Efficiently Testing T-Interval Connectivity in Dynamic Graphs☆

Arnaud Casteigts[a], Ralf Klasing[a], Yessin M. Neggaz[a], Joseph Peters[b]

[a]*LaBRI, CNRS, University of Bordeaux, France*
[b]*School of Computing Science, Simon Fraser University, Burnaby, BC, Canada*

Abstract

Many types of dynamic networks are made up of durable entities whose links evolve over time. When considered from a *global* and *discrete* standpoint, these networks are often modelled as evolving graphs, i.e. a sequence of graphs $\mathcal{G} = (G_1, G_2, ..., G_\delta)$ such that $G_i = (V, E_i)$ represents the network topology at time step i. Such a sequence is said to be T-interval connected if for any $t \in [1, \delta - T + 1]$ all graphs in $\{G_t, G_{t+1}, ..., G_{t+T-1}\}$ share a common connected spanning subgraph. In this paper, we consider the problem of deciding whether a given sequence $\mathcal{G}$ is T-interval connected for a given T. We also consider the related problem of finding the largest T for which a given $\mathcal{G}$ is T-interval connected. We assume that the changes between two consecutive graphs are arbitrary, and that two operations, *binary intersection* and *connectivity testing*, are available to solve the problems. We show that $\Omega(\delta)$ such operations are required to solve both problems, and we present optimal $O(\delta)$ online algorithms for both problems. We extend our online algorithms to a dynamic setting in which connectivity is based on the recent evolution of the network.

Keywords: T-interval connectivity, Dynamic graphs, Time-varying graphs

1. Introduction

Dynamic networks consist of entities making contact over time with one another. The types of dynamics resulting from these interactions are varied in scale and nature. For instance, some of these networks remain connected at all times [9]; others are always disconnected [6] but still offer some kind of connectivity over time and space (*temporal* connectivity); others are recurrently connected, periodic, etc. All of these contexts can be represented as dynamic graph classes. A dozen such classes were identified in [4] and organized into a hierarchy.

☆A short version of this paper appeared in the proceedings of CIAC 2015.
Part of this work was done while Joseph Peters was visiting the LaBRI as a guest professor of the University of Bordeaux. This work was partially funded by the ANR projects DISPLEXITY (ANR-11-BS02-014) and DAISIE (ANR-13-ASMA-0004). This study has been carried out in the frame of "The Investments for the Future" Programme IdEx Bordeaux CPU (ANR-10-IDEX-03-02).

Email addresses: arnaud.casteigts@labri.fr (Arnaud Casteigts), ralf.klasing@labri.fr (Ralf Klasing), mneggaz@labri.fr (Yessin M. Neggaz), peters@cs.sfu.ca (Joseph Peters)

Given a dynamic graph, a natural question to ask is to which of the classes this graph belongs. This question is interesting because most of the known classes of dynamic graphs correspond to necessary or sufficient conditions for given distributed problems or algorithms (broadcast, election, spanning trees, token forwarding, etc.). Thus, being able to classify a graph in the hierarchy is useful for determining which problems can be successfully solved and which algorithms can be executed on that graph. Furthermore, classification tools, such as testing algorithms for given classes, can be useful for choosing a good algorithm in settings where the evolution of a network is not known in advance. An algorithm designer can record topological traces from the real world and then test whether the corresponding dynamic graphs are included in classes that correspond to the topological conditions for the problem at hand [3]. Alternatively, online algorithms that process dynamic graphs as they evolve could accomplish the same goal without the need to collect traces.

Dynamic graphs can be modelled in a number of ways. It is often convenient, when looking at the topology from a global standpoint (e.g., a recorded trace), to represent a dynamic graph as a sequence of graphs $\mathcal{G} = (G_1, G_2, ..., G_\delta)$, each of which corresponds to the state of the dynamic graph at a given discrete time instant. (also known as *untimed* evolving graphs [2]). Solutions for testing the inclusion of such a dynamic graph in a handful of basic classes were provided in [3]; these classes are those in which a *journey* (temporal path) or *strict journey* (a journey that traverses at most one edge per G_i) exists between any pair of nodes. In this particular case, the problem reduces to testing whether the transitive closure of (strict) journeys is a complete graph. The transitive closure itself can be computed efficiently in a number of ways [1, 2, 10]. To the best of our knowledge, no further work has been done on testing properties of dynamic graphs.

