *Advances in neural information processing systems*, pages 1553–1560.