The University of Manchester

Dynamic Network Analysis in Julia

Weijian Zhang

September 2015

MIMS EPrint: **2015.83**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://www.manchester.ac.uk/mims/eprints

And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

Dynamic Network Analysis in Julia

Weijian Zhang, School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK
weijian.zhang@manchester.ac.uk

***Abstract*—This paper introduces EvolvingGraphs, a Julia software package for the creation, manipulation, and study of dynamic networks. We describe the underlying model of EvolvingGraphs and discuss the implementations of components and centrality in the case of dynamic networks. We make particular use of the parameterizable type system and multiple dispatch in Julia. Users can work on a variety of graph types with nodes and timestamps of any Julia type.**

I. Introduction

We describe EvolvingGraphs[1], a Julia software package for analyzing dynamic networks. A dynamic network is network in which the interactions among a set of elements change over time. Examples of dynamic networks include a network of mobile phone users interacting through messaging and the spread of diseases in communities. It is natural to model a dynamic network by an evolving graph G, defined as a sequence of static graphs $\{G_1, G_2, \ldots, G_n\}$, where $G_i = (V(i), E(i))$ is a snapshot of the evolving graph G at timestamp i. For example, Figure 1 shows an evolving graph with 3 timestamps t_1, t_2, t_3, where a green shaded circle denotes an active node (see Section II) and a pink circle denotes an inactive node.

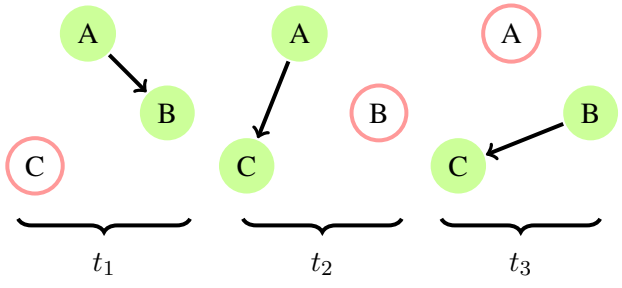

Fig. 1: An evolving graph with 3 timestamps.

Julia [1] is a high-level, high-performance dynamic programming language for technical computing. It takes advantage of LLVM-based [2] just-in-time (JIT) compilation to approach the performance of statically-complied languages like C, yet allows programmers to write clear, high-level code that closely resembles mathematical formulas. EvolvingGraphs makes particular use of Julia's multiple dispatch combined with its type system. Since user defined types are first class in Julia, it makes sense to implement new graph types to design user-friendly interfaces.

EvolvingGraphs is designed to have a similar interface to standard static graph packages such as Python's NetworkX[2] or Julia's Graphs [3]. To get a taste of EvolvingGraphs, we may consider representing the evolving graph of Figure 1:

```
> g = evolving_graph(Char, String)
> add_edge!(g, 'A', 'B', "t1")
> add_edge!(g, 'A', 'C', "t2")
> add_edge!(g, 'B', 'C', "t3")
> nodes(g)
3-element Array{Char,1}:
 'A'
 'B'
 'C'
> edges(g)
3-element Array{TimeEdge{V,T},1}:
 TimeEdge(A->B) at time t1
 TimeEdge(A->C) at time t2
 TimeEdge(B->C) at time t3
```

where the arguments of `evolving_graph` indicate the node type and timestamp type respectively.

This paper is organized as follows. In Section II, we introduce a node-active model for evolving graphs, including the definition of temporal path and temporal distance. In Section III, we consider the type system of EvolvingGraphs and discuss ways to represent evolving graphs. The concept of connected components in evolving graphs is introduced in Section IV. A breadth first search (BFS) based implementation for finding weakly connected components is also discussed. We consider a generalization of the Katz centrality in Section V and provide examples of usage in Section VI.

[1] http://evolvinggraphsjl.readthedocs.org/

[2] https://networkx.github.io/

[3] http://graphsjl-docs.readthedocs.org/

II. Node-Active Model

We assume $G = \{G_1, G_2, \ldots, G_n\}$ is a directed evolving graph, where each edge e is of the form (v_i, v_j, t_k), indicating a link from node v_i to node v_j at timestamp t_k. Unlike in most existing models [3] [4] [5] [6], in which the node set is assumed to be fixed throughout time, in our model (a) nodes are time-dependent and are changing over time; (b) nodes are present only if they are connected by edges. We say a node v at timestamp t, denoted by (v, t), is *active* if v is connected to at least one other node at timestamp t. For example, in Figure 1 the following nodes are active: $(A, t_1), (B, t_1), (A, t_2), (C, t_2), (B, t_3), (C, t_3)$. We disregard the inactive nodes when we study and analyze evolving graphs.