Recently, the class of T*-interval connected* graphs was identified in [8] as playing an important role in several distributed problems, such as determining the size of a network or computing a function of the initial inputs of the nodes. Informally, T-interval connectivity requires that there exists a common connected spanning subgraph for every T consecutive graphs in the sequence $\mathcal{G}$. This class generalizes the class of dynamic graphs that are connected at all time instants [9]. Indeed, the latter corresponds to the case that $T = 1$. From a set-theoretic viewpoint, however, every $T > 1$ induces a class of graphs that is a strict subset of the class in [9] because a graph that is T-interval connected is obviously 1-interval connected. Hence, T-interval connectivity is more specialized in this sense.

In this paper, we look at the problem of deciding whether a given sequence $\mathcal{G}$ is T-interval connected for a given T. We also consider the related problem of finding the largest T for which the given $\mathcal{G}$ is T-interval connected. We assume that the changes between two consecutive graphs are arbitrary and we do not make any assumptions about the data structures that are used to represent the sequence of graphs. As such, we focus on high-level strategies that work directly at the graph level. Precisely, we consider two graph-level operations as building blocks: *binary intersection* (given two graphs, compute their intersection) and *connectivity testing* (given a graph, decide whether it is connected). Put together, these operations have a strong and natural connection with the problems that we are studying. We first show that both problems require $\Omega(\delta)$ such operations using the basic argument that every graph of the sequence must be considered at least once. More surprisingly, we show that both problems can be solved using only $O(\delta)$ such operations and we develop optimal online algorithms that achieve these matching bounds. Hence, the cost of the operations – both of them linear in the number of edges – is counterbalanced by efficient high-level logic that could, for instance, benefit from dedicated circuits (or optimized code) for both operations.

The paper is organized as follows. Section 2 presents the main definitions and makes some basic observations, including the fact that both problems can be solved using $O(\delta^2)$ operations (intersections or connectivity tests) by a naive strategy that examines $O(\delta^2)$ intermediate graphs. Section 3 presents a second strategy, yielding upper bounds of $O(\delta \log \delta)$ operations for both problems. Its main interest is in the fact that it can be parallelized, and this allows us to classify both problems as being in **NC** (i.e. Nick's class). In Section 4 we present an optimal strategy which we use to solve both problems online in $O(\delta)$ operations. This strategy exploits structural properties of the problems to construct carefully selected subsequences of the intermediate graphs. In particular, only $O(\delta)$ of the $O(\delta^2)$ intermediate graphs are selected for evaluation by the algorithms. In Section 5, we extend our online algorithms to a dynamic setting in which the measure of connectivity is based on the recent evolution of the network.

2. Definitions and Basic Observations

Graph Model. In this work, we consider dynamic graphs that are given as untimed evolving graphs, that is, a sequence $\mathcal{G} = (G_1, G_2, ..., G_\delta)$ of graphs such that $G_i = (V, E_i)$ describes the network topology at (discrete) time i. The parameter δ is called the *length* of the sequence $\mathcal{G}$. It corresponds to the number of time steps that this graph covers. Observe that V is non-varying; only the set of edges varies. Unless otherwise stated, we consider *undirected* edges throughout the paper, which is the setting in which T-interval connectivity was originally introduced. However, the fact that our algorithms are high-level allows them to work exactly the same for T-interval strong connectivity (which is the analogue of T-interval connectivity for directed graphs [8]), provided that both basic operations (i.e. intersection and connectivity test) are given. As we shall discuss, these operations have linear cost in the number of edges in both directed and undirected graphs.

