Dynamic Network Analyzer
Building a Framework for the Graph-theoretic Analysis of Dynamic Networks

Benjamin Schiller and Thorsten Strufe
P2P Networks - TU Darmstadt
[schiller, strufe][at]cs.tu-darmstadt.de

Keywords: Networks, Graph theory, Measurements, Algorithm design and analysis

Abstract

With the rise of online social networks and other highly dynamic system, the need for the analysis of their structural properties has grown in the last years. While the recomputation of graph-theoretic metrics is feasible for investigating a small set of static system snapshots, this approach is unfit for the application in highly dynamic systems where we aim at frequent property updates. Based on the concept of data streams, new algorithms have been developed that update the computed properties based on changes instead of recomputing them regularly.

While there exists a plethora of frameworks and libraries for the analysis of static networks, there is currently no framework for the graph-theoretic analysis and development of new algorithms for dynamic networks.

In this paper, we discuss a set of requirements a framework must meet to implement the general workflow for analyzing dynamic networks. We then introduce the architecture of a first prototype for such a framework, the *Dynamic Network Analyzer* (DNA).

1. INTRODUCTION

Many systems in the focus of different research areas can be modeled as graphs. Common examples include transportation, computer, social, and biological networks. The graph-theoretic analysis of their underlying topologies allows to gain insights into their properties. In the case of evolving networks, whose structure changes over time, it is crucial to observe the transition of key properties to understand their characteristics and dynamics.

A prominent example of dynamic systems are social networks, whose development over time is frequently studied. Interesting graph-theoretic properties include the dynamics of community structures [11], evolution of paths lengths [7], and the development of user centrality and roles over time [5]. A good understanding of these properties is a prerequisite for the design of distributed online social networks and the analysis of communication flow and influence in social networks. Another example for dynamic systems is the field of transportation networks. Well-known problems in this area are the computation of dynamic shortest paths [6] as well as efficient time table queries for train schedules [14]. Solutions are directly applicable to personal navigation systems and route planning for freight-traffic.

A common approach for analyzing dynamic systems is to represent the changes to the network as a stream of updates. This data stream is modeled as an infinite sequence of events [1], where a random access to the updates is not possible. Hence, specialized algorithms for computing graph-theoretic properties in this setting are required. They must update the values under consideration based on the incoming updates to the graph instead of re-computing them since this would require an enormous overhead [12]. Many algorithms for the computation of graph-theoretic properties in dynamic networks have been developed, including connectivity measures [9], clustering coefficients [12], and shortest paths [8]. They can be used to gain insights into the evolution of specific properties of the system under investigation.

The performance and memory footprint of computing a specific property for a dynamic system highly depends on the input data, the chosen algorithm, and the data structures used for the graph representation. Many algorithms use graph representations in form of adjacency lists. Others require specific data structures like an adjacency matrix. Some algorithms even depend on the ordering of adjacency lists or the structure of the adjacency matrix. In all cases, the choice of data structures used to represent the graph topology has a big impact on the memory consumption and the runtime of algorithms. Therefore, the implementation of the used data structures must be interchangeable. This allows users to select a combination of data structures and algorithms that is suited best for a specific scenario. Thereby, the best trade-off between runtime and memory consumption can be used for every scenario.

Most existing graph-theoretic tools and frameworks like Pajek [4], Pregel [10], or SNAP [2] are designed for the analysis of static network topologies only and cannot easily be adapted to analyze dynamic graphs. With Stinger [3], Bader et al. introduced a dynamic graph structure that allows for efficient and parallel updates. While it is considered to be highly efficient, it only provides the underlying data structures and no framework for the computation of graph-theoretic properties on top. The Stanford Network Analysis Platform [1] by Leskovec is a library that also supplies data structures for dynamic graphs. In contrast to Stinger, it supports a large set of

[1] http://snap.stanford.edu

dynamic graph metrics but also lacks a surrounding framework for the comparison of different algorithms. Also, exchanging the basic graph data structures is not intended and an analysis of their impact on an algorithm's performance therefore not possible.

In this paper, we discuss the basics for building a framework for the graph-theoretic analysis of dynamic networks. From a general workflow, applicable to the stream-based analysis of dynamic systems, we deduce a set of requirements for such a framework. We propose a framework architecture that fulfills these requirements. Based on this architecture, we describe the components of the Dynamic Network Analyzer (DNA), a first shot at developing the desired framework.

