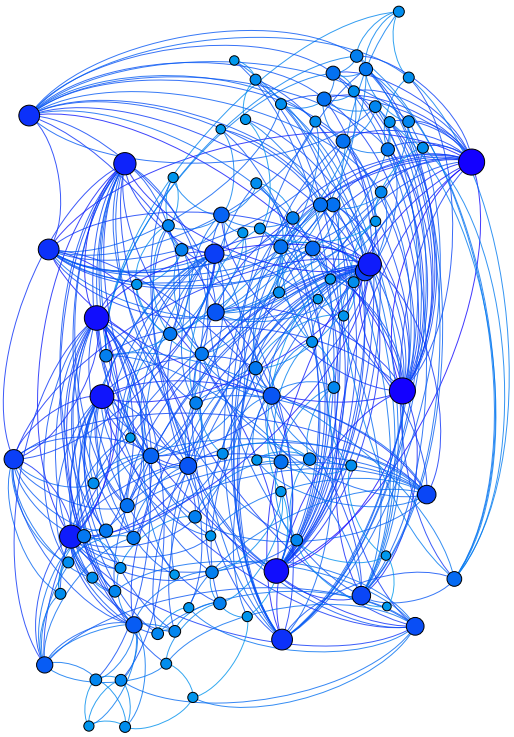
University of Zurich[UZH]

Social Network Analysis with Signal/Collect



Bachelor Thesis
December 1, 2014

Flavio Keller
of Wädenswil ZH, Switzerland

Student-ID: 11-714-615
flavio.keller@uzh.ch

Advisor: **Daniel Spicar**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I would like to take the opportunity to thank anybody involved in this Bachelor Thesis project.
First of all my thanks go to Prof. Abraham Bernstein and his research group for providing me with such an interesting environment and giving me the opportunity to develop and write this thesis.
Sincere thanks also go to my advisor, Daniel Spicar, PhD Student and Research Assistant at the DDIS (Dynamic and Distributed Information Systems Group), who supported and guided me through the whole project, always had some advice ready when I ran into problems and motivated me to work on this thesis.
Eventually, I would like to thank all my friends and family who kept me motivated on writing this thesis and especially those people who proofread my thesis and gave some advice on how to improve it.

Flavio Keller
Zürich, Switzerland
December 1, 2014

Zusammenfassung

Das Signal/Collect Framework, welches an der Universität Zürich entwickelt wurde, ist ein Ansatz, um die Hürde der Bearbeitung von Informationsfluss in grossen Netzwerken zu überwinden. Die Hauptstärke von Signal/Collect liegt aber in seiner Fähigkeit verteilt auf mehreren Maschinen zu arbeiten.
Das Hauptziel dieser Arbeit ist es, Masse zur Analyse von Netzwerken in Signal/Collect zu implementieren. Der Fokus liegt dabei auf Zentralitätsmassen und Netzwerkeigenschaften. Diese Masse zeigen auf, welche Teile des Netzwerks Einfluss auf das Gesamtnetzwerk haben oder welche Teile des Netzwerks eine Gemeinschaft bilden.
Die implementierte Lösung ist die Ergänzung eines bestehenden Tools, welches alle diese Netzwerkanalysemethoden ausführen kann.
Zusätzlich wurde eine weitergehende Methode implementiert, die sich "Label Propagation" nennt und eine Möglichkeit bietet, um zusammengehörende Teile in einem Netzwerk zu erkennen sowie auch diese über die Zeit zu verfolgen.
Die implementierten Funktionalitäten wurden schliesslich auf einem Cluster von Computern auf Korrektheit und Laufzeit evaluiert.

Abstract

The Signal/Collect framework, developed at the University of Zurich is an approach to face the challenge of handling and passing information in large graphs. Its main power lies in its ability to work distributed on multiple machines.
The main goal of this thesis is to implement Social Network Analysis measures on the Signal/Collect framework. The focus lies on centrality measures and network properties. These measures reveal what parts of a network have influence on the whole network or try to find communities.
The implemented solution is an extension of an existing graph tool with a plugin where all these Social Network Analysis measures can be executed.
Furthermore, a more advanced method called "Label Propagation" was implemented which is a way to detect communities in a network and makes it possible to see how these communities change over time.
The implemented functionalities were evaluated on a cluster of computers for correctness of the results as well as for computation time.

Table of Contents

1

Introduction

A Network (or synonymously graph) is a construct that consists of nodes (or synonymously vertices) that are connected via edges which transport some sort of information. Networks can be observed in many aspects of life and have become more relevant in recent decades. For instance, there are social networks, neural networks in a brain or a network of computers.
Networks may come in all different flavours. Every node can be connected with every other, one node is connected to all others which do not have any other connection, any structure one can imagine is actually possible.
There are also different types of edges in a network, namely directed or undirected. The difference is, that a directed edge transports information in only one direction whereas undirected edges are able to do this in both directions.
Distributed computing is a very crucial concept in today's world in order to make systems scalable or more efficient and keeping costs low at the same time. To achieve distributed computing, we need a network of computers. Distributed computing, as the name states, can be imagined as a computation task split up into many smaller tasks which are assigned to the machines in the network and after each machine is done, the results are gathered. The benefit thereof is that we can add a machine to the network or remove it from the network at any point in time and the computation power of a network may be changed instantly.
Since there is this large variety of how a network may be built, it is of interest to get knowledge on how this network is structured and whether there are efficiency problems (e.g bottlenecks) in this network when one wants to do distributed computing on such a network of machines.
In order to do so, it is helpful to use certain measures. These measures can be used to reveal if the network works efficiently and if not, in which parts of the network it fails to do so. They also should tell something about how the parts of the network are connected together. As mentioned before, it makes a difference if every machine is connected with

every other or not. A user of a network should eventually be able to get to some conclusion when analyzing a network with these measures. For instance, he or she wants to be able to decide if the network may be improved by changing its structure. This decision can be supported with these measures. This thesis is concerned with well-known measures of Social Network Analysis, which focus on questions of centrality in a network, show what parts have more influence on the whole network and make statements about parts of the network which form some kind of community [Newman, 2008].
The aim is to develop a tool that calculates basic and advanced methods of Social Network Analysis on a distributed computing environment. Eventually it should be shown that it is possible to implement such a tool on a distributed framework. Usually, the mentioned Social Network Analysis measures are gathered centrally. But this is only applicable if we have a non-distributed network. So every node in a network has to compute its measure value on its own. When each node has done so, we can gather the values and interpret them or form statistics.
The Signal/Collect programming model, developed at the University of Zurich, a tool for parallel graph proceessing is able to perform distributed computing in a network [Stutz et al., 2010]. The main concept is to let edges signal some sort of information whereas nodes collect the signals of all incoming edges and process this incoming information.
Hence, Signal/Collect is a suitable base to build the described tool upon. The developed tool should, in combination with an existing graph visualization tool, as well be able to illustrate the desired graph/network. Therefore, after discussing related and existing work, the thesis covers an evaluation of existing graph tools. Following that, the next part is about the actual implementation of the network analysis methods in Signal/Collect. It describes the implementation of basic and advanced measures as well as the implementation of network properties. Subsequently, the thesis is concerned with the evaluation of the implemented measures and closes with the chapters concerning conclusions and future work.

2

Related Work

It is important to get an overview of the field of science this thesis is located in and to see what other scientists already have discovered. The thesis is mainly dealing with graphs and the analysis of its structure. This field of science is not really new since a lot of people have already occupied themselves with issues of networks and their compositions. For that reason, a look on existing social network analysis tools is given, since most of them already implement the basic social network analysis methods which will also be implemented in this thesis. More detailed information about the abilities of these tools and their respective benefits and drawbacks for the implementation in this thesis can be found in chapter 3.
Additionally, the focus also lies on other interesting concepts of social network analysis.

2.1 Existing Tools

The software industry already produced quite a number of different programs and applications to do social network analysis and graph visualization. Hence, it is important to get to know how these applications work and perform social network analysis and to compare that with the way it should be implemented for Signal/Collect in this thesis.
Usually, these existing tools are able to compute the well-known basic methods, such as degree centrality or local cluster coefficient and do not cover advanced methods like label propagation, random walks or hierarchical clustering.
Existing tools are mostly able to work with either directed or undirected graphs whereas Signal/Collect is designed to work with directed graphs only. Nevertheless, Signal/Collect can model an undirected graph by just adding directed edges in both directions between two vertices. This has to be kept in mind when working with Signal/Collect. Usually the existing graph tools are not constructed for distributed computing. Therefore, for these tools it is enough to hold an instance of a graph and iterate then through its components in order to calculate some social network analysis method value.

Whilst Signal/Collect is able to do distributed computing, since this is its main goal [Stutz et al., 2010] and it is not only designed to do social network analysis, it is more complex to set up the whole environment in which the social network analysis method should be calculated. Moreover, Signal/Collect is designed to assign computation tasks to the vertices and therefore, not a single instance is able to do the whole computation on its own. All this makes the computation task more complex in order to calculate the desired values, but it should be possible to benefit from that when it comes to calculation on really large graphs, since Signal/Collect executes that computation distributed on the network's vertices.

2.2 Random Walks

A highly discussed and researched topic in network analysis nowadays is the detection of communities. Various methods and indicators exist which claim to detect some sort of community. One of them is the concept of random walks. A random walk is essentially a path through the vertices of a network that at each vertex randomly decides which outgoing edge it chooses to proceed its way. It is said that a random walker usually stays longer in a community and in that way communities should be discovered [Fortunato, 2010].
A specific work on random walks was done by Cooper et al [Cooper et al., 2013]. They actually did not intend to find communities of a graph, but they aimed to do estimations of network properties by using random walks in a network. The estimations were used for properties which are also discussed in this thesis. They claim that it is neither feasible to look at the whole graph nor is random sampling of a certain part of the graph for the purpose of finding the graphs global properties.
Unlike Signal/Collect, their work is intended for undirected graphs. For these estimations, they use the so-called "first return times" in order to estimate the desired properties. This value indicates how long it takes for a random walk that started at vertex v to return back to it. Based on this information, it should be possible to compute the network properties.
The indicator they use is different for every single vertex, which means that every vertex has to compute the value for itself which makes this concept interesting to use with Signal/Collect since it is also a framework that focuses on running code on single vertices (which are distributed) and it may be practical to apply random walks for some sort of research on Signal/Collect. Nonetheless, it should be kept in mind that the described first return time indicator is designed to be used in undirected graphs.

Beyond that, it could be imagined that random walks could be used differently with Signal/Collect and some sort of measure or concept that is intended to work with directed graphs.

2.3 Label Propagation

The concept of label propagation is a mechanism that may be used to find communities, too. It works by giving some (or even all) vertices a label and recording then how these labels spread in a network. It can then be assumed that all vertices with the same label belong together in some sort of community [Fortunato, 2010]. Like for random walks, there exists some work that is concerned with a similar topic.
Glenn Lawyer designed a measure called expected force (ExF) that should indicate the spreading power a vertex in a network has [Lawyer, 2014]. He looks at it as an extension to existing measures like degree centrality or eigenvalue centrality (PageRank) which indicate how important a vertex in a network may be in general, but do not reveal information about how crucial a vertex is when it comes to its spreading power.
The power of a node in a network is of particular interest when it comes to the analysis of how an epidemic may spread in a network. Whereas label propagation is concerned with what label may become the dominating one in a network (since there may exist multiple), expected force is more interested in which vertices in the network are responsible that the label was able to be spread across the network. Basic label propagation just works by every vertex picking the label that occurs the most at its respective incoming edges (or one at random if there exist multiple), whereas Lawyer's expected force has features like a vertex's immunity to a certain label or a vertex's ability to recover from a label and to take on another one. Obviously this is already a sophisticated implementation of label propagation which would be really complex to adapt with Signal/Collect. Therefore, it makes sense to implement a simpler implementation in order to see how well it performs. Later, a more complex solution or application can be searched and possibly implemented on the Signal/Collect framework.