Definition 1 (Intersection graph). Given a (finite) set S of graphs $\{G' = (V, E'), G'' = (V, E''), \ldots\}$, we call the graph $(V, \cap\{E', E'', \ldots\})$ the *intersection graph* of S and denote it by $\cap\{G', G'', \ldots\}$. When the set consists of only two graphs, we talk about *binary intersection* and use the infix notation $G' \cap G''$. If the intersection involves a consecutive subsequence $(G_i, G_{i+1}, \ldots, G_j)$ of a dynamic graph $\mathcal{G}$, then we denote the intersection graph $\cap\{G_i, G_{i+1}, \ldots, G_j\}$ simply as $G_{(i,j)}$.

Definition 2 (*T*-interval connectivity). A dynamic graph $\mathcal{G}$ is said to be T*-interval connected* if the intersection graph $G_{(t,t+T-1)}$ is connected for every $t \in [1, \delta - T + 1]$. In other words, all graphs in $\{G_t, G_{t+1}, ..., G_{t+T-1}\}$ share a common connected spanning subgraph.

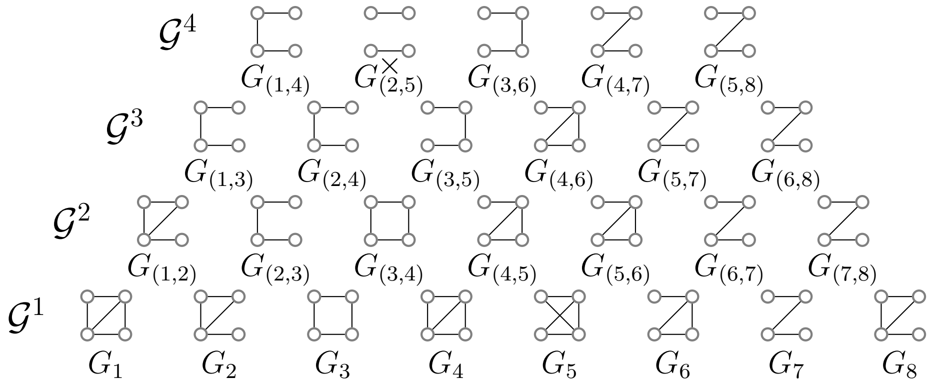


Figure 1: Example of an intersection hierarchy for a given dynamic graph $\mathcal{G}$ of length $\delta = 8$. Here, $\mathcal{G}$ is 3-interval connected, but not 4-interval connected; $\mathcal{G}^4$ contains a disconnected graph $G_{(2,5)}$ because G_2, G_3, G_4, G_5 share no connected spanning subgraph.

Definition 3 (Testing $\boldsymbol{T}$-interval connectivity). We will use the term T-Interval-Connectivity to refer to the problem of deciding whether a dynamic graph $\mathcal{G}$ is T-interval connected for a given T.

Definition 4 (Interval connectivity). We will use Interval-Connectivity to refer to the problem of finding $\max\{T : \mathcal{G}$ is T-interval connected$\}$ for a given $\mathcal{G}$.

Let $\mathcal{G}^T = (G_{(1,T)}, G_{(2,T+1)}, ..., G_{(\delta-T+1,\delta)})$. We call $\mathcal{G}^T$ the T^{th} *row* in $\mathcal{G}$'s intersection hierarchy, as depicted in Fig. 1. A particular case is $\mathcal{G}^1 = \mathcal{G}$. For any $1 \leq i \leq \delta - T + 1$, we define $\mathcal{G}^T[i] = G_{(i,i+T-1)}$. We call $\mathcal{G}^T[i]$ the i^{th} *element of row* $\mathcal{G}^T$ and i is called the *index of* $\mathcal{G}^T[i]$ *in row* $\mathcal{G}^T$.