Definition 2.1 (Temporal path): We define a *temporal path* $p((v_i, t_1), (v_j, t_n))$ between active nodes (v_i, t_1) and (v_j, t_n) to be an ordered sequence of active nodes (without repetition) $\{(v_i, t_1), (v_{i+1}, t_2), \ldots, (v_j, t_n)\}$ such that $t_1 \leq t_2 \leq \cdots \leq t_n$ and $((v_h, t_k), (v_{h+1}, t_{k+1}))$ is an edge at timestamp t_k if $t_k = t_{k+1}$ otherwise we have $v_h = v_{h+1}$. A *shortest temporal path* is a temporal path with the least number of unique nodes.

For example, there are two temporal paths from (A, t_1) to (C, t_3) in Figure 1:

1) $(A, t_1) \to (A, t_2) \to (C, t_2) \to (C, t_3)$
2) $(A, t_1) \to (B, t_1) \to (B, t_3) \to (C, t_3)$

The first one is the shortest temporal path since it passed 2 nodes A and C while the second one passed 3 nodes.

Definition 2.2 (Temporal distance): We define the *spatial length* of a temporal path $p((v_i, t_1), (v_j, t_n))$ to be the number of unique nodes in p. The *shortest temporal distance* $d((v_i, t_1), (v_j, t_n))$ between (v_i, t_1) and (v_j, t_n) is the spatial length of the shortest temporal path.

For example, the shortest temporal distance between (A, t_1) and (C, t_3) in Figure 1 is 2.

Definition 2.3 (Temporal connectedness): If there exists a temporal path from node (v_i, t_m) to node (v_j, t_n), we say (v_i, t_m) and (v_j, t_n) are *temporally connected*. We say node v_i and v_j are temporally connected if (v_i, t_m) and (v_j, t_n) are temporally connected for some timestamps t_m, t_n.

III. Representing Evolving Graphs

The graph type hierarchy in EvolvingGraphs is shown in Figure 2.

The root of all graph types is `AbstractGraph`. It has two children: `AbstractStaticGraph` (the abstract type of all static graphs) and `AbstractEvolvingGraph` (the abstract type of all evolving graphs). For both static and evolving graphs, we focus on directed graphs and model undirected graphs using directed graphs with twice as many edges. There are two kinds of static graphs in EvolvingGraphs: `TimeGraph` represents an evolving graph at a specified timestamp; `AggregatedGraph` represents a static graph constructed by aggregating an evolving graph, i.e., all the edges between each pair of nodes are flattened in a single edge. For example, the aggregated graph of Figure 1 is shown in Figure 3.

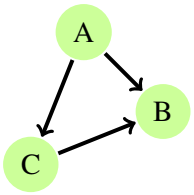

Fig. 3: The aggregated graph of Figure 1.

All static graphs in EvolvignGraphs are represented by adjacency lists. For evolving graphs, we consider three types: `EvolvingGraph`, `MatrixList` and `IntEvolvingGraph`, which are represented by (a) edge lists, (b) adjacency matrices (c) a mixture of adjacency lists and edge lists respectively. Other evolving graph types are variants of one of the three types.

The edge lists representation of G is specified by

1) the number of nodes n;
2) the list of edges in G, given as a sequence of ordered tuples (v_i, v_j, t_n), which represents an edge from v_i to v_j at timestamp t_n.

The evolving graph of Figure 1 can be represented as follows.

1) $n = 3$;
2) $(A, B, t_1), (A, C, t_2), (B, C, t_3)$.

We can also represent an evolving graph by a list of adjacency matrices $\{A_1, A_2, \ldots, A_m\}$. Let $V = \cup_i V(i)$ be the union of all the node sets $V(i)$. Then each A_k is a $|V| \times |V|$ matrix where the (i, j) entry is equal to 1 if and only if there exists an edge from the ith element of V to the jth element of V at timestamp k, and 0 otherwise. For the evolving graph of Figure 1, we have $V = \{A, B, C\}$ and

$$A_1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, A_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}.$$

Finally, we can represent an evolving graph G as a mixture of adjacency lists and edge lists:

1) the number of edges e;