3

Evaluation of Tools

The first task of this thesis covers the evaluation of existing network visualization tools. This is done to benefit from an existing tool's ability to visualize graphs. Since there is a variation of such tools, all with different abilities, the choice for an existing tool on which the implementation of social network analysis methods in Signal/Collect should be built upon had to be done before the actual implementation. Hence, the tool of choice has to fulfil some criteria in order to be suitable.

First of all, the tool obviously must be able to plot the graph of choice and if possible it should support a decent number of file formats. It should have as few restrictions as possible in terms of visualization.

Preferably, the tool should have an interface or some other connection to the Java world. Since Signal/Collect is implemented in the Scala programming language, developed at the EPFL [Odersky et al., 2004] , which runs in the Java Virtual Machine, it would have crucial advantages and would limit possible problems upfront if the evaluated program already supported Java.

Furthermore, the implemented code of this thesis is about to be released under the Gnu General Public License [Stallman, 2007] and for that reason, the tool of choice should comply with that license as well.

When developing new software that uses existing libraries, it is always helpful when there is an existing community which is supporting users of that library and also acts on a regular basis. Therefore, the amount of support that is available online should be assessed in order to solve appearing problems without losing time.

Last but not least, the tool should have the possibility to be easily extended. It should provide a possibility to develop plugins or something similar.

3.1 Gephi

The first tool to be evaluated is called Gephi. It is an open source software, initially developed by the University of Technology of Compiègne and now run by a consortium in France. It is an award winning tool and supports large graphs with its built-in OpenGL architecture. Furthermore it is built upon the Netbeans platform and a very modular tool [Bastian et al., 2009].
Since being implemented on the Netbeans platform, the tool is all written in Java Code. Moreover, the software is released under the Gnu GPL 3 license. The online platform has also a rich development section with an extensive plugin center, active forums, mailing lists and a wiki with manuals as well. When using the tool, the modularity is ubiquitous. Modules can be added, moved or deleted in a workspace to fit every user's needs.

3.2 Igraph

Igraph is a network analysis package that is distributed in three different programming languages, namely C, Python and R [Csárdi and Nepusz, 2006]. It comes with a large collection of network analysis tools and possibilities of graph visualization as well. Igraph is open source, free to use and is also released under GPL 3.
For instance, igraph can be used with the statistical computing environment R to use the implemented functions of the package. Essentially, when a function is added to the library, it can simply be called by a suitable visualization and calculation program. As already mentioned, Igraph has no connection to Java, so if Igraph is chosen, a library that is able to make a connection between these programming languages needs to be found.

3.3 Graphviz

Graphviz, developed by the AT & T labs and released under the Eclipse Public License is mainly built for visualization [Gansner and North, 2000]. It uses a DOT file and creates a visualized graph. Graphviz, like Igraph, would need some kind of connector between Java/Scala and .dot or Graphviz, respectively.

3.4 Tulip

Tulip is a tool that is written in the C++ programming language and represents a framework that can be used to develop algorithms, visual encodings, data models and specific visualizations [Auber, 2003]. It can be extended by writing a plugin in Python. Even a whole integrated development environment (IDE) is provided. Tulip supplies the developer with a support platform that contains a forum, mailing lists, a plugin center, tutorials and is released under GPL 3 as well. Tulip is a product of the Laboratoire Bordelais de recherche en informatique at the University of Bordeaux in France.

3.5 Other Tools

There are other tools available, which would have been suitable for this project. But they were not considered for the evaluation, since they completely missed one or more of the required criteria. For instance, they were not open source or would only work on a single operating system.

3.6 Decision

After some testing and using of the described tools and evaluating which one fits best to the mentioned criteria, a decision had to be made.
The tools described in detail are all released under some open source license, so they all meet the condition of being open source and complying with the license under which the software to be developed is released.
Graphviz clearly failed to meet the requirements right from the beginning as it is only able to visualize graphs and it does not seem to be extensible by some plugin. It also has no reasonable connection to Java and it was expected that it would be a complex task to establish this connection which is not the goal of this thesis.
The mentioned point is also applicable for all the discussed tools except Gephi. It was not known upfront how easy it would be to build this connection between different programming languages. There actually exist programs (e.g. Apache Thrift [Agarwal et al., 2007]) which would make this task easier to complete.
However, since Gephi already provides the possibility to develop own program parts and to easily reuse the existing program, it should be considered a major advantage Gephi has in comparison with the other tools. The evaluation so far let Gephi really stand out compared with the other tools and it consolidated this position even more by having a

solid support and active community to help. The developer section also presents itself as easy-to-use and has some step-by-step instructions.
At the point where the decision had to be made, the choice was clearly made in favour of gephi. The evaluation revealed no clear drawback for this tool where all others have a lack of meeting some of the set conditions.
Table 3.6 shows the evaluation in a more detailed way.

Name	License	Connection to Signal/Collect	Community/Support	Other
Gephi	GPL	Java Plugin	Great Support, active Community	Plugin Center for Developers
Igraph	GPL	Extend Python class, use Apache Thrift	Good documentation, Community is rather small	
Tulip	LGPL	Write Python class, use Apache Thrift	Great Support, extensive documentation	
Graphviz	GPL	Not possible	Mailing List and Forum	Can mainly be used for visualization

Table 3.1: Evaluation of existing tools

4

Implementation

This part of the thesis covers the implementation of the Social Network Analysis Methods in Signal/Collect.
It discusses an overview of Signal/Collect followed by the description of standard methods (centrality measures and others), namely Degree Centrality, PageRank (or Eigenvector Centrality), Betweenness Centrality, Closeness Centrality, the Local Cluster Coefficient and the so-called Triad Census.
The computation of network properties, such as the size of a network, the network's density, the diameter, reciprocity and distributions of degree and local cluster coefficients is covered after that, followed by the last section, which treats an advanced Social Network Analysis Method that is called Label Propagation.
It is described what these measures or properties state, how they are implemented in the Signal/Collect programming model and what problems had to be solved while implementing these measures.

4.1 Signal/Collect

The implemented software of this thesis builds on the programming model named Signal/Collect. It was and still is being developed at the University of Zurich and is a programming model for synchronous and asynchronous graph algorithms [Stutz et al., 2010]. Signal/Collect aims to process large data amounts efficiently by working distributed on multiple machines. The idea of Signal/Collect is that vertices of a graph act as the computational part that interact with signals which are passed along the edges. That is also where the name of the programming model originates from. A vertex collects its incoming signals and performs some computation with them. Based on the result of that computation, new signals are sent along the outgoing edges.
Formally, a Signal/Collect graph is a directed graph, which means that every edge has

a source vertex and a target vertex which indicates the edge's direction. A vertex instance holds information about its incoming and outgoing edges and neighbour vertices respectively. An undirected graph may be modelled by adding edges in both directions between two vertices.
The benefit of using the vertices as the computational units is that a complex and time-consuming computation task can be distributed on the different vertices which allows parallel computing. This parallelization of computation also has some drawbacks. It is for instance harder with Signal/collect to coordinate different vertices or to get information of the network as a whole instantly.

4.2 Standard Methods

The standard methods of Social Network Analysis try to answer a few important questions about a network such as which is the most central or the most influental part of a network. Hence, most of its measures are concerned with centrality [Newman, 2008].
Additionally, these basic methods also try to reveal in what subgraphs a graph can be broken down and finally they want to show what kind of triads (group of three nodes) are most common.

4.2.1 Degree Centrality

The degree of a vertex in a network is defined as the number of edges attached to it. In case a graph is a directed graph, it is also possible to distinguish between indegree (nr. of edges pointing to that vertex) and outdegree (nr. of edges that have the current vertex as source). To calculate the degree of a vertex v in a directed graph, one can simply add up its indegree and outdegree.

$$Degree_v = InDegree_v + OutDegree_v \tag{4.1}$$

Since Signal/Collect is intended to be used on directed graphs, the degree centrality has to be computed as in equation 4.1. Each vertex does this by itself, counting all edges coming in and going out and adding them up to the degree centrality. Consequently, a vertex with a high degree centrality can be said to be more important in a network.
In order to get the average of the degree centrality values (which is required for all centrality measures), a desirably efficient way had to be found to calculate it. The solution for this issue was to create a special vertex which is responsible to gather the

degree centrality values from all other vertices and calculate the average of them. For that reason, along with the special vertex, a special edge was created, dedicated to pass the degree centrality values to the average vertex. So, the graph is extended by this average vertex and average edges are added between the average vertex and every other vertex (in both directions). In that way, the benefits of Signal/Collect and distributed computing can be used and the average can be computed while the graph algorithm is still running and has not to be done after that. Attention has to be paid when calculating the degree centrality values on the other edges. They must not consider the edges that are connected with the average vertex since they do not belong to the actual graph.

4.2.2 PageRank (Eigenvector Centrality)

PageRank, invented by the founders of Google, Lawrence Page and Sergey Brin, is a measure that represents the importance of a node in a network [Page et al., 1998]. Depending on which vertices point to a certain vertex, this vertex computes its own PageRank value according to the incoming values. The higher the incoming values are, the more important a vertex itself is and every other vertex that this one points at with an outgoing edge gets more important as well. To avoid that every vertex is graded very important, a so-called damping factor is introduced that normalizes the PageRank values of the vertices. The equation for the computation of PageRank for a vertex v is then:

$$PageRank_v = 1 - d + d * (\sum_{i=1}^{n} \frac{PageRank_i * w_i}{\sum_{j=1}^{m} w_j}) \tag{4.2}$$

where d is the damping factor, n is the number of incoming edges, m is the number of outgoing edges of the current source vertex of the current incoming edge and w is the weight the current edge has. The damping factor is usually set to 0.85, since it proved to be the most suitable one in various experiments [Brin and Page, 1998].
Similar to degree centrality, a way had to be found to calculate the average without affecting the rest of the actual graph. The solution in this case was the same as described in section 4.2.1.

4.2.3 Betweenness Centrality

Betweenness centrality deals with how central a vertex in a network is. The concept of shortest paths is therefore of interest. A path is simply a sequence of vertices which connect vertex v and vertex w. Hence, the shortest path is the path which passes the smallest number of vertices between vertices v and w . Betweenness centrality then is

the number of shortest paths that pass vertex v divided by the number of shortest paths in the whole network. The resulting number contains information about the control vertex v over the flow of information in a network has.[Newman, 2008].

$$Betweenness_v = \frac{sp_v}{sp} \tag{4.3}$$

where sp_v is the number of shortest paths going through vertex v and sp is the number of all shortest paths. A vertex in Signal/Collect only knows information about direct neighbours. It can collect information of an incoming edge and it can signal through an outgoing edge. It does not have the big picture of the whole network and does not know information about neighbours of neighbours.
Since it is desired to get knowledge about shortest paths, all shortest paths have to be gathered in order to achieve the calculation of betweenness centrality for each vertex according to formula 4.3. In order to do so, an approach to gather these shortest paths in Signal/Collect has to be found.
The proposed approach then is that every edge signals its path from the source vertex u to the target vertex v and the target vertex stores this path in a list of paths. The target vertex will then pass all paths it has stored to its outgoing edges which will then signal these paths, extended by the target vertex they point at. If a target vertex already has a path stored with the same source vertex that the incoming path has, the new path will only be added if it is shorter than the one already stored, since we are only looking for shortest paths.
After each vertex has stored all its incoming shortest paths, the calculation of betweenness is executed. Regarding formula 4.3, the number of shortest paths at a single vertex has to be divided by the number of all shortest paths and in order to achieve that, all shortest paths have to be counted first, which is done centrally after the execution of Signal/Collect.
As soon as the shortest paths have been counted, the list is passed to each vertex. Each vertex then calculates its own betweenness centrality by filtering out the relevant shortest paths for itself and dividing this number by the total number of shortest paths. The vertex then passes its betweenness value back to the central graph instance which gathers all single vertex betweenness centrality values.