Observation 1. *A dynamic graph $\mathcal{G}$ is T-interval connected if and only if all graphs in $\mathcal{G}^T$ are connected.*

Computational Model. As shown in Observation 1, the concept of T-interval connectivity can be reformulated quite naturally in terms of the connectivity of some intersection graphs. For this reason, we consider two building block operations: *binary intersection* (given two graphs, compute their intersection) and *connectivity testing* (given a graph, decide whether it is connected). This approach is suitable for a high-level study of these problems when the details of changes between successive graphs in a sequence are arbitrary. If more structural information about the evolution of the dynamic graphs is known, for example, if it is known that the number of changes between each pair of consecutive graphs is bounded by a constant, then algorithms could benefit from the use of sophisticated data structures and a lower-level approach might be more appropriate.

Observation 2 (Cost of the operations). *Using an adjacency list data structure for the graphs, a binary intersection can be performed in linear time in the number of edges. Checking connectivity of a graph can also be done in linear time in the number of edges. In the* *case of undirected graphs, it can be done by building a depth-first search tree from an arbitrary root node and testing whether all nodes are reachable from the root node. Tarjan's algorithm for strongly connected components can be used for directed graphs. Hence, both the* intersection *operation and the* connectivity testing *operation have similar costs. In what follows, we will refer to them as* elementary *operations. One advantage of using these elementary operations is that the high-level logic of the algorithms becomes elegant and simple. Also, their cost can be counterbalanced by the fact that they are highly generic and thus could benefit from dedicated circuits (e.g., FPGA) or optimized code.*

Naive Upper Bound. One can easily see that both problems are solvable using $O(\delta^2)$ elementary operations based on a naive strategy. It suffices to compute the rows of $\mathcal{G}'s$ intersection hierarchy incrementally using the fact that each graph $G_{(i,j)}$ can be obtained as $G_{(i,j-1)} \cap G_{(i+1,j)}$. For instance, $G_{(3,6)} = G_{(3,5)} \cap G_{(4,6)}$ in Fig. 1. Hence, each row k can be computed from row $k-1$ using $O(\delta)$ binary intersections. In the case of T-Interval-Connectivity, one simply has to repeat the operation until the T^{th} row, then answer `true` iff all graphs in this row are connected. The total cost is $O(\delta T) = O(\delta^2)$ binary intersections, plus $\delta - T + 1 = O(\delta)$ connectivity tests for the T^{th} row. Solving Interval-Connectivity is similar except that one needs to test the connectivity of all new graphs during the process. If a disconnected graph is first found in some row k, then the answer is $k-1$. If all graphs are connected up to row δ, then δ is the answer. Since there are $O(\delta^2)$ graphs in the intersection hierarchy, the total number of connectivity tests and binary intersections is $O(\delta^2)$.

Lower Bound. The following lower bound is valid for any algorithm that uses only the two elementary operations *binary intersection* and *connectivity test*.

Lemma 1. $\Omega(\delta)$ *elementary operations are necessary to solve* T-Interval-Connectivity.

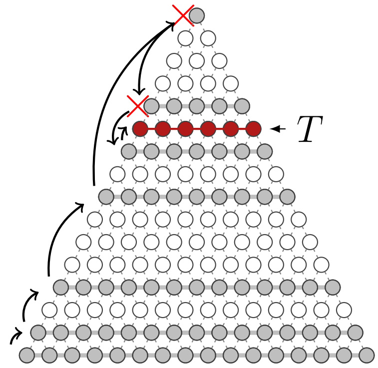
Proof (by contradiction). Let $\mathcal{A}$ be an algorithm that uses only elementary operations and that decides whether any sequence of graphs is T-interval connected in $o(\delta)$ operations. Then, for any sequence $\mathcal{G}$, at least one graph in $\mathcal{G}$ is never accessed by $\mathcal{A}$. Let $\mathcal{G}_1$ be a sequence that is T-interval connected and suppose that $\mathcal{A}$ decides that $\mathcal{G}_1$ is T-interval connected without accessing graph G_k. Now, consider a sequence $\mathcal{G}_2$ that is identical to $\mathcal{G}_1$ except G_k is replaced by a disconnected graph G'_k. Since G'_k is never accessed, the executions of $\mathcal{A}$ on $\mathcal{G}_1$ and $\mathcal{G}_2$ are identical and $\mathcal{A}$ incorrectly decides that $\mathcal{G}_2$ is T-interval connected. □

A similar argument can be used for Interval-Connectivity by making the answer T dependent on the graph G_k that is never accessed.

