A Toolkit for Real-time Analysis of Dynamic Large-Scale Networks

(Invited Paper)

Ruud van de Bovenkamp and Fernando Kuipers
Network Architectures and Services
Delft University of Technology

***Abstract*—Networks are used in many research domains to model the relationships between entities. We present a publicly available toolkit to extract graphs from datasets or data streams and to analyse their properties. The graph extraction is based on a set of rules that define the links between entities in a set or stream of self-contained events involving sets of entities. As the extracted graph is dynamic and, moreover, can be spread over multiple machines, we include the class of gossip algorithms to analyse them. In addition, the toolkit also contains algorithms to compute metrics of static snapshots of the dynamic graph.**

I. Introduction

Many research domains use networks as modelling tools. Networks can represent vastly different objects including physical infrastructures such as rail, road and waterways (e.g., [1]), flight routes and shipping lanes (e.g., [2], [3]), but also sewage systems and power grids (e.g., [4]), and, of course, the internet. Networks can be constructed from financial transactions [5], friendship or collaboration relations among individuals [6], (P2P) peers exchanging data (e.g., [7]), sports players or online gamers that have played on the same team [8], and more abstract things such as functional brain networks where the nodes in the network are brain regions that share a link when they show correlated activity [9], or co-purchase networks where nodes are items in a shop that share a link when they were purchased together [10]. In short, everything that can be represented as a collection of entities that have a relation can be modeled as a network.

The study of complex networks currently faces two main challenges: 1) The size of the studied networks has grown enormously. Especially online communities such as social networking sites have exploded in size and easily contain hundreds of millions of nodes. In addition, the availability of machine-readable data has enabled companies and researchers to construct increasingly large networks.

Very large networks are a problem for two reasons. First, in some cases the networks are so big that it is no longer possible to process them in a single machine. Often, the networks are not even stored in a single location, but are distributed over multiple servers in multiple locations. Second, the time complexity of the algorithms used to calculate certain properties does not scale well with the network size.

Simple network properties that are limited in scope to single nodes are not challenging to calculate. When our scope encompasses the entire network, however, the real challenges begin. Calculating how many of the shortest paths in a network go via a particular link (link betweenness), for example, is computationally expensive; other properties, such as the isoperimetric constant or the facility location problem, are even NP-hard to determine.

2) The second challenge that network science faces is that of network dynamics. Real networks are not fixed in time; they are constantly changing. In an online social network, for example, new users sign up and existing users leave, thereby changing the number of nodes. At the same time users make new friends or break old ties, which changes the topology. If we are not just interested in the links between user profiles, but only consider the users to be present in the network when they are logged in, the network changes are quick and substantial. Since a social network is very large, it is currently impossible to accurately track changes in network metrics as a function of time. Traditional methods of calculating properties of networks break down for very large and dynamic networks.

Before graph analysis can begin, there is the challenge of extracting a graph from a dataset or a data stream. We have developed a suite of tools, which is publicly available [11], that can be used for all steps needed to analyse the relationships between entities in datasets or data streams. Although other tools exist, such as Jung, Igraph, Pajek, and Cytoscape for static network analysis, Gephi and NetLogo for visualisation, DeSiNe [12] for traffic dynamics, and PeerSim for simulating gossip algorihtms, the strength of our suite is that all the components are in one place, using the same underlying data structures, and that we consider dynamic networks.

We will describe our toolkit in four sections: Section II will provide an overview of the elements in the toolkit and how they can be combined for real-time analysis of dynamic graphs. In Section III, the graph extraction from either a dataset or data stream is introduced in detail, after which in Section IV we include the class of gossip algorithms as a way to compute properties of the extracted graph. Finally, in Section V, we briefly describe the available tools in the toolkit to do static graph analysis on snapshots of the dynamic graph, and Section VI discusses some future work and conclusions.