4.2.4 Closeness Centrality

The concept of closeness centrality is similar to the one of betweenness centrality, since it is concerned with shortest paths as well. The resulting values gained from this algorithm

are however calculated differently. The closeness centrality of a vertex v is defined as the mean length of all shortest paths to every other vertex. Excluded from this average are the vertices that can not be reached from vertex v [Newman, 2008].
Closeness centrality of vertex v can then be defined as:

$$Closeness_v = \frac{\sum_{i=1}^{n} length(sp_i)}{n} \tag{4.4}$$

where n is the number of shortest paths attached to vertex v and sp is the current shortest path.
As stated before, closeness is also dealing with the concept of shortest paths and therefore the obvious solution is to reuse the algorithm which was introduced for betweenness centrality in section 4.2.3. For that reason, we can benefit from reusing the same Signal/-Collect algorithm to gather the shortest paths and only have to change the calculation of the measure. Every vertex gathers all incoming shortest paths and afterwards, the shortest paths are counted followed by the calculation of the closeness centrality carried out by each vertex itself. Finally, all values are gathered at the central graph instance.

4.2.5 Local Cluster Coefficient

The local cluster coefficient of a vertex v can simply be treated as the density of the local neighbourhood of vertex v (see also section 4.3.2) [Hanneman and Riddle, 2005]. That can also be expressed as "how many of my neighbours are neighbours among themselves".
A term which is related to the local cluster coefficient is community detection. The local cluster coefficient examines its immediate neighbourhood and does a statement about how well this neighbourhood is connected or how strong the community holds together.
To calculate the coefficient for a vertex v, the connections between its neighbours have to be counted and divided by the number of possible connections of these neighbours.
The number of possible connections can be expressed as:

$$p_v = n_v * (n_v - 1) \tag{4.5}$$

where p_v is the number of possible connections of vertex v and n_v is the number of neighbours of vertex v.
Expressed as a formula, the local cluster coefficient for vertex v is calculated as follows:

$$LocalClusterCoefficient_v = \frac{c_v}{p_v} \tag{4.6}$$

where c_v is the number of the connected neighbours of vertex v and p_v is the number of possible connections between these neighbours of vertex v as calculated in formula 4.5. Algorithmically, this problem is more complex to implement in Signal/Collect. Since vertices only know information about incoming edges, they are as well only able to evaluate this information. This issue is similar to the one for betweenness centrality and closeness centrality. However, the solution implemented for the local cluster coefficient is different. While running Signal/Collect, vertices collect information of their respective incoming neighbour vertices and when convergence is reached, this information of all vertices is evaluated centrally and unlike with betweenness and closeness nothing gets passed back to the vertices for further calculations.
As a first step, all vertices gather what outgoing edges each source vertex of its respective incoming edges have. Since each vertex already has its own collection of outgoing edges stored as a variable, this collection can simply be passed as a signal on all outgoing edges. After all vertices have stored this information, it gets evaluated centrally in the following way. The algorithm iterates through every vertex and then goes through its respective incoming and outgoing edges and counts how many connections exist between the source or target vertices at these edges. When counting is done, equation 4.6 is applied and the local cluster coefficient for each vertex calculated.

4.2.6 Triad Census

The idea of the triad census is based on the concept of triads. A triad is simply a combination of three vertices, no matter whether they are connected or not. When treating directed graphs (like in this thesis), there are 16 different possible connection combinations between three vertices [Batagelj and Mrvar, 2001].

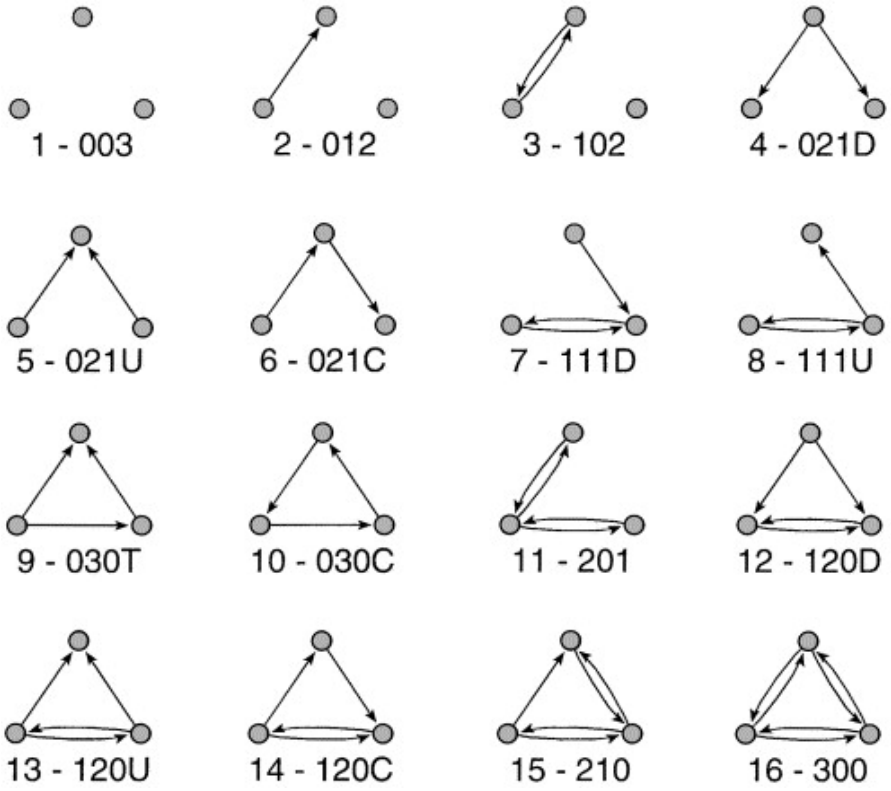


Figure 4.1: The 16 types of triad connections in directed Graphs

The codes below each type of connections between three vertices in figure 4.1 are a unique label consisting of three digits. Each digit represents a different information: digit 1 is the number of pairs of vertices connected by bidirected edges, digit 2 is the number of pairs of vertices connected by one edge and digit 3 is the number of unconnected pairs of vertices.
Some of these labels also have an extension ('D', 'U', 'C' or 'T') which help to distinguish a triad if the three-digit label is not enough. These extension letters are stating the following: D = Down, U = Up, C = Cyclic, T = Transitive.
Batagelj and Mrvar developed an algorithm to find all occurences of the different triad census types which was adapted for Signal/Collect and Scala in this thesis. The basic idea for the algorithm is to form three basic types of triads which are the null triad (003), dyadic triads (012 and 102) and connected triads (refer to figure 4.1). The procedure of the algorithm is intended to count the dyadic and connected triads first and to subtract the result of the number of all triads to get the number of null triads, such that the null triads do not have to be counted. The reason to do that is, that most triads in a

large graph are null triads and therefore it makes sense to count the other types first [Batagelj and Mrvar, 2001].
The algorithm for dyadic and connected triads proceeds by going through every possible triad, but it has to be assured that every set of vertices v, u, w is passed only once and does this by canonical selection [Batagelj and Mrvar, 2001]. The number of dyadic triads is determined first which is done by the following equation:

$$n- \mid R(u) \cup R(v) \mid -2 \tag{4.7}$$

where n is the number of vertices of the network and $R(v)$ is the set of neighbours a vertex v has. The number of dyadic triads formed by the vertices u and v is then the number of vertices in the network, leaving out the set of neighbours that are connected to either u or v and vertices u and v as well (hence, subtracted by 2 at the end).
To count the connected triads, Batagelj and Mrvar found a sophisticated method to do so. Each triad gets a code assigned, which is a number between 0 and 63. This code is found by using a 3 x 3 matrix and a code table as shown in figure 4.2.

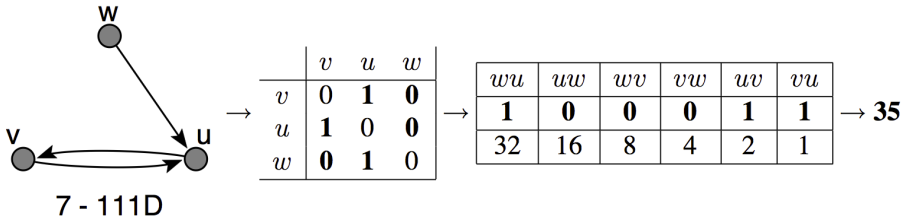


Figure 4.2: Elicitation of the triad code

The triad type as seen in figure 4.1 is then assigned by this triad code using table 4.1.

code	type	code	type	code	type	code	type	code	type	code	type	code	type	code	type
0	1	8	2	16	2	24	6	32	2	40	4	48	3	56	8
1	2	9	6	17	6	25	10	33	5	41	9	49	7	57	14
2	2	10	5	18	4	26	9	34	6	42	9	50	8	58	13
3	3	11	7	19	8	27	14	35	7	43	12	51	11	59	15
4	2	12	3	20	5	28	7	36	6	44	8	52	7	60	11
5	4	13	8	21	9	29	14	37	9	45	13	53	12	61	15
6	6	14	7	22	9	30	12	38	10	46	14	54	14	62	15
7	8	15	11	23	13	31	15	39	14	47	15	55	15	63	16

Table 4.1: Triad Types

The resulting algorithm which is adapted for Signal/Collect in Scala then looks as follows.

```
/*
 * scala SortedMap with the vertex' id as key and the actual
 * vertex object as value.
 * It is important to use a SortedMap here because the algorithm
 * needs to iterate through the vertices in ascending id order.
 */

var vertexMap = SortedMap[Int, TriadCensusVertex]()

for (vertex <- vertexMap.toMap) {
  if (vertex._2.neighbours.isEmpty) {
    for (outgoingneighbour <- vertex._2.outgoingEdges) {
      vertex._2.neighbours += outgoingneighbour._1.asInstanceOf[Int]
    }
  }
  for (neighbour <- vertex._2.neighbours) {
    if (vertex._2.id.asInstanceOf[Int] < neighbour) {
      var triadType = -1;
      val neighbourVertex = vertexMap.get(neighbour).get

      /*
       * vertices neighbouring to either the vertex or the current
       * neighbourVertex (excluding the vertex and the neighbour)
       */
      val neighboursOfBothVertices = vertex._2.neighbours union
          neighbourVertex.neighbours diff Set(vertex._1, neighbour)

      if (vertex._2.outgoingEdges.contains(neighbourVertex.id) &&
          neighbourVertex.outgoingEdges.contains(vertex._2.id)) {
        triadType = 3;
      } else {
        triadType = 2;
      }
      var countValue = countMap.get(triadType).getOrElse(0.toLong)

      countMap += ((triadType, countValue + (vertexMap.size -
          neighboursOfBothVertices.size - 2).toLong))
      for (neighbourOfBoth <- neighboursOfBothVertices) {
       if (neighbour < neighbourOfBoth ||
           (vertex._2.id.asInstanceOf[Int] < neighbourOfBoth &&
           neighbourOfBoth < neighbour &&
           !vertex._2.neighbours.contains(neighbourOfBoth))) {
          val neighbourOfBothVertex = vertexMap.get(neighbourOfBoth).get

          /*
           * The triadType is determined by the constant codeToType
           * array by using the triCode function
           */
          triadType =
              SignalCollectSNAConstants.codeToType(triCode(vertex._2,
              neighbourVertex, neighbourOfBothVertex))
```

```
          countValue = countMap.get(triadType).getOrElse(0)
          countMap += ((triadType, countValue + 1.toLong))
        }
      }
    }
  }
  var sum = 0.toLong
  for (i <- 2 to 16) {
    sum += countMap.get(i).getOrElse(0.toLong)
  }
  countMap += ((1, ((vertexMap.size.toLong * (vertexMap.size - 1).toLong
      * (vertexMap.size - 2).toLong) / 6).toLong - sum))
}
}
```

Listing 4.1: Triad Census Algorithm

In order to execute this algorithm which has the goal to analyze connections between vertices, all information about the vertices and its respective neighbours can be gathered via Signal/Collect. The way to receive the actual values for the triad census is similar to betweenness centrality, closeness centrality and the local cluster coefficient. In a first step, the vertices collect information about its own neighbours and after this is done, the measures are calculated from a central instance.