3. Row-Based Strategy

In this section, we present a basic strategy that improves upon the previous naive strategy, yielding upper bounds of $O(\delta \log \delta)$ operations for both problems. Its main interest is in the fact that it can be parallelized, and this allows us to show that both problems are in

Figure 2: Example of interval connectivity testing based on the computation of power rows. Here $\delta = 16$ and $T = 11$. The computation of power rows stops upon reaching $\mathcal{G}^{16}$ which contains a disconnected graph ($\times$). A binary search between rows $\mathcal{G}^{8}$ and $\mathcal{G}^{16}$ is then used to find $\mathcal{G}^{11}$, the highest row where all graphs are connected.



NC, i.e. parallelizable on a PRAM with a polylogarithmic running time. We first describe the algorithms for a sequential machine (RAM). The general strategy is to compute only some of the rows of $\mathcal{G}$'s intersection hierarchy based on the following lemma.

Lemma 2. *If some row $\mathcal{G}^k$ is already computed, then any row $\mathcal{G}^\ell$ for $k+1 \leq \ell \leq 2k$ can be computed with $O(\delta)$ elementary operations.*

Proof. Assume that row $\mathcal{G}^k$ is already computed and that one wants to compute row $\mathcal{G}^\ell$ for some $k+1 \leq \ell \leq 2k$. Note that row $\mathcal{G}^\ell$ consists of the entries $\mathcal{G}^\ell[1], \ldots, \mathcal{G}^\ell[\delta - \ell + 1]$. Now, observe that for any $k+1 \leq \ell \leq 2k$ and for any $1 \leq i \leq \delta - \ell + 1$, $\mathcal{G}^\ell[i] = G_{(i,i+\ell-1)} = G_{(i,i+k-1)} \cap G_{(i+\ell-k,i+\ell-1)} = \mathcal{G}^k[i] \cap \mathcal{G}^k[i+\ell-k]$. Hence, $\delta - \ell + 1 = O(\delta)$ intersections are sufficient to compute all of the entries of row $\mathcal{G}^\ell$. $\square$

***T*-Interval-Connectivity.** Using Lemma 2, we can incrementally compute "power rows" $\mathcal{G}^{2^i}$ for all i from 1 to $\lceil \log_2 T \rceil - 1$ without computing the intermediate rows. Then, we compute row $\mathcal{G}^T$ directly from row $\mathcal{G}^{2^{\lceil \log_2 T \rceil - 1}}$ (again using Lemma 2). This way, we compute $\lceil \log_2 T \rceil = O(\log \delta)$ rows using $O(\delta \log \delta)$ intersections, after which we perform $O(\delta)$ connectivity tests.

Interval-Connectivity. Here, we incrementally compute rows $\mathcal{G}^{2^i}$ until we find a row that contains a disconnected graph (thus, a connectivity test is performed after each intersection). By Lemma 2, each of these rows can be computed using $O(\delta)$ intersections. Suppose that row $\mathcal{G}^{2^{j+1}}$ is the first power row that contains a disconnected graph, and that $\mathcal{G}^{2^j}$ is the row computed before $\mathcal{G}^{2^{j+1}}$. Next, we do a binary search of the rows between $\mathcal{G}^{2^j}$ and $\mathcal{G}^{2^{j+1}}$ to find the row $\mathcal{G}^T$ with the highest row number T such that all graphs on this row are connected (see Fig. 2 for an illustration of the algorithm). The computation of each of these rows is based on row $\mathcal{G}^{2^j}$ and takes $O(\delta)$ intersections by Lemma 2. Overall, we compute at most $2\lceil \log_2 T \rceil = O(\log \delta)$ rows using $O(\delta \log \delta)$ intersections and the same number of connectivity tests.