II. Overview Toolkit

The main idea behind the toolkit is to combine three parts, graph extraction, static and dynamic graph analysis, to analyse

978-1-4799-3206-1/13/$31.00

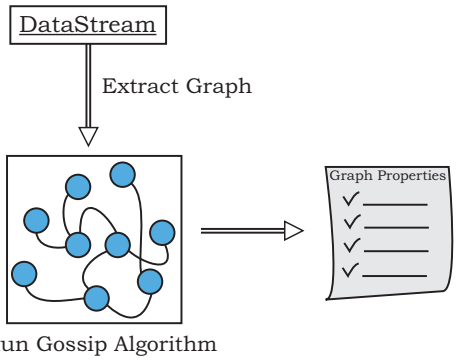

Figure 1. Main steps in the data stream analysis. The data stream is passed through a filter to extract a dynamic graph containing the relations between entities of interest in the data stream. On this large and dynamic graph a gossip (or alternative) algorithm is ran to compute properties of interest.

data streams. The analysis is illustrated in Figure 1. The data stream that we want to analyse is passed through a filter that extracts a dynamic graph based on a set of rules. This set of rules determines what will become nodes in the graph and what will become links. We call this mapping a dataset onto a graph. The details of how to extract graphs from a dataset and how different mappings influence the resulting graph are explained in more detail in Section III. The novel application here is that instead of mapping a fixed or static dataset onto a static graph, we apply these techniques to a data stream. The result of mapping a data stream onto a graph is not a static graph but a dynamic one. The filter continuously outputs graph dynamics: nodes and links are joining or leaving the graph depending on the data that is streamed through the filter.

We use a gossip algorithm to continuously compute or update those graph properties that we are interested in, although other dynamic algorithms could also have been used. Gossip algorithms have proven to be ideal candidates to perform computations in large, dynamic and distributed systems. In Section IV, we will introduce gossip algorithms in more detail and explain how we simulate gossip algorithms. We will give an example algorithm to compute averages or to find minima and maxima over node values.

III. Graph Extraction

Graphs are commonly used as models to study the properties of various different physical or non-physical problems. A communication network, for example, can consist of a group of computers connected via Ethernet links. For such a network we can then compute how many links can be removed without losing connectivity. In this example, the step from the physical world of computers and cables to the more abstract graph model is very intuitive. The computers are our nodes, and the cables are our links.

When we study less tangible networks, such as social networks or implicit networks in datasets, we have to decide what are going to be the nodes in the graph and what are going to be the links. More formally, a dataset D is mapped onto graph G via a mapping function $M(D)$. A simple undirected and unweighted graph $G = (\mathcal{N}, \mathcal{L})$ consists of a set $\mathcal{N}$ of N nodes and a set $\mathcal{L}$ of L links. In a weighted graph, a link weight w is associated to every link in $\mathcal{L}$. In a directed network, a link between two nodes also has a direction.

A. Extraction rules

A *mapping* is a set of rules that define the nodes and links in a graph. Entities are often mapped to nodes, while relations between entities are mapped to links. Entities are usually readily identifiable as persons, events, or objects and are therefore intuitively mapped onto nodes.

Mapping relations to links, however, is more challenging. Entities can be related to each other in many different and often subtle ways and due care should be given to which relations produce insightful graphs. For example, some relations between entities may be the result of chance, while others have a clear origin. The difference between random and meaningful relations is often expressed by a notion of strength. Strength, represented by a link weight, adds another dimension of complexity to representing and understanding the characteristics of a network.

Our tool to extract graphs from datasets was developed in the context of identifying implicit social networks in the game data from online social games [8], but is not restricted to be used in the gaming or social network setting. Graphs can be extracted from every dataset that consists of a collection of self-contained events that involve a group or groups of entities. In [8], the self-contained events were matches played in an online game and the groups of entities were two (not necessarily equally sized) teams of players. The data was crawled from a website and then parsed into a *Match* object containing meta-data such as the start and end times of the match, and score-related information.

Graphs were extracted from the *Match* objects based on two different sets of rules: global rules and local rules. Global rules apply to the dataset as a whole: for example, a link exists between two players that were present in the same game. Local rules apply to individual potential links. For example, while a global rule dictates that two players have a link if they have been in a match together, a local link can dictate that that match must have had a duration of at least 20 minutes.

Both local and global rules can either be simple, or compound. Simple rules consist of a single requirement such as "in the same game together" or "played in the afternoon", whereas compound rules are logical combinations of simple rules. The rules can be specified in a configuration file in bracket notation, for example ([link won] & [duration > 20]) | ([link lost] & [duration < 10]).