4.3 Network Property Statistics

Network properties differ from the other social network methods in a sense that they characterize the environment where the measures are calculated. That means that in this section, the attention lies more on the entire network or graph and a Signal/Collect algorithm is not always necessary to gain the needed information about a network. Nevertheless, since some of the properties also make use of already implemented functionalities (e.g shortest paths), they can be reused in an adapted way to calculate the respective property.
These properties are not only important to get some information about a graph, they can also contribute to the evaluation of the calculation of the social network methods, since they may have some influence on how fast these measures can be computed with Signal/Collect.

4.3.1 Graph Size

The size of a graph is defined as its number of nodes [Hanneman and Riddle, 2005]. Basically, counting the number of nodes would be enough. Nevertheless, with regard to the before described algorithms to calculate the social network measures, namely degree centrality (section 4.2.1) and PageRank (section 4.2.2), attention has to be paid to the special vertex and the special edges introduced to calculate the average. Each vertex labeled as an average vertex should not be added to the size of the graph, since it is not of interest.

4.3.2 Graph Density

The density (or its counterpart called sparseness) is a network property that gives information about how connected the vertices in a network are. The calculation of the density of a network is indicated as the number of edges e of graph g divided by the number of possible connections among the vertices in that graph g (the number of possible connections is described in formula 4.5), whereas n is the number of vertices in graph g [Hanneman and Riddle, 2005].

$$Density_g = \frac{e_g}{n*(n-1)} \tag{4.8}$$

When the density is higher, more signals can be sent through this network, which may have an influence on the computation time depending on the implemented algorithm.

4.3.3 Graph Diameter

A simple definition of the diameter of a graph is, that it is the longest shortest path of a network, meaning that every (reachable) vertex w can be reached in at most this number of steps, starting at vertex v [Hanneman and Riddle, 2005]. Considering shortest paths for a network's diameter also means that the already existing implementation of gathering shortest paths of a graph in Signal/Collect can be reused (refer to section 4.2.3). In detail, all shortest paths are collected first, in the mentioned Signal/Collect manner. After that, the algorithm iterates through all shortest paths and picks the longest one to determine the diameter of the network.

4.3.4 Graph Reciprocity

The reciprocity of a graph is concerned with how many paths from vertex v to vertex w also have a path in the opposite direction. The reciprocity of a graph is then defined as the fraction of paths that have a path in the opposite direction divided by the number of all shortest paths, too [Hanneman and Riddle, 2005]. The already implemented functionality of gathering all shortest paths for Betweenness centrality, Closeness centrality and the network diameter can as well be reused for the reciprocity. Consequently, the program iterates through all shortest paths and counts the ones that also have one in the other direction and finally calculates the reciprocity.

4.3.5 Degree Distribution

Two important network properties are dealing with distributions of the described social network method measures. In particular, these two are degree distribution and local cluster coefficient distribution (see next section, 4.3.6).
Degree distribution is concerned with, as the name states, how the degrees are distributed in a network. In other words, it describes how many nodes a particular degree centrality have. After getting the degree of every node, it can be visualized in a histogram which degree centrality values appear most often, which degree centrality is the highest in this network and so on.
This may reveal some valuable information about a network. For example, the degree distribution can show that the network is scale-free, which means that the distribution follows a power law and has an asymptotic behaviour. It has been shown that a lot of networks are scale-free which makes it an important concept [Barabási et al., 2000].
It also may occur that the degree distribution shows that the network is a Small-world

network, meaning that the vertices in a network can reach any other vertex with a low number of steps. In other words, the network has a low diameter and a short average path length. A popular model of small-world network was developed by Watts and Storgatz [Watts and Strogatz, 1998].
Implementation-wise, the Signal/Collect functionality to calculate the degree centralities (as described in section 4.2.1) can be reused here. The algorithm for degree centrality can be run with the current graph and after that is done, the degrees can be stored in a data structure to be visualized later.

4.3.6 Local Cluster Coefficient Distribution

The calculation of the cluster distribution of a graph works very similar like the one for degree distribution. The main and obvious difference is, that it gathers local cluster coefficient values rather than degree centrality values and makes use of the existing implementation of the local cluster coefficient. Everything else behaves the same as with degree distribution.
The local cluster coefficient distribution may be utilized when dealing with network effects. For example, when there exists a large proportion of high local cluster coefficients, it can be valuable to make use of that. Popular examples are business clusters, such as the silicon valley where a lot of technological companies are located. Such clusters are assumed to increase productivity and competitiveness of a cluster [Porter, 2000] and therefore it can be beneficial for a company to move there as well.

4.4 Advanced SNA Methods

Besides the already discussed standard social network analysis methods in section 4.2, another task for this thesis was to implement a more advanced method, which is also considered to be suitable to be implemented with Signal/Collect. The decision thereby fell on Label Propagation which is described in the following subsection.

4.4.1 Label Propagation

There are different examples of how label propagation can appear in real life. One of them for instance is the spreading of a virus or disease (the label in this context) in a social network of humans. It can be imagined that one person in a network has this virus and with label propagation, its spreading (or its disappearance) can be observed.

Another one is the so-called network effect. Usually, there is a product (the label in this context) which has a higher utility when other people also have this product (think of mobile communication applications) and people in a social network would rather use such a product when a friend or colleague uses the same. In that way, the label propagates through a network. In either case, observations of communities can be made since all vertices in a graph that have the same label can be considered as a community.
Label propagation is implemented for Signal/Collect in the following way.
The graph is set up with each vertex getting a unique initial label (e.g. its ID). When execution starts, each edge signals the label of its source vertex to its target vertex. The target vertex then collects all these labels and picks the one which occurs the most, or if there are multiple labels having the same number of occurences, a label is picked randomly to be the new label of the target vertex (The label has not necessarily be picked randomly, different approaches may be applicable depending on the context.). After a certain threshold of number of steps, some labels may dominate in a network whilst other labels disappear and even convergence may take place. This way of using label propagation is adapted from Fortunato, as presented in his paper on community detection in graphs [Fortunato, 2010].
The aim of the implementation is to let the user set how many steps of label propagation he wants to execute and afterwards he gets a graphical overview how the distribution of labels developed over this chosen number of steps. In that way, convergence does not have to take place necessarily but a user can see how the distribution of label develops over the chosen number of steps and which labels may already be dominant or not. Signal/Collect proves to be suitable to implement label propagation since it is a very flexible tool and it is possible to set how many steps should be executed. Setting the steps is desirable in order to compare different number of steps and see how the labels propagate after different number of steps and for that reason, Signal/Collect is a beneficial tool to use in order to implement Label Propagation. Even more, Signal/Collect's design of using directed graphs is also well-fitting for label propagation as it is used here. In different contexts where it may be desirable to adapt label propagation to other needs, it is also an easy task to change the algorithm slightly (e.g. picking the label).

4.5 Software Architecture

This section describes the structure of the implemented software. The first part is concerned with how the core of the software (the implementation of the Social Network Analysis methods) is segmented, particularly which piece is responsible for the calcula-

tion of the social network measures and network properties and how the recorded results are packaged and stored. The second subsection focuses on the integration into the Gephi platform. It describes how the interfaces look like and how the plugin is structured.

4.5.1 Architecture of the SNA Module Implementation

The implementation concerned with the different social network analysis methods, the network properties and label propagation, is written in Scala. Because the implementation is using the Signal/Collect programming model, which is written in Scala too, this is a natural choice. Java would be an alternative because Signal/Collect has a Java API. The structure of the main module is partitioned in one main package and two sub packages. Whereas the first sub package is concerned with the Signal/Collect algorithm implementations of the social network analysis methods, the second sub package deals with parsing the GML Files into Signal/Collect graphs, vertices and edges. The main package then is concerned with general tasks that are used in every graph, such as calculating network properties and objects that are responsible to package the results and send them to the interfaces which then take care to send them further to the user. Figure 4.3 shows a diagram of this core software.

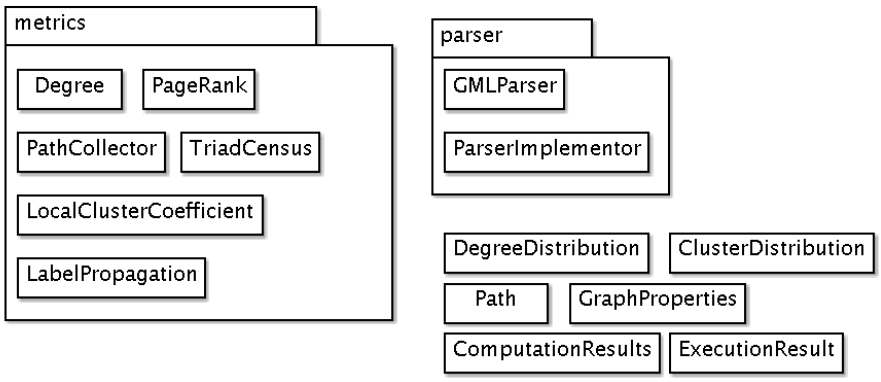


Figure 4.3: The architecture of the core implementation. The two packages “metrics” and “parser” represent the two sub packages whereas the whole diagram is showing the main package in which everything is contained

4.5.2 Connection to Gephi

Gephi is written in Java. Signal/Collect is written in Scala. Luckily, they both work on the Java Virtual Machine [Lindholm and Yellin, 1999]. That keeps the risk of running into problems low when trying to build a connection between these two parts of the implementation.
The concept of the connection module is to have some interfaces on the Signal/Collect side, which can be accessed from the Gephi plugin. In particular, every social network analysis method should get its own interface in order to achieve some sort of modularity and separation of concerns. The diagram in figure 4.4 shows the described architecture graphically with an UML class diagram (some details are omitted, the focus is set on the main overview of the model).

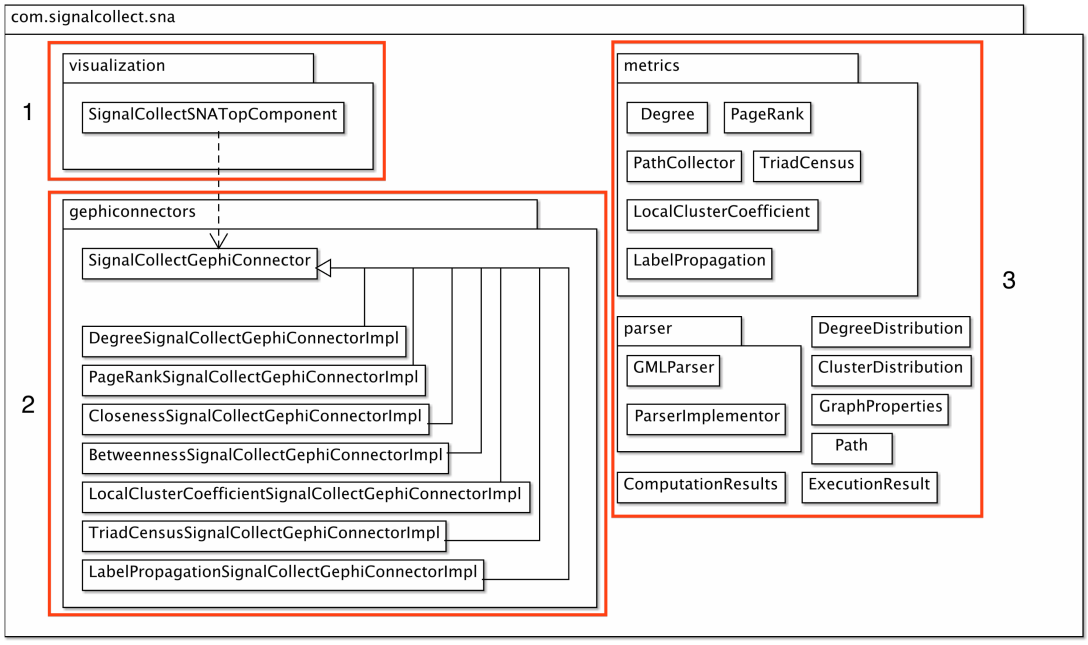


Figure 4.4: The architecture of the whole program, including the connection to Gephi. The different modules are assigned with numbers and mean the following:

1. The Gephi module. It accesses Signal/Collect via the interfaces in module 2.
2. The interfaces for outside access. They provide functions to do Social Network Analysis with Signal/Collect.
3. The Social Network Analysis module that contains all implementations described in this chapter.