Now we establish that these problems are in **NC** by showing that our algorithms are efficiently parallelizable.

Lemma 3. *If some row $\mathcal{G}^k$ is already computed, then any row between $\mathcal{G}^{k+1}$ and $\mathcal{G}^{2k}$ can be computed in $O(1)$ time on an EREW PRAM with $O(\delta)$ processors.*

PROOF. Assume that row $\mathcal{G}^k$ is already computed, and that one wants to compute row $\mathcal{G}^\ell$, consisting of the entries $\mathcal{G}^\ell[1], \dots, \mathcal{G}^\ell[\delta - \ell + 1]$, for some $k + 1 \leq \ell \leq 2k$. Since $\mathcal{G}^\ell[i] = \mathcal{G}^k[i] \cap \mathcal{G}^k[i + \ell - k]$, $1 \leq i \leq \delta - \ell + 1$, the computation of row $\mathcal{G}^\ell$ can be implemented on an EREW PRAM with $\delta - \ell + 1$ processors in two rounds as follows. Let P_i, $1 \leq i \leq \delta - \ell + 1$, be the processor dedicated to computing $\mathcal{G}^\ell[i]$. In the first round P_i reads $\mathcal{G}^k[i]$, and in the second round P_i reads $\mathcal{G}^k[i + \ell - k]$. This guarantees that each P_i has exclusive access to the entries of row $\mathcal{G}^k$ that it needs for its computation. Hence, row $\mathcal{G}^\ell$ can be computed in $O(1)$ time on an EREW PRAM using $O(\delta)$ processors. □

***T*-Interval-Connectivity on an EREW PRAM.** The sequential algorithm for this problem computes $O(\log \delta)$ rows. By Lemma 3, each of these rows can be computed in $O(1)$ time on an EREW PRAM with $O(\delta)$ processors. Therefore, all of the rows (and hence all necessary intersections) can be computed in $O(\log \delta)$ time with $O(\delta)$ processors. The $O(\delta)$ connectivity tests for row $\mathcal{G}^T$ can be done in $O(1)$ time with $O(\delta)$ processors. Then, the processors can establish whether or not all graphs in row $\mathcal{G}^T$ are connected by computing the logical AND of the results of the $O(\delta)$ connectivity tests in time $O(\log \delta)$ on a EREW PRAM with $O(\delta)$ processors using standard techniques (see [5, 7]). The total time is $O(\log \delta)$ on an EREW PRAM with $O(\delta)$ processors.

Interval-Connectivity on an EREW PRAM. The sequential algorithm for this problem computes $O(\log \delta)$ rows. Differently from T-INTERVAL-CONNECTIVITY, a connectivity test is done for each of the computed graphs (rather than just those of the last row) and it has to be determined for each computed row whether or not all of the graphs are connected. This takes $O(\log \delta)$ time for each of the $O(\log \delta)$ computed rows using the same techniques as for T-INTERVAL-CONNECTIVITY. The total time is $O(\log^2 \delta)$ on an EREW PRAM with $O(\delta)$ processors.

4. Optimal Solution

We now present our strategy for solving both T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY using a linear number of elementary operations (in the length δ of $\mathcal{G}$), matching the $\Omega(\delta)$ lower bound presented in Section 2. The strategy relies on the concept of *ladder*. Informally, a ladder is a sequence of graphs that "climbs" the intersection hierarchy bottom-up.

Definition 5. The *right ladder of length l at index i*, denoted by $\mathcal{R}^l[i]$, is the sequence of intersection graphs $(\mathcal{G}^k[i],\ k = 1, 2, \dots, l)$. The *left ladder of length l at index i*, denoted by $\mathcal{L}^l[i]$, is the sequence $(\mathcal{G}^k[i - k + 1],\ k = 1, 2, \dots, l)$. A right (*resp.* left) ladder of length $l - 1$ at index i is said to be *incremented* when graph $\mathcal{G}^l[i]$ (*resp.* $\mathcal{G}^l[i - l + 1]$) is added to it, and the resulting sequence of intersection graphs is called the *increment* of that ladder.