Although all our examples are in the setting of online games, the same machinery can be used in different settings. For example, the self-contained events could be items in a shopping cart, or people on a bus, etc. The flexibility of our tools for graph extraction enabled us to study the effect of different mappings and thresholding on the social networks extracted from gaming data.

B. Link Set

Every time a link is formed as a result of the extraction rules, it is stored in a link set. However, we might want a link to be formed a few times before we want to consider it as a valid link in the graph. To achieve this, we count how many times a link is formed (we increment the link weight every time it is formed) and only add it to the graph when the link occurred more than a threshold value. In this case, the link set can be seen as an intermediate step between the data stream and the graph. The link set is implemented as a balanced search tree, which is a data structure that is designed to have approximately the same number of nodes to the left and right of each node. The fact that a tree is balanced guarantees that a search in the tree will be of $O(\log n)$, where n is the number of entries in the tree. The balanced search tree that we use in the link set is an implementation of the red and black tree [13].

Each link that is added to the tree has a unique value associated with it that serves as an identifier for the link but also enables links to be ordered lexicographically. The link id is simply the binary concatenation of the ids of the source and destination nodes. If links are bidirectional, the link id is the concatenation of the bigger and smaller node ids of the endpoints.

Every time a link is added to the link set, its link id is compared to that of the root node. If it is greater than that of the root node, it is compared to the right child of the root node, otherwise to the left child. This continues until either (i) a node is found that has the same value as the link to be inserted, or (ii), it is smaller (larger) than the current node but larger (smaller) than its left (right) child. In the first case, the link weight of that link will be incremented; in the second case, the link is added to the tree in the current position and the tree is rebalanced.

Since the link set can take up quite some memory, especially when it contains hundreds of millions of links, it is not unlikely that the tree runs out of memory. The solution to this problem depends on whether the tree is used to store the links from a static dataset or a data stream. For the static case, the link set can be stored to disk when it reaches a certain size. After storing the link set to disk it is cleared and can be filled again until it reaches the predefined size again, after which it is merged with the tree on disk. In this fashion, the stored link set can grow very large. The stored link set can be read from disk once to extract the graph, possibly while applying a threshold on the link weight. By writing the link set to disk it is also easier to create graphs from a link set using different threshold settings later.

If the link set is the result of streaming data through the filter, writing an intermediate file to disk if memory consumption grows too large is no longer an option because of the long disk access time which makes merging the tree in memory with the tree on the disk too slow. Also, since we want to have exact knowledge of when a link is added to the network, we would have to merge the tree in memory and the one on disk after each link addition. This effectively reduces the link set to exist on disk only. As the entire link set has to be kept in memory, we either have to use a machine with a very large amount of memory, or we have to spread the link set over a number of machines in a computation cluster.

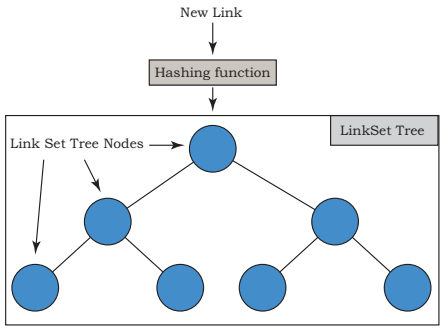

Figure 2. Schematic overview of inserting links in the distributed link set. A new link is first passed through a hashing function to get a hash of the link. The link and hash are then offered to the centre node of the LinkSet tree. Based on the hash, the link will end up in one of the partial link sets in the tree.

We have implemented a tree structure of partial link sets that allows the link set to be distributed over multiple machines. Each individual partial link set is used to store a part of the links. The simplest way to spread the possible links over the link sets is to divide the space between the highest and lowest known link evenly between the number of link sets. However, since it is not known beforehand how many links will be stored in the link set, the distribution over the partial link sets might become skewed over time. If the links are too unevenly spread over the partial link sets, they have to be rebalanced to make sure that all link sets grow at approximately the same rate.