4.6 User Interface

The screenshot in figure 4.5 shows the user interface of the plugin that is implemented in Gephi. It consists of five different modules with different responsibilities. One module implements a file chooser in order to let the user choose his desired GML-file to be used. The others are all concerned with Social Network Analysis. One runs the metrics, another gathers the degree and local cluster coefficient distributions, one displays the properties and the last is dealing with label propagation.

Figure 4.5: Screen shot of the user interface. The five different modules are concerned with the following:

1. The file chooser. A user can choose a GML-file from his machine to use it for Signal/Collect SNA
2. SNA module in which a user can calculate the metrics for the chosen graph.
3. Text area to show the properties of the graph
4. Show charts of the degree distribution and cluster distribution
5. Part to execute label propagation with the possibility to set the number of signal and collect steps

5

Evaluation

In order to assess how well the software performs and to compare the different parts of the implementation, an evaluation is performed. This chapter splits into four parts. The first part discusses the evaluation of the algorithm runtimes of the different implemented social network analysis methods whilst section two tries to analyse this runtime evaluation. Section three describes the evaluation for correctness of the software (Testing of the software) and part four assesses general benefits and drawbacks of using Signal/Collect for this thesis.

5.1 Runtime Evaluation

Having efficient software is vital and therefore testing the software developed for this thesis for efficiency and computation time is a relevant task. The evaluation was done with graphs of different sizes and different properties. The computation time of Signal/-Collect was measured, evaluated and then put into a graphic representation of it.
The presented histograms and charts are showing the runtimes against the number of nodes (the size) of a network.
For the evaluation different graph files of real world networks were used. In appendix A these graphs and their properties are listed. The files are all in the GML format.
The runtime evaluation process for the different graph files and social network analysis methods consisted of 10 runs through each graph with each social network analysis method. Ten runs are considered to be enough to obtain a statistically sufficient runtime value with the average over all these runs, since unexpected influences on the runtime and other effects of outliers can be minimized.

Setup of the Evaluation Environment

The evaluation was executed on a cluster of machines, provided by the University of Zurich. This cluster features 12 machines, each possessing 24 processors. A code project

dedicated for the evaluation was developed and then uploaded to this cluster where the evaluation could be run.
The evaluation itself was executed on three of these machines, using 23 processors and 80 GB of memory on each machine. The runtime evaluation contains the time used for executing the algorithm in Signal/Collect and, if needed, the calculation of the desired measure. Parsing of the graph and packaging of the results is not included in the evaluated times.

5.1.1 Basic Implementations

The first part of the runtime evaluation discusses the runtime evaluation of social network methods that have implementations which only make use of a single Signal/Collect run, meaning that after this first Signal/Collect execution, the single vertices are done with their work. In particular, the evaluation of degree centrality, PageRank and the local cluster coefficient is covered in this section.

Degree Centrality

The runtime evaluation of degree centrality by doing 10 calculation runs of the different graphs can be seen in figure 5.1.

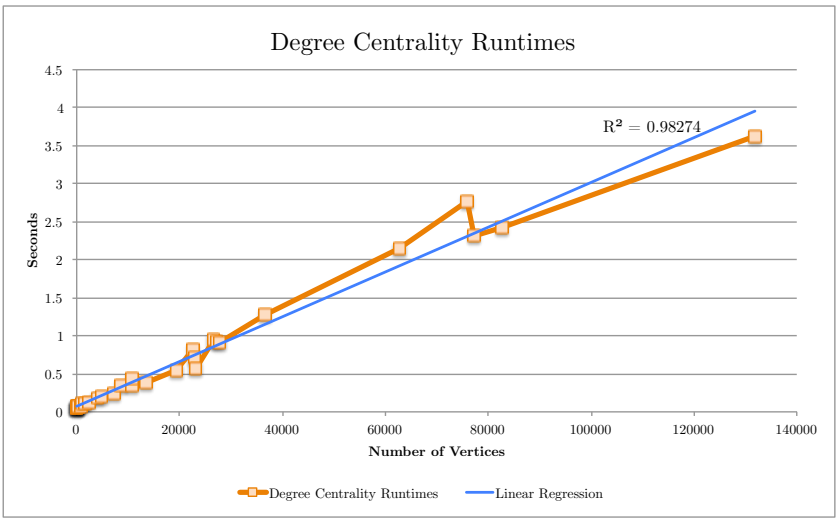


Figure 5.1: Evaluation result of the degree centrality runtimes

Regarding the graph of the evaluation, a linear behaviour (with some minor outliers)

of the runtimes in comparison to the number of vertices in a graph can be observed. A regression analysis (indicated with the blue trendline) proves that the result of this evaluation is relatively close to linearity, since its R^2 value is close to 1, which means that the data points lie close to the linear function. Therefore it can be assumed that the calculation time of degree centrality depends on the number of vertices in a graph and that other properties play a subsidiary role considering the calculation.
Regarding how degree centrality is implemented, it can be assumed that this linear behaviour should be the case since each vertex just counts its outgoing and incoming edges which usually does not take a lot of signal steps and therefore the difference of the calculation mainly consists of gathering all the vertex values and that depends particularly on the number of vertices in a graph.

PageRank

Figure 5.2 shows the trend of the runtimes for the PageRank algorithm in Signal/Collect.

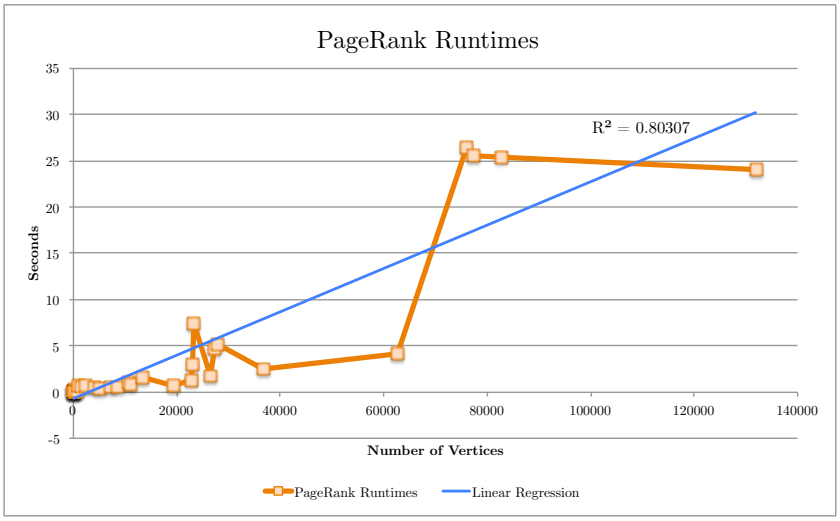


Figure 5.2: Evaluation result of the PageRank runtimes

Compared to Degree centrality, PageRank does not have such a clear pattern of the runtimes dependent on the number of vertices. This is also shown by the linear regression or R^2 value of 0.8, which means that not the whole curve could be described with linear regression. Consequently, different factors must have some effect on the runtimes. Nevertheless, the number of vertices seems to be a factor that influences the runtime of the PageRank calculation, but it is not the only dominant one. Considering how PageRank

is implemented, an indication of why the runtime of the calculation has higher variation than with degree centrality can be seen. The state of a vertex has the possibility to change more often because the collect function does not work with a fixed number like degree centrality, but rather the PageRank value of the source vertex of the incoming edge. Hence, the probability that it takes longer for the algorithm to converge is higher than with degree centrality which just gathers the number of incoming edges when collecting. Subsection 5.2.1 tries to find a more profound explanation for the result of this evaluation by assessing how the network structure may have influenced the runtimes of PageRank.

Local Cluster Coefficient

Figure 5.3 shows the recorded runtimes of the local cluster coefficient evaluation on the different graphs. Compared to degree centrality and PageRank, not all graphs could have been evaluated, since the evaluating machine ran into out of memory errors, which indicates that after a certain threshold of graph size, it is not possible to run the implemented algorithm anymore.

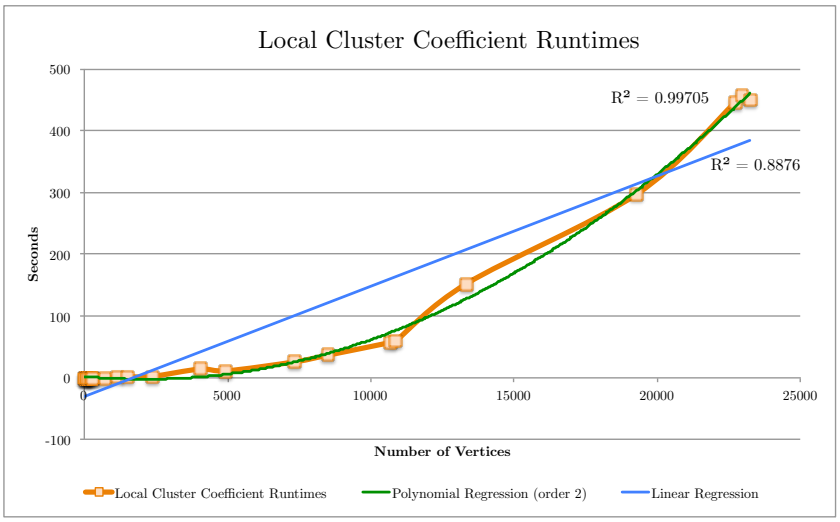


Figure 5.3: Evaluation result of the Local Cluster Coefficient runtimes

The local cluster coefficient evaluation has recorded runtimes which seem to be influenced by the number of vertices, since some dependency of the runtimes on the number of vertices can be observed. In fact, the runtimes show that they are close to a polynomial curve (of order 2) with a polynomial regression value R^2 of 0.99. Moreover, the

plot shows that the recorded values are rather polynomial than linear. Comparing the implementation of local cluster coefficient to the one of degree centrality (which showed a linear behaviour) may explain this polynomial behaviour. It can be seen that the algorithm of the local cluster coefficient has to iterate through all vertices and then as well through all neighbours of each vertex in order to calculate its local cluster coefficient whereas degree centrality only has to count the neighbours and does not do anything further with them.

Comparison of the basic implementations

Considering figures 5.1, 5.2 and 5.3, one big difference exists between the runtimes among these social network analysis methods. The local cluster coefficient has apparently significantly higher runtimes than the other two for the same graphs. This can possibly be answered with the evaluation itself where the local cluster coefficient showed a polynomial behaviour and the other two were closer to a linear behaviour. Usually, a polynomial curve ascends faster than a linear curve and the higher the x-value gets, the bigger is the difference between the y-values of a linear and a polynomial curve.

5.1.2 Implementations using Shortest Paths

The evaluation of betweenness centrality and closeness centrality is separated from the other social network methods since the calculation of them is more complex and hence, it can be assumed that it is more time consuming as well. Since both methods use the same algorithm for gathering shortest paths, they are displayed both in figure 5.4.