The remainder of this paper is structured as follows: We discuss preliminaries and introduce our terminology in Section 2.. We outline requirements for a framework for the graph-theoretic analysis of dynamic networks in Section 3.. In Section 4., we present the architecture and components of the *Dynamic Network Analyzer* (DNA), a first implementation of such a framework we deveoped. Finally, we summarize the paper in Section 5. and give an outlook on the next steps for the development of DNA.

2. PRELIMINARIES AND TERMINOLOGY

In this Section, we present our terminology for metrics, values, graphs, and their components. Then, we discuss updates that reflect the dynamic changes of the system under consideration and how they are presented during the analysis, either as a stream or in form of batches.. Based on a general workflow for the analysis of dynamic networks, we describe how updates can be used to update graph data structures and values computed by metrics.

2.1. Metrics, values, and metric collections

A metric is a quantitative measure of a graph-theoretic property. Its result can be a single value, a list of values, a distribution, or any other kind of data. Common examples are the average node degree, the local clustering coefficients, and the hop plot. In the remainder of this paper, we do not differentiate between data types and simply refer to them as values.

Many metrics investigate similar properties of a graph. An example are triangle-based metrics like the clustering coefficient, transitivity, and local clustering coefficients. All three metrics require an algorithm to count the number of triangles in the whole network or connected to each node. Such closely related metrics can easily be computed together, in most cases without overhead compared to the computation of a single metric. Therefore, we assume that similar metrics are grouped together to reduce their computation complexities. We denote such a set of related metrics as metric collection.

2.2. Graphs and weights

A graph $G_i = (V_i, E_i)$ at time i consists of a set of nodes V_i and a set of edges E_i representing the connections between these nodes. Depending on the type of network represented by the graph, the edges can be directed or undirected. Loops can be allowed or prohibited as well as the existence of parallel edges.

To carry further information, weights can be added to nodes and edges respectively, assigning a value c to each corresponding entity. Assume without loss of generality that $c \in \mathbb{R}$. Then, weight functions for nodes and edges are defined as follows:

$$w_i^V : N_i \to \mathbb{R} \qquad w_i^E : E_i \to \mathbb{R}$$

As the graph changes over time, the weights can change as well, rendering the functions w_i^N and w_i^E dependent on the timestamp i.

2.3. Updates

Each change that occurs in a dynamic system between time $t = i$ and $t = j, i < j$ is described as an update $u \in U_{i,j}$. To reflect all potential changes that may happen to a system modeled as a weighted graph, six different update types have to be considered: Adding or removing a node ($u^{V_+} = v \notin V_i$, $u^{V_-} = v \in V_i$), adding or removing an edge ($u^{E_+} = e \notin E_i$, $u^{E_-} = e \in V_i$), and changing the weight of a node or edge ($u^{V_w} = (v,c)$, $u^{E_w} = (e,c), v \in V_j, e \in E_j, c \in \mathbb{R}$).

2.4. Stream of updates

In general, the aforementioned updates arrive in a continuous data stream. Especially when investigating the properties of a live systems as they happen, single updates can arrive one at a time, potentially from different sources. Hence, one possibility to analyze dynamic systems is to consider the stream of updates one at a time.

2.5. Batch of updates

In some situations, it may be preferable to apply the updates in batches instead of importing each update individually. It allows for the parallelization of update processing and thereby can speed up the analysis. In addition, a batch $U_{i,j}$ can be pre-processed to remove redundant updates like, e.g., the addition and removal of the same edge in a single batch.

This potential performance improvement comes at the price of a decreased measurement granularity since changes are not considered one at a time. Also, it may increase the computation complexity since a parallel application of updates can lead to dependencies between certain updates, which has to be handled additionally.

In the remainder of this paper, we only use the notion of batches. Applying a stream of updates is basically the same as applying batches that contain only a single element. Note that

a stream can simply be transformed into batches by grouping updates as they arrive until a specified batch size is reached or after a pre-defined time has passed.