Our approach to spread links over the partial link sets is centred around a hashing function. An ideal hashing function has a uniform output probability for every input. If each of the partial link sets is responsible for a part of the output range of the hashing function, a new link has equal probability of being assigned to any of the partial link sets. In this way, the distribution of the links over the partial link sets will be uniform. The implementation of our tree of partial link sets is based on the *LinkSetTreeNode* as illustrated in Figure 2. First the link id of a new link is hashed. The link and its hashed id are offered to the central node in a LinkSetTree. From the central node the link is passed on until the *LinkSetTreeNode* where the link should be stored is found.

The pseudo-code in Algorithm 1 illustrates how the correct *LinkSetTreeNode* is found. Each *LinkSetTreeNode* has a lower and upper limit of link ids that it is responsible for. It also has a pointer to a smaller and larger *LinkSetTreeNode.* When a link is offered to the node, it checks whether the (hashed) id is smaller or larger than the lower and upper limit, and, if it is, it passes the link on to the smaller or larger node. If the hashed link id is within the range of the current *LinkSetTreeNode*, it will send the link to its partial link set via a TCP connection.

The distributed link set is essentially a client-server application where the server uses the LinkSetTree to find out which

```
Object LinkSetTreeNode()
    Fields:
    int lowerlimit, upperlimit
    LinkSetTreeNode() smaller, larger
Method insert(link)
    if link.id < lowerlimit then
        smaller.insert(link)
        return
    end
    if link.id > upperlimit then
        larger.insert(link)
        return
    end
    send link to link set via tcp
    return
```

Algorithm 1: Link insertion in distributed link set

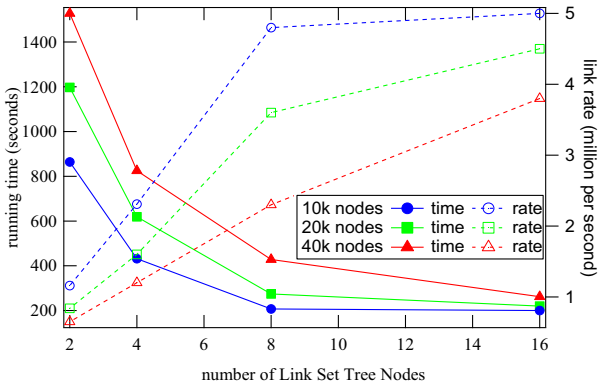

Figure 3. Running time for the addition of 10^9 links to the distributed link set as a function of the number link set tree nodes for three different random graphs. The number of unique links in the tree is approximately 5×10^7, 2×10^8 and 5.7×10^8 for the three curves.

client should store which link. The clients use an ordinary link set as described above to store the links. Clients do not all have to have the same amount of memory available. Each client can indicate the maximum number of links it thinks it can store when it contacts the server. The server can use this information to distribute the output range of the hash function in such a way that each client is responsible for a part of the range proportional to its storage capacities.

In addition to a distributed implementation of the link set, the partial link sets can also be exploited to speed up the storage in a single large memory machine. Figure 3 shows a measurement of the running time of adding a billion links to the link set in three different scenarios as a function of the number of partial link sets. Our test machine featured 16 logical cores and 48 GB of memory. In this experiment the added links are randomly generated. The source and destination nodes are drawn uniformly from a set of integers between 0 and $N-1$, where N is the number of nodes in the graph and is chosen to range from 10^4 to 4×10^4. Clearly, the larger N is, the larger the number of possible unique links. In our experiment the number of recorded unique links was approximately 5×10^7, 2×10^8 and 5.7×10^8. Also shown in Figure 3 is the rate of link additions. A link addition in this case is simply an offered link, of which there were 10^9 in every experiment.

Figure 3 shows that using more partial link sets leads to lower running times. The reduction in running time is larger when the number of unique links in the tree is larger. This is caused by the computational complexity of finding the right insert point in the search tree for larger trees. Figure 3 also shows that for a small number of unique links in the tree, the overhead of sending the links over TCP outweighs the computational efficiency of smaller trees. For example, there is virtually no benefit of using 16 in stead of 8 partial link sets for a graph with 50 million unique links.