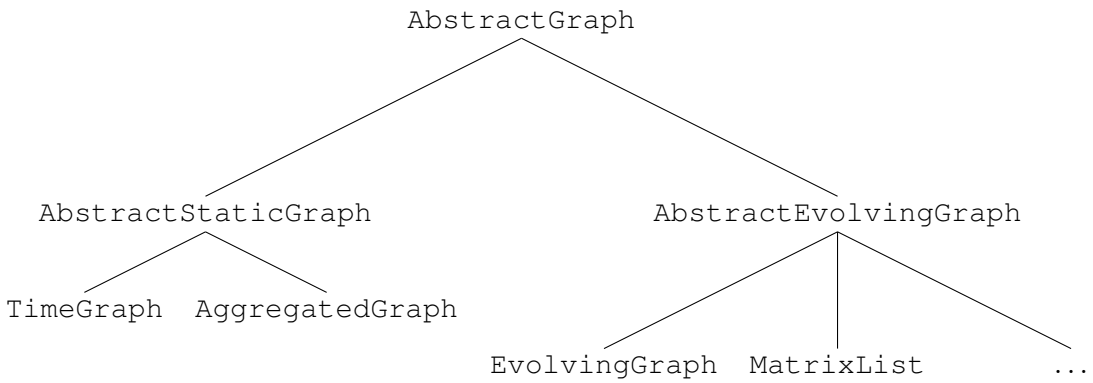

Fig. 2: Graph type hierarchy.

2) the list of active nodes, given as a sequence of ordered pairs (v_i, t_n), which represents an active node v_i at timestamp t_n;
3) the list of timestamps;
4) m lists $E_1, E_2, \ldots, E_m$, where E_i contains all the edges at timestamp i;
5) the lists of out-neighbours of each active node.

Recall that in a static graph the out-neighbours of a node v are all the nodes pointed by node v. For evolving graphs, we define the out-neighbours of a node v at time t to be the set of all active nodes pointed by node v at time t and the active node v itself at time t_i, where $t_i > t$. For example, the out-neighbours of (A, t_1) in Figure 1 are (B, t_1), (A, t_2). The evolving graph of Figure 1 may be represented as follows:

1) $e = 3$;
2) $(A, t_1), (B, t_1), (A, t_2), (C, t_2), (B, t_3), (C, t_3)$;
3) t_1, t_2, t_3;
4) $E_1 : (A, B, t_1); E_2 : (A, C, t_2); E_3 : (B, C, t_3)$;
5) $(A, t_1) : (B, t_1), (A, t_2)$;
   $(B, t_1) : (B, t_3)$;
   $(A, t_2) : (C, t_2)$;
   $(C, t_2) : (C, t_3)$;
   $(B, t_3) : (C, t_3)$;
   $(C, t_3) : \emptyset$.

Basic graph functions, like checking directedness (`is_directed`) or finding out-neighours (`out_neighbors`) are defined for both static and evolving graphs. The implementations of these functions are dispatched based on the type of graph.

IV. Components

In this section, we discuss an algorithm for computing weakly connected components in EvolvingGraphs. We start by introducting the notion of weak connectedness.

Definition 4.1 (Weak connectedness): We say two nodes (v_i, t_m) and (v_j, t_n) are *weakly connected* if information can flow from (v_i, t_m) to (v_j, t_n), i.e., (v_i, t_m) and (v_j, t_n) are temporally connected.

Note that our notion of connectedness is reflexive and transitive but not symmetric. The order of time breaks the symmetry. In fact, (v_i, t_m) and (v_j, t_n) are weakly connected only if $t_m \leq t_n$. We can think of an edge from v_i to v_j at timestamp t_n as information flows from v_i to v_j at timestamp t_n. In modern technology communication, a message can be received immediately after it is sent. Thus we assume that the information flow duration is 0, which is different from models like that in [7]. Given an active node (v, t), we form the information passing tree by collecting all the temporal paths that start at (v, t).

Definition 4.2 (Information source): An *information source* is a root (an active node) of the information passing tree. We say an information source (v, t) is a *global information source* if (v, t) is an information source and no node is temporally connected to (v, t).

For example, (A, t_1) is a global information source in Figure 1. Using the notion of weak connectedness and information source, we define the *weakly connected components* of an evolving graph G as follows.

Definition 4.3 (Weakly connected components): A *weakly connected component* associated with a node (v_j, t_n) is the set of nodes in G which are weakly connected by the same information source as (v_j, t_n).