Lemma 4. *A ladder of length l can be computed using $l-1$ binary intersections.*

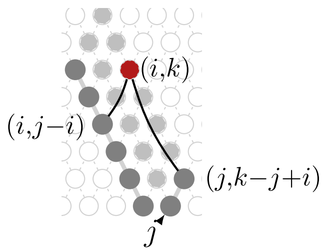
Proof. Consider a right ladder $\mathcal{R}^l[i]$. For any $k \in [2,l]$ it holds that $\mathcal{G}^k[i] = \mathcal{G}^{k-1}[i] \cap G_{i+k-1}$. Indeed, by definition, $\mathcal{G}^{k-1}[i] = \cap\{G_i, G_{i+1}, ..., G_{i+k-2}\}$. The ladder can thus be built bottom-up using a single new intersection at each level.

Consider a left ladder $\mathcal{L}^l[i]$. For any $k \in [2,l]$ it holds that $\mathcal{G}^k[i-k+1] = G_{i-k+1} \cap \mathcal{G}^{k-1}[i-k+2]$. Indeed, by definition, $\mathcal{G}^{k-1}[i-k+2] = \cap\{G_{i-k+2}, G_{i-k+3}, ..., G_i\}$. The ladder can thus be built bottom-up using a single new intersection at each level. □

Lemma 5. *Given $\mathcal{L}^{l_\ell}[j-1]$ and $\mathcal{R}^{l_r}[j]$, any pair (i,k) such that $j - l_\ell \le i < j$ and $j - i < k \le j - i + l_r$, $\mathcal{G}^k[i]$ can be computed by a single binary intersection, namely $\mathcal{G}^k[i] = \mathcal{G}^{j-i}[i] \cap \mathcal{G}^{k-j+i}[j]$.*

Proof. By definition, $\mathcal{G}^k[i] = \cap\{G_i, G_{i+1}, ..., G_{i+k-1}\}$ and $\mathcal{G}^{j-i}[i] = \cap\{G_i, G_{i+1}, ..., G_{j-1}\}$ and $\mathcal{G}^{k-j+i}[j] = \cap\{G_j, G_{j+1}, ..., G_{i+k-1}\}$. It follows that $\mathcal{G}^k[i] = \mathcal{G}^{j-i}[i] \cap \mathcal{G}^{k-j+i}[j]$. By definition, $\mathcal{G}^{j-i}[i] \in \mathcal{L}^{l_\ell}[j-1]$ and $\mathcal{G}^{k-j+i}[j] \in \mathcal{R}^{l_r}[j]$, so only a single binary intersection is needed. □

Informally, the constraints $j - l_\ell \le i < j$ and $j - i < k \le j - i + l_r$ in Lemma 5 define a rectangle delimited by two ladders and two lines that are parallel to the two ladders as shown in the figure to the right. The pairs (i,k) defined by the constraints, shown in light grey in the figure, include all pairs that are strictly inside the rectangle, and all pairs on the parallel lines, but pairs on the two ladders are excluded.



***T*-Interval-Connectivity.** We describe our optimal algorithm for this problem with reference to Fig. 3 below which shows two examples of the execution of the algorithm (see Algorithm 1 for details). The algorithm traverses the T^{th} row in the intersection hierarchy from left to right, starting at $\mathcal{G}^T[1]$. If a disconnected graph is found, the algorithm returns `false` and terminates. If the algorithm reaches the last graph in the row, i.e. $\mathcal{G}^T[\delta - T + 1]$, and no disconnected graph was found, then it returns `true`. The graphs $\mathcal{G}^T[1], \mathcal{G}^T[2], \ldots, \mathcal{G}^T[\delta - T + 1]$ are computed based on the set of ladders $\mathcal{S} = \{\mathcal{L}^T[T], \mathcal{R}^{T-1}[T+1], \mathcal{L}^T[2T], \mathcal{R}^{T-1}[2T+1], \ldots\}$, which are constructed as follows. Each left ladder is built entirely (from bottom to top) when the traversal arrives at its top location in