C. Graph dynamics

Going from a link set to an evolving graph involves two opposing processes: growing and shrinking. The growing dynamics are more easily extracted from the link set. Every time a link is found in the streamed data, it is inserted into the link set as described above. If the link was not yet in the link set, it is added, otherwise, the weight of the link already present in the link set is incremented by one. While incrementing the link weight, it is easy to check whether it goes from just below the threshold to over the threshold, and, if it does, to add the link to the graph. If either of the nodes adjacent to the link is not present in the graph yet, it is also added. This process will, however, lead to an ever growing graph. It is probably desirable to include a mechanism to remove links that are no longer deemed important.

There are various options available for shrinking the graph and the link set. First of all it is possible to periodically remove all the links from the link set whose link weight is smaller than a certain threshold value. A periodic cleaning of the link set reduces the amount of memory that is used by links that were formed only once or twice, probably as the result of a chance encounter. If the cleaning threshold is lower than the threshold used to create the graph, however, the periodic cleaning does not lead to links being removed from the graph.

Another way to introduce shrinking is to use the time information of link creation. It is possible to use (i) full time information: each increment has a time stamp, (ii) partial time information: only the time stamp of increments within a fixed window from the last increment are kept, and possibly the maximum number of increments ever recorded in that interval, or (iii) minimal time information: only the time stamp of the last increment is used. By keeping this time information, it is possible to apply global rules such as “a link exists between players that played more than 10 times together within any 24 hour period.” Obviously, the more time information is kept, the more storage is needed. Even when storing time information, shrinking of the link set and graph will have to be done

periodically and involves variations on the theme of removing links that have not been formed the last x time units.

IV. Dynamic Graph Analysis

In order for the dynamic graph resulting from the graph extraction to offer insight in the relationships between entities in the data stream, graph properties should be computed. Ideally, algorithms that update a metric following graph dynamics, instead of recomputing the metric, are used. However, this is not always possible. Moreover, if the link set (and graph) is spread out over multiple machines, both distributed and dynamic algorithms are needed. Gossip algorithms have proven to be useful to determine properties of large, distributed and dynamic graphs, which makes them ideal to analyse the graph extracted from a data stream.

In a gossip algorithm, nodes typically select one of their neighbours periodically to send, request or exchange information with. Depending on whether nodes send, request or exchange information, the process is called Push, Pull, or Push-Pull. The timing of nodes becoming active can either be synchronous or asynchronous. In the synchronous case, each node becomes active once during a time window. Nodes will need to be able to loosely synchronise to prevent too much misalignment of time windows. In the asynchronous case, a node typically becomes active when an exponentially distributed timer expires. In other words, the node's activity follows a Poisson process.

In a gossip algorithm, nodes send a limited amount of information and keep very little state information. Nonetheless, these algorithms are capable of delivering global network information to all nodes without central control while being robust against message delivery failure. Data aggregation is a particularly fruitful application of gossip algorithms. The goal of aggregation algorithms is to inform every node in the network about some global property without the aid of a central authority. In the setting of a sensor network, this could be the average reading over all sensors, or the maximum or minimum reading. The work of Boyd [14] contains a wealth of references on distributed averaging. Also, see [15] for applications of gossip algorithms in signal processing.

The simple format of gossip algorithms makes them very suitable to be deployed in a general setting. Our toolkit offers such a setting where a set of data structures and functions is offered to simulate the behaviour of the algorithm in a somewhat idealised setting. The simulator offers both a synchronous and asynchronous mode of operation. The pseudo-code of the synchronous simulation mode is given in Algorithm 2. In the synchronous mode, the node ids are stored in an array which is shuffled each iteration. During the iteration, the nodes are processed in the order of which they appear in the array. The shuffling and iteration-wise operation of the simulation ensures that every node will be active only once during an operation window and at the same time it mimics the behaviour of loosely synchronized clocks at the nodes. Without shuffling the nodes will be processed in the same pattern every iteration.

```
Data: array order
      containing node
      ids
while true do
    shuffle order;
    for node n in order
    do
        activate n;
    end
end
```

Algorithm 2: Synchronous Simulation

```
Data: Timeline T
while true do
    take ticket from T;
    activate owner;
    set new ticket time;
    insert ticket in T;
end
```

Algorithm 3: Asynchronous Simulation

Alternatively, the node array can be shuffled after a particular number of cycles.