A node v_i in an evolving graph can belong to multiple components but the partition is unique. For example, suppose at timestamp t_1, A passed a message to B and C passed a message to D. At timestamp t_2, B passed a message to D (see Figure 4). Then there are two information sources in this evolving graph: (A, t_1) and (C, t_1) and the weakly connected components are:

- $(A, t_1), (B, t_1), (B, t_2), (D, t_2)$;
- $(C, t_1), (D, t_1), (D, t_2)$.

Note (D, t_2) belongs to both components.

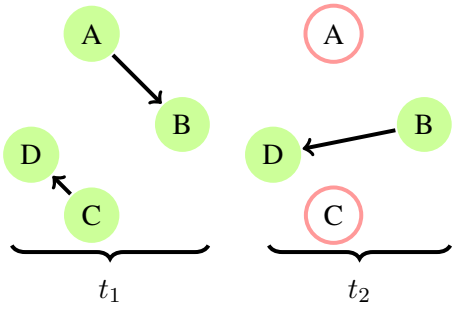

Fig. 4: An evolving graph with 2 time windows.

To explore an evolving graph, we need to pay attention to the order of time. In particular, we can not go to a node at a previous timestamp. Recall that the out-neighbours of a node (v,t) are the active nodes pointed by node v at timestamp t and the active node v itself at timestamp t_i, where $t_i \leq t$. Using this notion of out-neighbours, we can extend the BFS algorithm for the case of evolving graphs. Here is the BFS algorithm in EvolvingGraphs:

```
function breath_first_visit(
    g::AbstractEvolvingGraph, s::Tuple)
    level = Dict(s => 0)
    i = 1
    fronter = [s]
    reachable = [s]
    while length(fronter) > 0
        next = Tuple[]
        for u in fronter
            for v in out_neighbors(g, u)
                if !(v in keys(level))
                    level[v] = i
                    push!(reachable, v)
                    push!(next, v)
                end
            end
        end
        fronter = next
        i += 1
    end
    reachable
end
```

This algorithm finds all the reachable nodes from a given starting node (s,t_1). Let t_1 be the earliest timestamp of G and let $\tilde{V}$ be the set of all the active nodes of G and $E = \cup_t E(t)$ be the set of all edges of G. Since the algorithm explores the out-neighbours of each (reachable) active node, the computational cost is

$$O\left(\sum_{(v,t)\in\tilde{V}} Adj[(v,t)]\right) = O(|E|).$$

To determine the weakly connected components of G, we need to find all the global information sources and then use BFS to find all the reachable nodes from these global information sources. We may use the function `weakly_connected_components(g)` in EvolvingGraphs to discover the weakly connected components of an evolving graph `g`. Let $\bar{V}$ be the set of global information sources. The computational cost for finding the weakly connected components is $O(|\bar{V}||E|+|\bar{V}|^2/2)$, where calling BFS has complexity $O(|\bar{V}||E|)$ and checking connected components has complexity $O(|\bar{V}|^2/2)$.

V. Katz Centrality

Let A be an adjacency matrix of a static graph G. The Katz centrality rating [8] of a node i is the ith row sum of $(I-\alpha A)^{-1}$, where $\alpha < 1/\rho(A)$, the spectral radius of A. The Katz centrality vector r can be computed by solving

$$(I-\alpha A)r = \mathbf{1},$$

where $\mathbf{1}$ is a vector of ones. It follows from the analysis on static networks that the centrality matrix C_n of an evolving network [4] can be formulated as

$$C_n = (I-\alpha A_1)^{-1}(I-\alpha A_2)^{-1}\cdots(I-\alpha A_n)^{-1}, \quad (1)$$

where $\{A_1, A_2, \ldots, A_n\}$ are the corresponding adjacency matrix representations of the evolving graph $G = \{G_1, G_2, \ldots, G_n\}$ and $\alpha < 1/\max_k \rho(A_k)$. The (i,j) entry of the matrix C_n gives a weighted count of the number of dynamic walks from node i to node j. The broadcast and receive communicability vectors are

$$C_n\mathbf{1} \quad \text{and} \quad C_n^T\mathbf{1},$$

respectively. We can compute the broadcast vector using the following algorithm in Julia:

```
function katz_centrality(
g::AbstractEvolvingGraph, alpha::Real)
    n = num_nodes(g)
    ts = timestamps(g)
    v = ones(Float64, n)
    A = spzeros(Float64, n, n)
    spI = speye(Float64, n)
    for t in ts
        A =  spmatrix(g,t)
        v = (spI - alpha*A)\v
        v =  v/norm(v)
    end
```

```
    return v
end
```