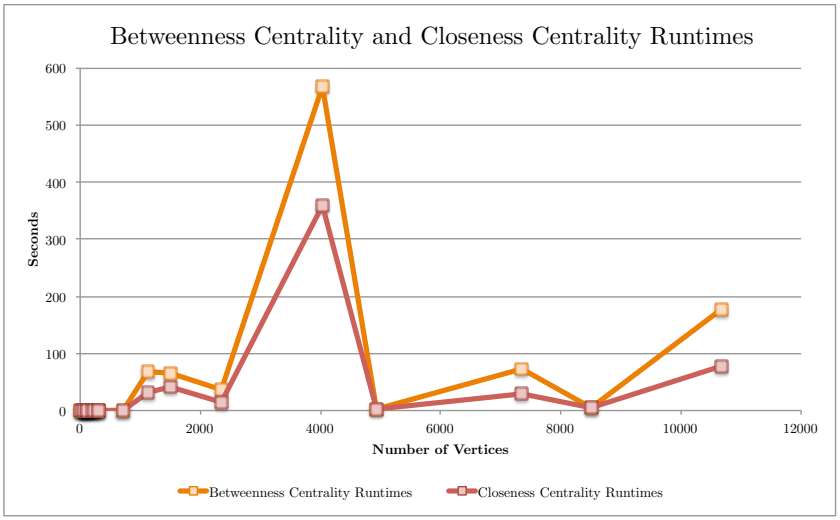


Figure 5.4: Evaluation result of Betweenness centrality compared to Closeness centrality runtimes

Closeness centrality

Figure 5.4 shows a behaviour of closeness centrality runtimes which does not allow an explanation only using the size of the different graphs. There obviously is a high variation of runtimes with different graph sizes and no dependency of the size can be interpreted. Furthermore, different regressions also showed relatively low R^2 values, which do not allow to draw satisfying conclusions of it.
Hence, other reasons for this behaviour of the runtimes have to be found. Section 5.2 tries to find what may also have an impact on the runtime of closeness centrality (and other measures which have a similar behaviour as well).

Betweenness centrality

The same observation that was made for closeness centrality can be applied to betweenness centrality. There seems to be a significant variation between the runtimes with

growing size of the graph. The next subsection shows a comparison between these two social network analysis methods in order to emphasize that closeness centrality and betweenness centrality show similar results.

Closeness centrality compared to Betweenness centrality

In figure 5.4, it can be seen that the two curves are progressing very similarly and have high or low runtimes with the same graphs. The explanation for that may be found in the implemented algorithm. Both these measures let the vertices gather their respective shortest paths first, pass them back to the central graph instance and then calculate their values. Mathematically, the correlation between the runtimes of closeness centrality and betweenness centrality lies with 0.995 very close to one, which is nearly perfect correlation.
It can also be seen that betweenness usually has a higher calculation time than closeness. Another look at how the corresponding vertex values are calculated shows that betweenness centrality has a more complex way to filter out the paths that go through a vertex from the path list and this presumably takes more time than filtering the paths which start at a certain vertex.

Why only the smaller graphs were evaluated

In contrast to the evaluation of degree centrality and PageRank (see section 5.1.1), the plots of betweenness centrality, closeness centrality and local cluster coefficient cover a smaller number of graphs. The largest graph that was evaluated only has about 12000 (25000 with the local cluster coefficient) vertices compared to the ones of degree centrality or PageRank which were able to run significantly larger graphs. When trying to run larger Signal/Collect graphs with the implemented algorithms for closeness and betweenness, the program failed to execute the algorithm to the end because it suffered from using too much memory on the machine. This was the reason for the program to throw an exception and stopping the execution.
A closer look at how the calculation of these social network methods works shows that the collection of *all* shortest paths has to be a task that takes a long time and even longer with every other collect step. It seems logical that a graph with a larger size tends to have a higher number of shortest paths and the more shortest paths exist the longer it takes for each vertex lying on various shortest paths to elicit and store all of them.
Of course, this all depends not only on the size of the graph but also on how this graph

is connected. Imagine two graphs with the same number of vertices but one has a significantly higher number of edges. Under normal circumstances, the graph with the higher number of edges must have a much higher number of shortest paths and for that reason it also has to take longer to gather all of them and afterwards calculate closeness centrality or betweenness centrality.

5.1.3 Triad Census

The figure displayed below shows the runtime evaluation of the implementation of the triad census.

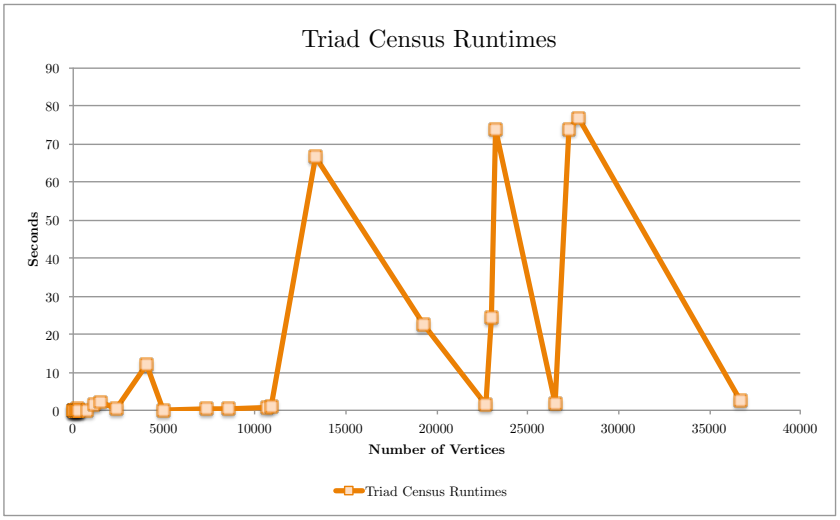


Figure 5.5: Evaluation result of the Triad Census runtimes

The same observation as for betweenness centrality and closeness centrality applies to the evaluation of the triad census algorithm. A high runtime is not necessarily dependent on the size of a graph. Indeed, a high volatility of runtimes can be seen and different reasons for that have to be taken into account as well. Unlike with closeness centrality and betweenness centrality, the reason why the algorithm behaves in such an inconsistent way with varying graph size should not only be sought in the structure of the graph but also in the way the triad census types are determined (see also the algorithm in listing 4.1). Subsection 5.2.4 discusses in detail what triad census type or the distribution of the census types may have had an influence on the runtime of a certain graph.

5.1.4 Label Propagation

The performance evaluation for the label propagation algorithm differs from the evaluation of the other social network analysis methods since the number of signal and collect steps can be determined by the user and the algorithm stops as soon as this number of steps is reached and does not continue until convergence is reached as in all the other cases.
Hence, it is worthwile to see how the chosen number of steps influences the runtime of the algorithm. Since the collect algorithm for label propagation is rather simple anyway (it only collects the incoming labels and chooses one without complex calculations), other impacts are discussed in the section 5.2.
The evaluation of label propagation also does not consider whether convergence is reached or not, it focuses on the influence of the number of signal and collect steps.

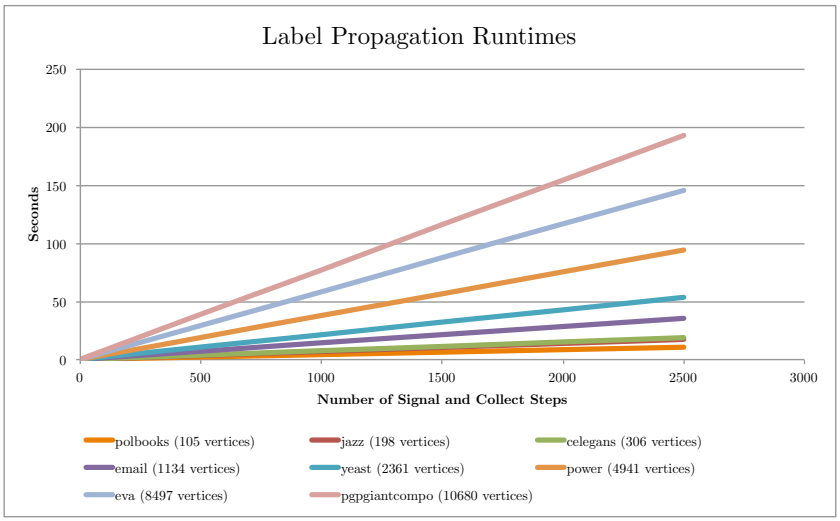


Figure 5.6: Evaluation result of the Label Propagation runtimes

The evaluation was done with 8 different graphs of different sizes (number of vertices in braces). As it can be seen in figure 5.6, all graphs behaved linearly with growing number of signal and collect steps. The graphs with a larger number of vertices also have a higher computation time with the same number of steps.
As a conclusion, it can therefore be said that label propagation, as implemented in this project, is linearly dependent on the size of the graph and the number of steps the user chooses. Other impacts of the environment do not appear to have a significant impact.

5.2 Analysis of the Results

After having seen how the different social network analysis method runtimes behave over different graph sizes, in this section it is assessed what may also have influenced the described results. This is also being underlined with some illustrating data taken from the actual results or with an additional plot in order to demonstrate that the proposed effects may really have an impact on the calculation time of these algorithms.

5.2.1 Influence of the Network Structure on PageRank

Degree centrality and local cluster coefficient showed runtime results in section 5.1, which were also underlined with regression (or R^2) values, that support an explanation with linear or polynomial models. On the other hand, PageRank's evaluation showed results which can not be interpreted clearly. Therefore, an additional evaluation was performed with different graphs in order to show that the behaviour of the implemented algorithm is not that volatile. For that reason, 35 synthetic graphs were generated by a python web graph generator. These graphs are all power law graphs (or scale-free [Barabási et al., 2000]) and therefore all have the same structure. The evaluation of these synthetic graphs compared with the real-world graphs is shown in figure 5.7.

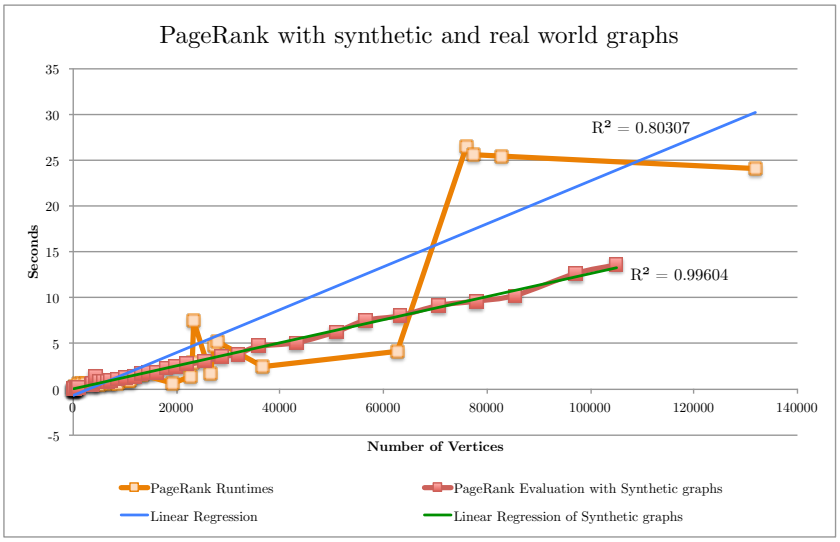


Figure 5.7: Evaluation result of the PageRank (real-world and synthetic graphs) runtimes

As the plot shows, the synthetic graphs have evaluation results which apparently

indicate a linear dependency of the runtime on the graph size with its R^2 value that is very close to 1. Therefore, it can be assumed that the varying network structures of the real world graphs have had an influence on the runtimes which led to a rather low linear regression value.

5.2.2 Influence of Edge to Node Ratio on Path Collection

As seen in section 5.1.2 the runtimes for betweenness centrality and closeness centrality are highly volatile among different graphs and do not not have a clear correlation to the number of nodes. Since both SNA methods use the same implementation in Signal/Collect, this analysis section can be generalized for the path collection
Different possible reasons for the observed volatility of runtimes had to be found. Considering the recorded runtime values, one cause could be the ratio of edges to nodes. Assuming that on average, each vertex has more edges attached to it when this ratio is higher and keeping in mind that the mentioned centrality measures are doing a lot more signal and collect steps than for instance degree centrality in order to gather shortest paths, it can be reasoned that the number of edges compared to the number of vertices could have an impact on the calculation time.
Of course, this is only a part of the explanation of the observed runtimes on some graphs having a high number of edges but the same number of vertices. Nevertheless, table 5.1 tries to demonstrate this impact by doing an exemplary comparison of three graphs and their runtimes for closeness centrality.