2.6. Workflow for the application of updates

The updates of each batch are used to adapt the graph data structure and update the values of metrics. The overall procedure of the application of batches is shown in Figure 1. It

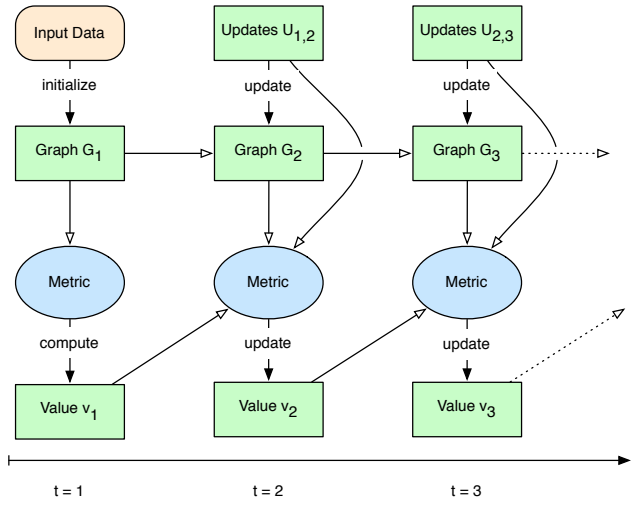

Figure 1: Workflow for analyzing dynamic networks

also represents the general workflow applied for the graph-theoretic analysis of dynamic networks.

At the beginning of an analysis ($t = 1$), the data structures for representing the graph are initialized based on some input data. This instance of the graph (G_1) is used as input for a metric to compute the target value v_1.

At time $t = 2$, the batch $U_{1,2}$ with all changes that occurred since $t = 1$ is applied: The data structures are updated by adding new nodes and edges, removing obsolete ones, and updating weights in the data structures from G_1, transforming the graph into G_2. In addition, the metric processes the updates to adapt the target value according to the given changes. As input, this adaptation uses the graph G_2, the old value v_1, the batch $U_{1,2}$, and possibly auxiliary information that the metric maintains to enable the incremental updates.

For every new batch $U_{i,i+1}$, the same procedure is executed, effectively transforming the graph G_i into G_{i+1} and updating the value v_i to v_{i+1} based on the batch $U_{i,i+1}$, the graph G_{i+1}, the value v_i and auxiliary information maintained by the metric.

2.7. Data structure updates

The batch $U_{i,j}$ represents the set of all updates that occurred between timestamps i and j, i.e., the union of all node/edge additions/removals and weight changes:

$$U_{i,j} := U_{i,j}^{V_+} \cup U_{i,j}^{V_-} \cup U_{i,j}^{E_+} \cup U_{i,j}^{E_-} \cup U_{i,j}^{V_w} \cup U_{i,j}^{E_w}$$

Applying this batch of updates to the graph G_i then yields the network's topology at timestamp j as follows:

$$\begin{aligned} V_j &:= (V_i \cup U_{i,j}^{V_+}) \setminus U_{i,j}^{V_-} \\ E_j &:= (E_i \cup U_{i,j}^{E_+}) \setminus (U_{i,j}^{E_-} \cup (U_{i,j}^{V_-} \times V_i) \cup (V_i \times U_{i,j}^{V_-})) \end{aligned}$$

Analogously, the weights of nodes and edges at timestamp j can be expressed as follows:

$$w_j(v/e) := \begin{cases} w^* & (v/e, w^*) \in U^{E_w/V_w} \\ w_i^{V/E}(v/e) & v/e \in V_i/E_i \\ - & \text{otherwise} \end{cases}$$

2.8. Metric updates

During the application of a batch (cf. Figure 1), the value of a metric is updated based on the batch of changes, the graph, the old value, and auxiliary information stored by the metric. It is important to distinguish the time when the update of a value is triggered in relation to the update process of the graph structures.

Some metrics perform these updates based on the graph structure after the batch was applied, i.e., they take G_j and $U_{i,j}$, as for simplicity stated above. Others require the graph before the application of the updates as input, i.e., G_i and the batch $U_{i,j}$.

While processing in batches is beneficial in some cases, it might not be feasible in others. Therefore, there are metrics that require the application of each update at a time and hence need to preprocess each update at a time before or after it was applied to the graph.

For some metrics, an incremental update might not be feasible because the storage required for maintaining auxiliary information exceeds the available memory. It is also possible that no algorithm for an efficient batch application is known. Such metrics have to be re-computed for the whole graph after the application of each batch to the data structures.