A short list of graph functions implemented in EvolvingGraphs is shown in Table I. By considering walks that started recently as more important than walks that started a long time age [9], Grindrod and Higham introduce a time-dependent factor $e^{-\beta\Delta t_n}$, $\Delta t_n = t_n - t_0$. A variant of (1) is given as

$$S_n = (I+e^{-\beta\Delta t_n}S_{n-1})(I-\alpha A_n)^{-1}-I, \quad n = 1, 2, \ldots, \tag{2}$$

where $S_0 = 0$, the zero matrix. To see how these two formulae are related, we write $(I - \alpha A_n)^{-1}$ as $(I + \alpha A_n+\alpha^2 A_n^2+\cdots)$ and expand the right-hand side of (2). The function `katz_centrality` in EvolvingGraphs has more input options than the above implementation. In particular, we can specify the following parameters:

- α: a real-valued scalar which controls the influence of long distance walks;
- β: a real-valued scalar which controls the influence of old walks;
-
  - `mode = :broadcast` (by default) generates the broadcast centrality vector;
  - `mode = :receive` generates the receiving centrality vector;
  - `mode = :matrix` generates the communicability matrix.

VI. Examples of Use Cases

Suppose we model a network of online social users interacting through messaging by the evolving graph of Figure 5. Each node i represents a user in the network and an edge from node i to node j at timestamp t represents user i sent a message to user j during timestamp t and $t + 1$. Let us first generate this evolving graph:

```
> g = evolving_graph(Int, String);
> add_edge!(g, 1, 2, "t1");
> add_edge!(g, 1, 3, "t2");
> add_edge!(g, 4, 5, "t2");
> add_edge!(g, 2, 3, "t3");
> add_edge!(g, 3, 1, "t3");
> add_edge!(g, 5, 6, "t3");
> g
Directed EvolvingGraph
(6 nodes, 6 edges, 3 timestamps)
```

Now `g` is an evolving graph with 6 nodes, 6 edges and 3 timestamps. We can use the functions `nodes`, `edges` and `timestamps` to have a quick check to see if we have generated the evolving graph correctly:

```
> nodes(g)
6-element Array{Int64,1}:
 1
 4
 2
 3
 5
 6
> edges(g)
6-element Array{TimeEdge{V,T},1}:
 TimeEdge(1->2) at time t1
 TimeEdge(1->3) at time t2
 TimeEdge(4->5) at time t2
 TimeEdge(2->3) at time t3
 TimeEdge(3->1) at time t3
 TimeEdge(5->6) at time t3
> timestamps(g)
3-element Array{String,1}:
 "t1"
 "t2"
 "t3"
```

We use the function `weak_connected` to find out if two users "talked" to each other between timestamp t_1 and t_3.

```
> weak_connected(g, 1, 3)
true
> weak_connected(g, 1, 5)
false
```

So user 1 talked to user 3 but not to user 5. We can use the function `weak_connected_components` to detect small communities in the network.

```
> weak_connected_components(g)
2-element Array{Array{Tuple,1},1}:
 Tuple[(1,"t1"),(2,"t1"),(1,"t2"),
 (1,"t3"),(2,"t3"),(3,"t2"),(3,"t3")]

 Tuple[(4,"t2"),(5,"t2"),(5,"t3"),
 (6,"t3")]
```

We see users $1, 2, 3$ form a small community and users $4, 5$ form another small community. At each timestamp, `g` may be represented as an adjacency matrix:

```
> int(matrix(g, "t2"))
6x6 Array{Int64,2}:
 0  0  0  1  0  0
 0  0  0  0  1  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
```

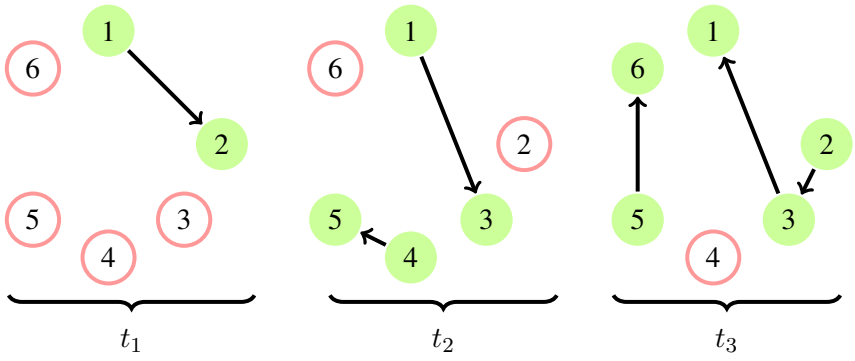

Fig. 5: An evolving graph with 3 time windows.