Graph name	nr. of nodes	nr. of edges	ratio	average runtime in seconds
yeast	2361	7182	3.041931385	14.2806
facebook	4039	88234	21.84550631	358.1881
power	4941	6594	1.334547662	2.2281

Table 5.1: The influence of the edge to node ratio on closeness centrality

Comparing the graphs in table 5.1, it can be seen that the two graphs “facebook” and “power” have a similar number of nodes but the ratio of edges to nodes in “facebook” is significantly higher. In detail, the ratio is more than 16 times higher in “facebook” whereas the computation time is even more than 160 times higher although having a similar number of vertices, which shows that there is an impact of the ratio.
Even the graph “yeast” has a more than 6 times higher computation time than “power”, although it has less than 50 percent of the number of nodes but a higher edge to node

ratio as well.

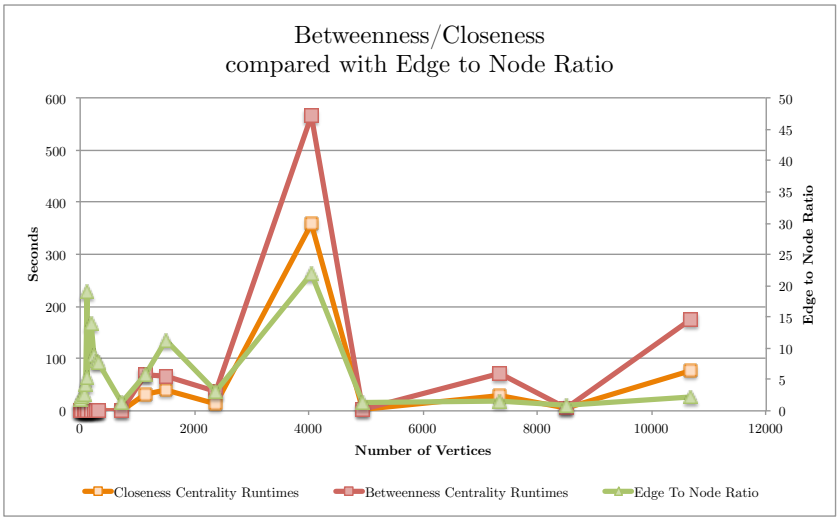


Figure 5.8: Evaluation result of the Closeness centrality and Betweenness centrality runtimes compared with the edge to node ratio

Figure 5.8 shows a comparison of the edge to node ratio with the runtimes on all evaluated graphs. It can be seen that the ratio and the runtimes show a similar behaviour as the graphs get larger. A mathematical analysis with the calculation of the correlation between the ratio and the closeness centrality runtimes emphasizes this observation as well. The overall correlation between these two variables is then 0.563 whereas the correlation of graphs with a size greater than 750 nodes have a correlation of 0.897 and the graphs of size greater than 2000 nodes even have a correlation of 0.983. All these observations demonstrate that the edge to node ratio certainly has an influence on the path collection algorithm.

5.2.3 Influence of Network Properties on Path Collection

It is also not too far fetched to take other network properties into consideration regarding their influence on the runtimes of the path collection algorithm. Since there exists such a variation among the runtimes it is important to observe how the network properties introduced in section 4.3 are structured and to possibly come up with an explanation thereof. As explained in the network properties section, they are indicators to show how a network is built up and it can be assumed that the structure of a network may have influence on the calculation time of a certain social network analysis method. The

following tables take the runtimes of closeness centrality into consideration, since the runtimes of betweenness centrality behave similarly as seen in section 5.1.2.

Influence of Diameter

Firstly, a look at the network diameter of selected graphs and their respective runtimes of closeness centrality.

Graph name	nr. of nodes	nr. of edges	diameter	average runtime in seconds
polblogs	1490	16715	9	41.1144
yeast	2361	7182	16	14.2806
facebook	4039	88234	17	358.1881
power	4941	6594	12	2.2281

Table 5.2: The influence of the network diameter on closeness centrality

Comparing the different times the algorithm takes to calculate the graph's closeness values with their respective diameter values, a clear correlation between these values can not be observed. The graph "facebook" has a similar diameter like the graph "yeast", but the runtime of "facebook" is significantly higher and even "polblogs" with a diameter that is lower than the one of "yeast" has a higher runtime than "yeast". Therefore, it can be said that the impact of the network diameter on the runtime of a social network analysis method is not decisive. The calculated correlation between diameter and the runtimes over all evaluated graph is 0.486, which goes along with the explanation made for table 5.2. Yet, it can not be said that there is no impact of the network diameter at all on the runtime, but the relatively low correlation value shows that the impact is minor.

Influence of Reciprocity

The next property to analyze its influence of is the reciprocity of a graph, which means how many paths in a graph also have a path in the other direction.

Graph name	nr. of nodes	nr. of edges	reciprocity	average runtime in seconds
email	1134	6554	0.999117389	32.0144
polblogs	1490	16715	0.242561245	41.1144
facebook	4039	88234	0	358.1881
power	4941	6594	0	2.2281

Table 5.3: The influence of the network reciprocity on closeness centrality

The comparison of the runtimes with their respective reciprocity of the four chosen graphs above is not showing an observable influence that reciprocity may have, either. The detailed view on the two graphs "facebook" and "power" even shows that the runtimes seem to be nearly independent of the reciprocity since both have the reciprocity value 0 and completely different runtimes while having similar graph sizes. When adding the graph "email" to this analysis, which is actually a considerable smaller graph, that has a reciprocity value close to 1, it gets clearer that the runtime has no correlation to the value of the graph's reciprocity. Indeed is the calculated correlation value between reciprocity and the evaluation runtimes of all evaluated graphs 0.149, which demonstrates that the impact of the reciprocity is low.

Influence of Density

The following table shows some selected graphs together with their density value and runtime in seconds in order to be able to interpret some conclusion out of these results.

Graph name	nr. of nodes	nr. of edges	density	average runtime in seconds
yeast	1490	16715	0.001288954	14.2806
facebook	4039	88234	0.005409982	358.1881
power	4941	6594	2.70E-04	2.2281
geom	7343	11898	2.21E-04	30.0189

Table 5.4: The influence of the network density on closeness centrality

Once again, the observation that can be made in this table is that it can not be said whether the density of a graph has an effect on the runtime of the program. The calculated correlation value underlines here as well the observation with a value of 0.236. Unfortunately, the outcome is the same as for reciprocity and the diameter.

Eventually, it can be said that there is no clear impact of the network properties on the

runtime of the implemented path collection algorithm. This may be rather surprising since the discussed network properties actually are well-accepted indicators to compare different graphs and it could have been expected that different properties may result in different runtimes. Obviously, they are not the main reason for varying runtimes and other effects have to be responsible for the volatile evaluation results. Probably, the calculation capacity of the graph or vertices is a more important influence on this algorithm or as well the described impact of edge to node ratio in section 5.2.2.

5.2.4 Influence of the Triad Census Type Distribution on the Triad Census Algorithm

The runtime evaluation of the triad census also showed some high variance of runtimes among the graphs. This may on one hand be explained with the edge to node ratio as in subsection 5.2.2, however when evaluating the runtime of the triad census, it is also of interest how the distribution of triad types may have some impact on the runtimes of certain graphs. Referring to section 4.2.6 and its algorithm in listing 4.1, it gets clear that triad types 1 and 2 occur most often and each graph usually has a large number of these triad types. Hence, it can be assumed that these types do not have big influence on the runtimes. For that reason, the number of occurrences of the other triad types which may influence the speed of the calculation of the triad types is given more attention.

name	nodes	edges	5	6	7	9	10	avg. runtime in s
p2p-Gnutella04	10876	39994	152450	179230	0	901	33	1.255
days	13332	148038	11737794	8586744	0	1170290	0	66.5866
as-22july06	22963	48436	11954446	486283	0	46873	0	24.2973
eatrs	23219	304937	4992572	5959409	1100144	295205	7372	73.6955
p2p-Gnutella24	26518	65369	183661	235029	0	956	30	1.8617

Table 5.5: The influence of the distribution of triad types on triad census calculation

Table 5.5 shows some exemplary evaluated average runtimes together with the occurrences of selected triad census types. The chosen types are considered more interesting for the evaluation since some occur in every viewed graph (types 5,6 and 9) and others occur only in a few of them (types 7 and 10).
The numbers in the top row indicate which triad census type is in that specific column. What kind of triad type the numbers show can be looked up in figure 4.1. The numbers in the table show how many occurrences a triad census type has in the respective graph.

Looking closer at the selected graphs and comparing several of them with each other shows some correlation between the occurrence of certain triad types and the graph's runtime. Regarding graph "p2p-Gnutella24" and "as-22july06", it is shown that "p2p-Gnutella24" has a larger size and the higher number of edges, but the calculation is performed much faster than with "as-22july06". Taking triad types 5 and 9, it can be seen that "as-22july06" has a substantial higher number of occurrences of these types than "p2p-Gnutella24" which is also reflected in a runtime that is more than 12 times higher.
Comparing the graphs "days" and "p2p-Gnutella04" or the graphs "eatrs" and "p2p-Gnutella24" also shows that the runtimes are fairly different between these graphs. Looking at how the triad types occur, this can be explained similarly as for the graphs "p2p-Gnutella04" and "as-22july06".
The comparison between the graphs "as-22july06" and "eatrs" shows, in addition to the before mentioned high number of the presented triad types, that a considerable difference of runtimes between graphs of almost the same size can still occur. Both graphs have a similar total number of the presented triad types, but still have differing runtimes. Looking at the number of edges of these two graphs therefore may be one answer as to why it takes much longer on average to determine the triad types of the graph "eatrs".

This leaves the question as to what the bigger influence has, the occurrences of certain triad types or the number of nodes or edges respectively. For sure, a high number of edges has some influence on the Signal/Collect algorithm. However, compared to the collection of shortest paths the collect function used for Triad Census is rather simple and should not lead to such high runtimes. Therefore, the main responsibility of high calculation time can be attributed to the algorithm of detemining the triad census type, as presented in listing 4.1. So, a high average runtime on a certain graph can be explained by having a high proportion of triad types which are more complex to determine which is more time consuming.

5.3 Evaluation for Correctness (Testing)

The evaluation for correctness of the project is in comparison to the runtime evaluation not concerned with a fast execution of the implemented code but rather with correctly executed code. It should demonstrate that the code calculates exactly the values of the described functionalities which it should.

5.3.1 Unit Tests

The Unit Tests were performed with the same graph for each test. The reason to do so was that this graph contains most of the special but occurring types of connections between vertices. There is a vertex with only outgoing edges and one with only incoming edges. There are undirected connections (edge in both directions), cycles and a part of the graph is separated from the rest of it. In that way, it can be ensured that the implemented algorithms work in every case and calculate the correct value. The described graph is illustrated in figure 5.9.

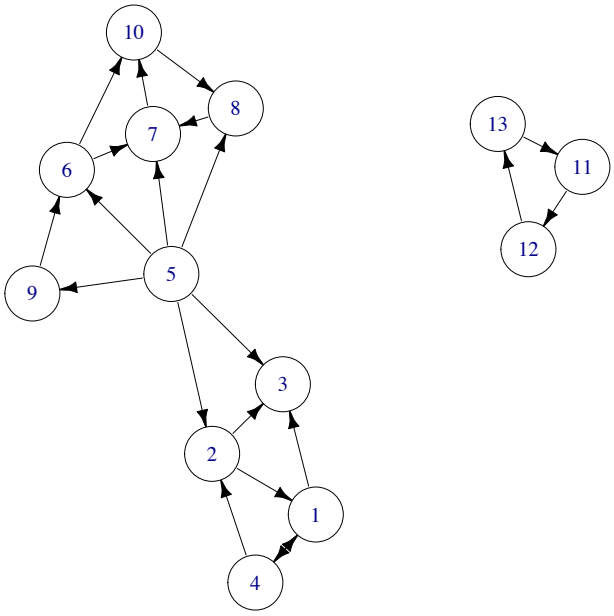


Figure 5.9: The graph used for testing purposes

The use of this small-sized graph also makes it possible to calculate the values that should be expected before they are run in that test.