To map these different requirements, we distinguish five modes of metric updates:

1. **Before Batch** (*BB*), i.e., input G_i and $U_{i,j}$
2. **After Batch** (*AB*), i.e., input G_j and $U_{i,j}$
3. **Before Single** (*BS*), i.e., input $G_{i,j}^{k-1}$, $U_{i,j}$, and $u_k \in U_{i,j}$
4. **After Single** (*AS*), i.e., input $G_{i,j}^{k}$, $U_{i,j}$, and $u_k \in U_{i,j}$
5. **Re-Computation** (*RC*), i.e., input G_j

Here, $G_{i,j}^k$ denotes the graph at timestamp i after the application of the first k updates of the batch U_i, j.

Note that a metric can apply updates of different modes in order to adapt a value.

Take for example the computation of the largest connected component of a network. Well-known algorithms for updating this metric are based on single updates, hence the *BS* or *AS*

mode would be used. In case of metrics that update the value independently of each other, the *BB* or *AB* modes would be preferred since they allow for the parallelization of the updates in the batch. A simple re-computation of the degree distribution would be performed in *RC* mode.

3. FRAMEWORK REQUIREMENTS

In this Section, we give a set of requirements that a framework must meet in order to comply with the workflow shown in Figure 1 and allow for the analysis of various systems modeled using different types of graphs.

3.1. Supported graph types

A framework for the graph-theoretic analysis of dynamic networks should support the basic and most commonly used graph types: directed and undirected graphs with or without loops as well as multigraphs. It should also be possible to add weights of arbitrary type to nodes and edges.

3.2. Implementation of new metrics

A well-defined interface should allow the easy implementation of new metrics and approaches for their computation. It must include possibilities to specify actions for each mode of computation as described in Section 2.8.. This should enable the combination of multiple metrics in the analysis of a specific system.

3.3. Implementation of new data structures

As already mentioned, the data structures used for representing and accessing the graph information have a high influence on the performance of computing the investigated metrics. Therefore, it is crucial for a framework to allow for the implementation of various data structures for storing the graph information for the different types of graphs.

3.4. Combination of metrics and data structures

When creating a framework for the graph-theoretic analysis of dynamic networks, the main focus is to allow for the simple implementation and evaluation of various metrics as well as data structures for representing the system graph. Therefore, it should be easy to combine metrics with different data structures to find the desired trade-off between computation performance and memory requirements. Also, arbitrary graph topologies and update combinations should be supported to allow for the evaluation of a diverse set of scenarios.

3.5. Classes of input data

A framework should support three classes of input data for initializing the graph as well as creating updates: online, offline, and generated.

Online data describes a stream of data that arrives at the system in real time like a stream of tweets. From such input data, the initial graph topology as well as updates can be modeled. This is necessary to support the analysis of live data streams and networks modeled from them using the framework.

As offline data, we describe input data that is already completely available at the beginning of the analysis like e.g., consecutive snapshots from an online social network. After reading the initial graph from a file, updates for the transition to the next snapshot are read and applied from files as well. As a pre-processing option, it should also be possible to create updates from two given input graphs instead of providing them explicitly. The support of offline data allows the analysis of collected network snapshots and traces from already recorded systems.

Generated data is not given in form of actual structural information but as a specification how the initial graph as well as batches of updates should be generated. An example would be a graph generator that initializes a random graph with a given number of nodes and edges as well as a batch generator that creates random edge additions and removals. Such graph and batch generators allow for the test and evaluation of metrics and their implementation in well-understood network models and is therefore crucial in the development process of new algorithms for the computation of metrics.

3.6. Aggregation of multiple runs

In order to achieve statistically significant results, a framework must support the execution and aggregation of multiple runs of the same experiment. Each of those runs should be reproducible.

3.7. Result visualization

Lastly, the results of the values computed by the metrics as well as statistics about runtime and memory consumption of the respective setup should be visualized to quickly show the impact of different settings and investigate the properties of the analyzed system.

4. ARCHITECTURE AND COMPONENTS OF DNA

In this Section, we describe the architecture and components of the *Dynamic Network Analyzer* (DNA), our first implementation of a Java-based framework for the graph-theoretic analysis of dynamic networks. While the development of DNA is still a work-in-progress, it already meets most of the requirements stated in Section 3..

The Architecture of the *Dynamic Network Analyzer* is shown in Figure 2. It provides interfaces and basic implementations for graphs, updates, and batches. The initialization of graphs is provided by the graph generator component while the update generator component supplied batches of updates.