We can use `katz_centrality` to find out the "important" users in a network. Here users 1 and 4 are the most important users in terms of broadcasting information and users 2 and 3 are the most important users in terms of acting as information receivers.

```
>  katz_centrality(g, 0.2, 0.3,
mode =:broadcast)
 (6,0.0)
 (2,0.27241247410068437)
 (3,0.27241247410068437)
 (5,0.27241247410068437)
 (4,0.32591575357149416)
 (1,1.5259157535714944)
>  katz_centrality(g, 0.2, 0.3,
mode =:receive)
 (4,0.0)
 (5,0.2715964613095785)
 (1,0.32673176636260004)
 (6,0.32673176636260004)
 (3,0.7440089354102629)
 (2,1.0)
```

VII. Conclusion

The software package EvolvingGraphs is written in Julia, a new dynamic programming language for technical computing. We discussed a node-active model for evolving graph and showed a few algorithms implemented in EvolvingGraphs. EvolvingGraphs currently performs all computations serially. In the future, we will design and implement parallel graph algorithms in EvolvingGraphs with the aim of handling extremely large scale dynamic network problems.

Acknowledgment

The author is grateful for many useful comments from Prof. Nicholas J. Higham and Mario Berljafa.

References

[1] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[2] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[3] D. V. Greetham, Z. Stoyanov, and P. Grindrod, "On the radius of centrality in evolving communication networks," *Journal of Combinatorial Optimization*, vol. 28, no. 3, pp. 540–560, 2014.

[4] P. Grindrod, M. C. Parsons, D. J. Higham, and E. Estrada, "Communicability across evolving networks," *Physical Review E*, vol. 83, no. 4, p. 046120, 2011.

[5] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora, "Components in time-varying graphs," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 22, no. 2, p. 023101, 2012.

[6] J. Tang, M. Musolesi, C. Mascolo, and V. Latora, "Temporal distance metrics for social network analysis," in *Proceedings of the 2nd ACM workshop on Online social networks*. ACM, 2009, pp. 31–36.

[7] S. Huang, A. W.-C. Fu, and R. Liu, "Minimum spanning trees in temporal graphs," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 419–430.

[8] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[9] P. Grindrod and D. J. Higham, "A matrix iteration for dynamic network summaries," *SIAM Review*, vol. 55, no. 1, pp. 118–128, 2013.

TABLE I: Examples of graph functions implemented in EvolvingGraphs, where `g` is an evolving graph.

Function name	Description
`is_directed(g)`	returns true if `g` is a directed graph and false otherwise.
`undirected!(g)`	converts a directed graph to an undirected graph.
`num_nodes(g)`	returns the number of nodes in `g`.
`nodes(g)`	returns a list of nodes of `g`.
`edges(g)`	returns a list of edges of `g`.
`timestamps(g)`	returns a list of timestamps of `g`, in ascending order.
`add_edge!(g, v1, v2, t)`	adds an edge from `v1` to `v2` at timestamp `t` to `g`.
`add_graph!(g, tg)`	adds a time graph `tg` to `g`.
`out_neighors(g, (v,t))`	returns the out-neighbours of node `(v,t)` in `g`.
`aggregated_graph(g)`	converts `g` to an aggregated graph.
`issorted(g)`	returns true if the timestamps of `g` are sorted and false otherwise.
`sorttime!(g)`	sorts `g` so that the timestamps of `g` are in ascending order.
`slice!(g, t_min, t_max)`	slices `g` between the timestamp `t_min` and `t_max`.
`matrix(g, t)`	generates an adjacency matrix representation of `g` at timestamp `t`.
`spmatrix(g, t)`	generates a sparse adjacency matrix representation of `g` at timestamp `t`.
`matrix_list(g)`	converts `g` to a list of adjacency matrices represented by `MatrixList`.
`shortest_temporal_path(g, (v1, t1), (v2, t2))`	finds the shortest temporal path from `(v1, t1)` to `(v2, t2)`.
`temporal_connected(g, v1, v2)`	returns true if `v1` are `v2` are temporally connected and false otherwise.
`weak_connected_components(g)`	returns the weakly connected components of `g`.