Measure Test

The first test case covers the social network analysis methods. Each of these methods is run on the described graph (see figure 5.9) and the calculated values of the vertices are compared with the expected values that have been calculated before. Regarding betweenness centrality and closeness centrality, the calculated values could differ on

certain vertices since there may be different shortest paths of the same length from vertex v to vertex w and they do not have to go through vertex u necessarily. Therefore, only the vertices that have the same value of betweenness centrality and closeness centrality in each case could be tested.

Property Test

The property test case ensures that the network properties of the graph are calculated correctly. This test case tests in particular the introduced network properties as well as the degree distribution and the local cluster coefficient distribution of the presented graph.

5.4 General Analysis

To end this chapter, the general advantages and drawbacks of using the Signal/Collect programming model for this thesis are assessed.
The most obvious statement of the runtime evaluation is also influencing the general picture of using Signal/Collect for this thesis. As the detailed evaluation of closeness centrality, betweenness centrality and the triad census showed Signal/Collect's performance is rather bad when executing algorithms whose collect functions use a lot of computation and memory depending on how a graph is structured. Other tools may do so as well, but it is still a major drawback when it was the aim to achieve a good performance for these algorithms. Moreover, the memory overhead which happened over a certain threshold of graph size and number of edges is a bad sign regarding efficient computation. On one hand, the implemented algorithm is maybe not sophisticated enough for the desired performance, on the other hand, the limitations given by Signal/Collect may also influence that the tool performs in such a volatile manner and reaches memory overheads.
However, clear advantages of using Signal/Collect for this thesis can be seen on most of the other implemented functionalities. The use of the framework really simplified implementing PageRank, degree centrality or label propagation, just to name a few.

6

Conclusions

As stated in the introductory chapter, the aim of this thesis is to develop a tool that calculates basic and advanced methods of Social Network Analysis on a distributed computing environment. Eventually it should be shown that it is possible to implement such a tool on a distributed framework. To conclude this thesis, it is assessed if this has been achieved successfully and what kind of improvements still might be needed.

6.1 General Conclusions

As the final product of this thesis shows, it is indeed possible to develop such a tool on a distributed computing framework. All of the upfront set functionalities have been implemented on Signal/Collect. Although everything is working and some of the implemented features perform really well and show good evaluation results, others, unfortunately, suffer from high memory consumption and high variability when running them on Signal/Collect. This can mainly be attributed to the implemented algorithms which probably are not the most efficient but so far the only ones that could have been thought of in order to calculate the desired social network analysis method.
Signal/Collect with all its built-in functions proved to be beneficial when implementing some of the desired functionalities. For some of the Social Network Analysis methods, Signal/Collect was a perfectly suitable framework to use with its dedication to directed graphs. For others, the programming model was not very helpful while developing the tool. Taking the degree centrality, Signal/Collect made it really easy to compute that metric, since it already provides the number of incoming and outgoing edges at a vertex such that it was an easy task to calculate the degree centrality. On the other hand, the constraints of Signal/Collect did not facilitate the implementation of gathering shortest paths or determining the triad census types. Finally, the general conclusion is a double-edged with some clear advantages of using Signal/Collect, but also some drawbacks which were observed.

6.2 High Memory Consumption of Shortest Path Algorithm

A particularly unsatisfying performance showed the shortest path algorithm. While not being necessarily slow, the algorithm to gather all shortest paths, according to the evaluation, seems to be very memory consuming. Remembering that every vertex has to store all its incoming shortest paths, it is standing close to reason that a large graph needs a lot of memory to gather all relevant information in order to calculate all desired values. The Signal/Collect programming model with its feature of distributing the computation also does not help here, because all vertices have to store the shortest paths which arrive at a vertex. That may lead to the conclusion that Signal/Collect in that case is not a suitable framework to run such algorithms.
In addition, the evaluation revealed also that it was not possible to run this algorithm on really large graphs, which would have been desired since Signal/Collect is aiming to do work on large graphs.
Improvement may come when it is possible to stream the shortest paths directly to a central instance, but at the end of the day, all the shortest paths have to be stored somewhere to be able to calculate the respective social network analysis methods. Therefore it will be complex to improve the runtime and memory consumption in order to gather all shortest paths.
Another way of solving this problem may be the use of approximations. Bader et al. already provided a possible solution for betweenness centrality which uses an adaptive sampling technique that does not need to calculate a 100 percent exact betweenness but still produce satisfying results [Bader et al., 2007]. A similar technique may be adapted to Signal/Collect in order to construct a more efficient path collection algorithm.
Nevertheless, Signal/Collect is not a bad framework in general. Regarding the results of degree centrality, PageRank, local cluster coefficient or label propagation where such a load of memory is not needed, Signal/Collect seems to perform pretty well.

7

Future Work

7.1 Adaption of Label Propagation

As seen in section 4.4.1, label propagation has some sophisticated opportunities to be observed in real life. The implementation as presented in this thesis is however rather naïve and some extension or improvement could make it more powerful.
One option is to choose the label differently in order to reach some desired propagation of the labels. For example, a certain sort of label is always more important or has always the higher importance than others and a node takes this label even if it is not the one that has the majority.
Another opportunity is to make label propagation more interactive. For example, a user may decide what to do after a certain number of steps, if the algorithm should be executed further, if labels should be added or deleted or to change the priorities or importance of the labels and so on.

7.2 Introduction of other Advanced Methods

A few of the most important social network analysis methods are now implemented for Signal/Collect. But there are still some remaining methods out there. One of them is hierarchical clustering. As the presented and implemented local cluster coefficient, hierarchical clustering is concerned with community detection but it is not only interested in finding communities but also with finding some hierarchies in these communities.
The concept of random walks, as introduced in section 2.2, is also conceivable to be implemented in Signal/Collect. It does not necessarily need to implement the functionality as in the described related work, it should only make use of the general concept of random walks for which various different usages are imaginable.

7.3 Improved Path Collection

As the runtime evaluation proved, betweenness centrality and closeness centrality do not have a good performance in Signal/Collect yet. Most likely this is due to the complex and time consuming way of how the paths are collected and since the goal is to collect all shortest paths there.
One possible solution to address this problem is to get rid of counting the shortest paths after they have been collected or even better to be able to collect shortest paths and calculate closeness and betweenness centrality in the same step.
Another way to improve the performance of path collection is to make some assumptions and have simplifications or estimations. Probably it is enough to gather a certain amount of shortest paths in order to be able to calculate the centrality values. Future work may show in some experiments that only a determined number of signal and collect steps depending on the size of the graph can be enough to calculate sufficient results. Referring to Cooper et al., whose work shows that some mathematical assumptions in combination with random walks can be enough to estimate the global network properties [Cooper et al., 2013]. By doing similar applications to Signal/Collect, it may be shown that the same can be achieved for a better performance of path collection while still showing accurate results.

A

Appendix

A.1 The used Datasets

name	nr. of nodes	nr. of edges	density	diameter	reciprocity
example	10	18	0.2	3	0.387096774
karate	34	78	0.0695	3	0
dolphins	62	159	0.042	6	0
polbooks	105	441	0.0404	8	0
ht2009_15min	113	2163	0.171	4	0
football	115	613	0.0468	9	0
jazz	198	2742	0.0703	9	0
airlines	235	2101	0.0382	4	0.995726496
celegansneural	297	2345	0.0267	14	0.840961504
celegans	306	2345	0.0251	14	0.8409615
codeminer	724	1025	0.00196	5	0.007465782
email	1134	6554	0.00935	7	0.999117389
polblogs	1490	16715	0.008575189	9	0.242561245
yeast	2361	7182	0.001288954	16	0
facebook	4039	88234	0.005409982	17	0
power	4941	6594	2.70E-04	12	0
geom	7343	11898	2.21E-04	12	0
eva	8497	6711	9.32E-05	11	0.002658443
pgpgiantcompo	10680	24316	2.13E-04	13	0
p2p-Gnutella04	10876	39994	3.38E-04	26	0
days	13332	148038	8.33E-04	13	0
comichero	19286	96519	2.60E-04	2	0
p2p-Gnutella25	22687	54705	1.06E-04	22	0

as-22july06	22963	48436	9.19E-05	16	0
eatrs	23219	304937	6.04E-04	7	0.123626272
p2p-Gnutella24	26518	65369	9.30E-05	29	0
hep-th	27240	341923	4.62E-04	37	0.002774549
hep-th-new	27770	352285	4.58E-04	37	0.002773834
p2p-Gnutella30	36682	88328	6.56E-05	24	0
p2p-Gnutella	62586	147892	3.78E-05	31	0
soc-Epinions1	75879	405740	8.84E-05	16	0.405226035
soc-slashdot0811	77360	469180	1.51E-04	12	0.866935294
wordnet	82670	67837	9.93E-06	26	0.000206356
soc-sign-epinions	131829	817888	5.46E-05	16	0.273545426
com-amazon	288799	703284	8.43E-06	22	0

Table A.1: Data Sets

Bibliography

[Agarwal et al., 2007] Agarwal, A., Slee, M., and Kwiatkowski, M. (2007). Thrift: Scalable cross-language services implementation. Technical report, Facebook.

[Auber, 2003] Auber, D. (2003). *Tulip: A huge graph visualisation framework*. P. Mutzel and M. Junger.

[Bader et al., 2007] Bader, D. A., Kintali, S., Madduri, K., and Mihail, M. (2007). Approximating betweenness centrality. In *Algorithms and models for the web-graph*, pages 124–137. Springer.

[Barabási et al., 2000] Barabási, A.-L., Albert, R., and Jeong, H. (2000). Scale-free characteristics of random networks: the topology of the world-wide web. *Physica A: Statistical Mechanics and its Applications*, 281(1):69–77.

[Bastian et al., 2009] Bastian, M., Heymann, S., and Jacomy, M. (2009). Gephi: An open source software for exploring and manipulating networks.

[Batagelj and Mrvar, 2001] Batagelj, V. and Mrvar, A. (2001). A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*.

[Brin and Page, 1998] Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*.

[Cooper et al., 2013] Cooper, C., Radzik, T., and Siantos, Y. (2013). *Fast Low-Cost Estimation of Network Properties Using Random Walks*. Springer.

[Csárdi and Nepusz, 2006] Csárdi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, Complex Systems.

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

[Gansner and North, 2000] Gansner, E. R. and North, S. C. (2000). An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*.

[Hanneman and Riddle, 2005] Hanneman, R. and Riddle, M. (2005). *Introduction to Social Network Methods*. University of California.

[Lawyer, 2014] Lawyer, G. (2014). Understanding the spreading power of all nodes in a network: a continuous-time perspective. *CoRR*, abs/1405.6707.

[Lindholm and Yellin, 1999] Lindholm, T. and Yellin, F. (1999). *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[Newman, 2008] Newman, M. (2008). *The mathematics of networks*. Palgrave Macmillan, Basingstoke.

[Odersky et al., 2004] Odersky, M., Altherr, P., Cremet, V., Dargos, I., Dubochet, G., Emir, B., McDirmid, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Spoon, L., and Zenger, M. (2004). An overview of the scala programming language. Technical report, EPFL, Lausanne.

[Page et al., 1998] Page, L., Brin, S., Rajeev, M., and Winograd, T. (1998). The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University.

[Porter, 2000] Porter, M. E. (2000). Location, competition, and economic development: Local clusters in a global economy. *Economic development quarterly*, 14(1):15–34.

[Stallman, 2007] Stallman, R. (2007). Gnu general public license, version 3. *http://www.gnu.org/licenses/gpl-3.0.html*. Accessed: 2014-10-06.

[Stutz et al., 2010] Stutz, P., Bernstein, A., and Cohen, W. (2010). *Signal/Collect: Graph Algorithms for the (Semantic) Web*. Proceedings of the ISWC, Shanghai.

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

List of Figures

List of Tables

List of Listings