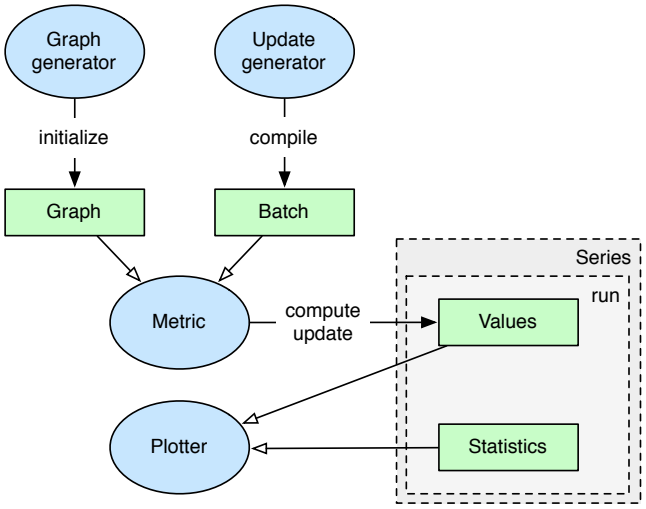

Figure 2: Architecture of the *Dynamic Network Analyzer*

The metric components allow for the computation of values grouped into metric collections. The series components is a wrapper for generating multiple runs of the same scenario. For each of these runs, the computed and updated values for each batch application are stored together with statistics about runtimes and memory requirements. These results are visualized using the plotter component.

4.1. Graphs and data structures

As its basic component, DNA provides data structures for storing, updating, and querying graphs. So far, only weighted, directed graphs have been implemented. All data structures support the assignment of weights to nodes and edges. For simplicity, updates for adding or removing nodes have been omitted in this initial version to simplify the update process of graph data structures.

Complementary to basic implementations of graphs, the graph component allows for the addition and use of further ones through a well-defined interface.

4.2. Graph generator

The graph generator component provides the initialization of the graph. In its current implementation, it can read the graph from a file or generate it from scratch based on a specification.

4.3. Update generator and batches

The update generator provides the updates of graph structure and weights to the system. Currently, it is only supported to read updates from files or generate them from a specification. It can either pass down each update separately as a stream or combine multiple updates into batches of predefined size.

4.4. Metrics

The metric component supplies interfaces for the implementation of arbitrary metrics that comply with the aforementioned five modes of updating their values. This implies the support for metrics that apply any combination of the five value update modes: *BB*, *AB*, *BS*, *AS*, and *RC*.

In addition, the metric component supplies basic means to compare the values computed by different implementations of the same metric. This enables the verification of potentially complex update-based implementations using simple re-computations. Also, values of an approximation can automatically be compared with their exactly computed counterpart.

Please note that the metric component in DNA can combine the computation of multiple metrics in a single implementation. Thereby, it reflects the concept of metric collections as introduced in Section 2..

4.5. Series

Runs have to be repeated multiple times and their results must be aggregated. The series component was developed to serve this purpose. Taking a graph generator, an update generator, and a list of metrics as input, the series performs multiple runs of the same setup. The results and statistics for each batch application of every run are stored in a predefined folder structure.

The procedure applied during the generation of a series is outlined in Algorithm 1. As input, it takes a graph generator

```
Input: GraphGenerator gg, UpdateGenerator ug, Metric m, int
       runs, int batches
Output: Series data
foreach r ∈ [1..runs]do
    Graph g = gg.generate(); ▷ graph initialization
    m.init(g); ▷ initial computation
    m.write(run.$r/batch.0/metric.0/)
    Runtimes.write(run.$r/batch.0/runtimes)
    Statistics.write(run.$r/batch.0/statistics)
    foreach b ∈ [1..batches]do
        Batch batch = ug.generate(); ▷ generate batch
        m.updateBeforeBatch(g, batch); ▷ BB mode
        foreach Update u : batch do
            m.updateBeforeUpdate(g, b, u); ▷ BS mode
            g.apply(u); ▷ update application
            m.updateAfterUpdate(g, b, u); ▷ AS mode
        end
        m.updateAfterBatch(g, batch); ▷ AB mode
        m.reComputation(g); ▷ RC mode
        m.write(run.$r/batch.$b/metric.0/)
        Runtimes.write(run.r/batch.b/runtimes)
        Statistics.write(run.r/batch.b/statistics)
    end
end
Aggregation.write(aggr/);
```

Algorithm 1: Series generation

and an update generator that supplies the initial graph as well as batches of updates to it. In addition, the metric to be evaluated is given as well as the desired number of runs and batches to be executed. For brevity, only a single metric is given as input in this procedure. Normally, a list of metrics is given, all of which are computed in parallel during the execution.