In the asynchronous case, the simulator has to keep track of the expire times of every node's internal timer. These times are stored as tickets in a time line. A ticket consists of a time value and an owner value. The owner value indicates which node issued the ticket and the time value indicates when the timer of that node expires. The time line is implemented as a balanced search tree. Storing the expire times in a tree makes it easy to chance the distribution of the waiting times. The simulator continuously takes the ticket with the smallest time value from the tree and makes the node perform its action. When the node is done, it issues a new ticket to be inserted in the time line with a time value that is an exponentially distributed random value removed in time from the current moment. The tree structure ensures that insert and remove operations take $O(\log N)$ time, where N is the number of tickets in the time line.

The network structure that is used in the gossip simulation is designed to make it easy to facilitate network dynamics. All nodes are stored in a linked list, making additions easy and quick. Removals are slower since the node to be removed has to be located first. In case of heavy churn, nodes could also be stored in a search tree, just like the tickets in the asynchronous simulation mode. Each node internally also stores a linked list with pointers to its neighbours. Just as node dynamics is optimised for growing networks, the linked lists for the links between nodes also favours adding links in terms of running time. For heavy link removal it could also be better to use a tree structure. Every time a node tries to send a message over a link, it has to check whether the node the link points to is still present. If not, it should remove the link.

In addition to the timing and network dynamics the simulator also offers a structured way of implementing a gossip algorithm. Every class implementing the node behaviour interface can be loaded as a gossip algorithm. The simulator offers two ways of performing the actual contact between nodes: the direct method and the message method. In the direct method, the active node can directly interact with the procedures and fields of the behaviour code of its neighbours. This makes the interaction between nodes quicker, but it requires all the nodes to be in a single machine. In the message method, the active

node prepares a message containing all the values that it wants to exchange with its neighbour and passes that message to the simulator. The simulator then delivers the message to the target neighbour and sends the reply to the active node. Although this requires a few intermediate steps, this method allows nodes to be spread over different machines. Spreading nodes over different machines can be beneficial when the network grows very large.

A basic example of a gossip algorithm that can be used to compute properties of the large dynamic graph that is the output of the graph extraction filter is Gossipico [16]. Gossipico can be used to sum, average or find the minima and maxima over node related values. A typical example function of Gossipico is to count the number of nodes and links in the graph and to compute the maximum degree. These functions can be combined in Gossipco, but it is also possible with the current simulation framework to run multiple gossip algorithms in the same network.

V. Additional Features

An alternative to the dynamic graph analysis described in Section IV is to take snapshots of the dynamic graph and perform traditional static graph analysis. Depending on the metric of choice, a dynamic/gossip algorithm may not realise a significant speed-up compared to a static algorithm. To minimize the need of additional programs, our toolkit contains a set of implemented algorithms to compute various network metrics. The spectral metrics (eigenvector centrality, algebraic connectivity, spectral gap) rely on the eigenvalue decomposition of the CERN Colt libraries in some cases. In other cases an implementation of the power method is used.

The hop-count related metrics, such as the average shortest path or the eccentricity can be computed individually or as a by-product of computing the node betweenness. The betweenness code is an implementation of Brandes' work [17] on betweenness algorithms. The coreness algorithm is an implementation of Batagelj [18]. Other supported operations are the computation of the number of connected components and various ways of exporting networks including Cytoscape and Pajek formats.

Another feature that is offered by the graph analysis tools is the ability to generate synthetic networks according to numerous different network models. The code to generate Erdős-Rényi random graphs and preferential attachment graphs is based on work by Batagelj and Brandes [19].

VI. Conclusions

We have presented a toolkit that offers all necessary tools to analyse datasets or data streams using network science. The links between entities in the dataset or data stream can be extracted using different types of link rules. These links are stored in a link set that can be spread out over multiple machines if the link set becomes large. When a data stream is analysed, the link set and extracted graph will be dynamic. In order to analyse the properties of a large dynamic graph that is possibly spread out over multiple machines, we have offered the class of gossip algorithms. The toolkit comes with a simulation environemnt for gossip algorithms as well as traditional metrics to study snapshots of a dynamic graph.