For each run, a new graph instance is initialized by the graph generator. The metric component computes its values for the initial graph which are written to the initial output folder, together with runtimes and statistics. Afterwards, the given number of batches is generated by the update generator. Each batch is applied to the graph and the metric is updated accordingly. In order to comply with the five modes for updating metric values described before, methods for updating the metric before, during, and after updating the graph structures are called. Please note, that it depends on the metric which methods are actually implemented and hence used to update the computed values. Finally, the computed metric values are written to the filesystem together with runtimes and statistics.

After each run is executed, the generated data is aggregated and written to a dedicated directory. The data of each run is stored in a separate folder (*run1*, *run2*, ...). For every batch, the computed values and statistics about its application are stored in a corresponding folder (*batch.1*, *batch.2*, ...). The values of the metric computations after the initialization of the graph are stored in *batch.0*. Inside of these batch folders, the values of each metric collection after performing the batch's updates are stored together with runtimes and statistics.

4.6. Plotter

The plotter component uses the data produced by a series to generate plots for visualizing its results. This includes plots of the metric values and their development over time, runtimes of metric and graph updates, and other statistics computed during the application of each batch.

In the current implementation, Gnuplot [13] is used to generate plots from the generated results and statistics.

5. SUMMARY

In this paper, we outlined the basics for building a framework for the graph-theoretic analysis of dynamic networks. First, we described a general workflow for the analysis of dynamic networks in a scenario based on data streams as input for the changes in the system. Then, we listed a set of requirements a framework must to meet in order to comply with this workflow and allow for a straight-forward development of new algorithms and data structures and the analysis of arbitrary input data. Based on these requirements, we developed a first prototype of such a framework, the *Dynamic Network Analyzer* (DNA).

So far, DNA already meets most of the requirements for the desired framework. While not all considered graph types and update mechanisms are yet implemented, well-defined interfaces for data structures and metrics have already been developed. Different implementations of metrics and data structures can already be freely combined. Currently, only offline and generated input data is supported, leaving the integration of online data streams as future work. The series component already fulfills the basic requirements regarding the execution and aggregation of multiple runs. A global integration of seed-based pseudo-random behavior of generators ensures the reproducibility of experiments. The visualization of results is covered by the plotter component.

Currently, we are extending DNA's data structures to allow for the addition and removal of nodes as part of the update process. We are also working on the integration of online data sources to the graph and update generator components. Also, the pre-processing of multiple graph snapshots into initial graph and batches of updates is an ongoing task.

Furthermore, we are considering the integration of interfaces to graph databases as a data structure into DNA. In addition, we are investigating possibilities to add general capabilities and interfaces for the parallelization of metric computations as present in many frameworks and libraries for the analysis of static network topologies.

*Bibliography

[1] Babcock et al. Models and issues in data stream systems. In *ACM SIGMOD-SIGACT*, 2002.

[2] Bader et al. Snap, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Parallel and Distributed Processing*, 2008.

[3] Bader et al. Stinger: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Technical Report*, 2009.

[4] Batagelj et al. Pajek-program for large network analysis. *Connections*, 1998.

[5] Braha et al. Time-dependent complex networks: Dynamic centrality, dynamic motifs, and cycles of social interactions. In *Adaptive Networks*. 2009.

[6] Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Journal of the Transportation Research Board*, 1998.

[7] Kossinets et al. Empirical analysis of an evolving social network. *Science*, 2006.

[8] Likhachev et al. Anytime dynamic a*: An anytime, replanning algorithm. In *International conference on automated planning and scheduling (ICAPS)*, 2005.

[9] Madduri et al. Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In *Parallel & Distributed Processing*, 2009.

[10] Malewicz et al. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of data*, 2010.

[11] Mucha et al. Community structure in time-dependent, multiscale, and multiplex networks. *Science*, 2010.

[12] Edigerand others. Massive streaming data analytics: A case study with clustering coefficients. In *Parallel & Distributed Processing*, 2010.

[13] Racine. gnuplot 4.0: a portable interactive plotting utility. *Journal of Applied Econometrics*, 2006.

[14] Stølting et al. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 2004.