Although the dynamic and distributed link set is fully implemented in the toolkit, as well as a sample gossip algorithm, more work is needed to determine which graph metrics can be computed using gossip algorithms and which cannot. For those metrics that cannot be computed using gossip algorithms, new (preferably dynamic) algorithms are needed.

We have performed basic measurements of the rate at which links can be inserted in the link set as a function of how many partial link sets are used, but how large the link set can realistically grow remains unexplored. Future work is needed to determine the absolute performance limits of our toolkit.

References

[1] P. Angeloudis and D. Fisk, "Large subway systems as complex networks," *Physica A*, vol. 367, pp. 553–558, 2006.

[2] R. Guimera, S. Mossa, A. Turtschi, and L. Amaral, "The worldwide air transportation network: Anomalous centrality, community structure, and cities' global roles," *PNAS*, vol. 102, no. 22, pp. 7794–7799, 2005.

[3] P. Kaluza, A. Kölzsch, M. Gastner, and B. Blasius, "The complex network of global cargo ship movements," *Journal of the Royal Society Interface*, vol. 7, no. 48, pp. 1093–1103, 2010.

[4] R. Albert, I. Albert, and G. Nakarado, "Structural vulnerability of the north american power grid," *PRE*, vol. 69, no. 2, p. 025103, 2004.

[5] H. Wang, E. van Boven, A. Krishnakumar, M. Hosseini, H. van Hooff, T. Takema, N. Baken, and P. Van Mieghem, "Multi-weighted monetary transaction network," *Advances in Complex Systems*, vol. 14, no. 5, pp. 691–710, 2011.

[6] M. E. J. Newman, "The structure of scientific collaboration networks," *PNAS*, vol. 98, no. 2, pp. 404–409, 2001.

[7] Y. Lu, J.D.D. Mol, F.A. Kuipers, and P. Van Mieghem, "Analytical Model for Mesh-based P2PVoD," in *Proc. of the 10th IEEE International Symposium on Multimedia (IEEE ISM 2008)*, Berkeley, California, USA, December 15-17, 2008.

[8] R. van de Bovenkamp, S. Shen, A. Iosup, and F. A. Kuipers, "Understanding and recommending play relationships in online social gaming," in *COMSNETS 2013*, 2013.

[9] C. J. Stam, A. Hillebrand, H. Wang, and P. Van Mieghem, "Emergence of modular structure in a large-scale brain network with interactions between dynamics and connectivity," *Frontiers in Computational Neuroscience*, vol. 4, no. 133, 2010.

[10] G. Oestreicher-Singer and A. Sundararajan, "Recommendation networks and the long tail of electronic commerce," *Available at SSRN 1324064*, 2010.

[11] R. van de Bovenkamp. Network analysis toolkit. [Online]. Available: https://github.com/TUDelftNAS/

[12] T. Kleiberg, B. Fu, F.A. Kuipers, S. Avallone, B. Quoitin, and P. Van Mieghem, "DeSiNe: a flow-level QoS simulator of Networks," in *Proc. of the first International Workshop on the Evaluation of Quality of Service through Simulation in the Future Internet (QoSim)*, Marseille, France, March 3, 2008.

[13] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[14] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized gossip algorithms," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2508–2530, 2006.

[15] A. Dimakis, S. Kar, J. Moura, M. Rabbat, and A. Scaglione, "Gossip algorithms for distributed signal processing," *Proceedings of the IEEE*, vol. 98, no. 11, pp. 1847–1864, 2010.

[16] R. van de Bovenkamp, F. A. Kuipers, and P. Van Mieghem, "Gossip-based counting in dynamic networks," in *Networking 2012*, 2012.

[17] U. Brandes, "On variants of shortest-path betweenness centrality and their generic computation," *SOCIAL NETWORKS*, vol. 30, no. 2, 2008.

[18] V. Batagelj and M. Zaversnik, "An o(m) algorithm for cores decomposition of networks," *CoRR*, vol. cs.DS/0310049, 2003.

[19] V. Batagelj and U. Brandes, "Efficient generation of large random networks," *PRE*, vol. 71, no. 3, p. 036113, 2005.