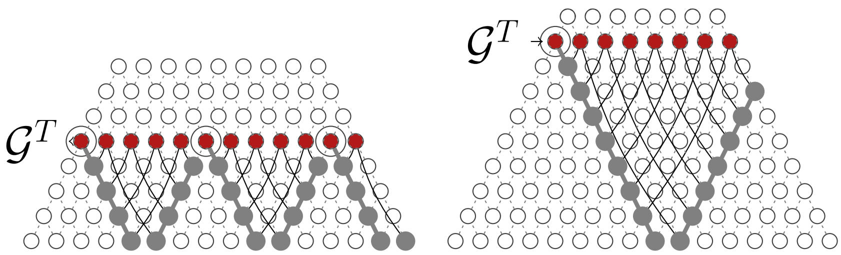


Figure 3: Examples of the execution of the optimal algorithm for T-Interval-Connectivity with $T < \delta/2$ (left) and $T \ge \delta/2$ (right). $\mathcal{G}$ is T-interval connected in both examples.

row T (i.e. where the last increment is to take place). For instance, $\mathcal{L}^T[T]$ is built when the walk is at index 1 in row T, $\mathcal{L}^T[2T]$ is built at index $T+1$, and so on. If a disconnected graph is found in the process, the execution terminates returning `false`.

```
k ← T                                                    // current row (non-changing)
i ← 1                                                    // current index in the row
next ← 1                                                 // trigger for next ladder construction
// walk until stepping out of the intersection hierarchy
while i ≤ δ − k + 1 do
    if i = next then
        next ← i + k
        if ¬computeFromRight(k, i, next) then
            return false
    else
        computeFromIntersection(k, i, next)
    if ¬isConnected(𝒢^k[i]) then
        return false
    i ← i + 1
return true
---------------------------------------------------------------------------------------------
function computeFromRight(k, i, next):                   // compute the left ladder ℒ^k[i]
    k' ← 1                                               // row of first increment
    i' ← next − 1                                        // index of first increment
    while k' < k do
        if ¬isConnected(𝒢^{k'}[i']) then
            return false                                 // a disconnected graph was found
        k' ← k' + 1
        i' ← i' − 1
        𝒢^{k'}[i'] ← 𝒢^{k'−1}[i' + 1] ∩ G_{i'}            // “increment” the ladder
---------------------------------------------------------------------------------------------
function computeFromIntersection(k, i, next):            // “increment” the right ladder
    k' ← k − next + i                                    // row of increment (right ladder)
    𝒢^{k'}[next] ← 𝒢^{k'−1}[next] ∩ G_{next+k'−1}         // “increment” right ladder
    𝒢^k[i] ← 𝒢^{next−i}[i] ∩ 𝒢^{k'}[next]                 // compute intersection based on Lemma 5
```

Algorithm 1: Optimal algorithm for T-Interval-Connectivity

Differently from left ladders, right ladders are constructed gradually as the traversal proceeds. Each time that the traversal moves right to a new index in the T^{th} row, the current right ladder is incremented and the new top element of this right ladder is used immediately to compute the graph at the current index in the T^{th} row (using Lemma 5). This continues until the right ladder reaches row $T-1$ after which a new left ladder is built.

The set $\mathcal{S}$ of ladders constructed by this process includes at most $\lfloor \delta/T \rfloor$ left ladders and $\lfloor \delta/T \rfloor$ right ladders, each of length at most T. By Lemma 4, the set of ladders $\mathcal{S}$ can be computed using less than 2δ binary intersections. Based on Lemma 5, each of the $\delta - T + 1$ graphs $\mathcal{G}^T[i]$ in row T can be computed at the cost of a single intersection of two graphs in $\mathcal{S}$. At most $\delta - T + 1$ connectivity tests are performed for row T. This establishes the following result which matches the lower bound of Lemma 1.

Theorem 1. T-Interval-Connectivity *can be solved with* $\Theta(\delta)$ *elementary operations, which is optimal (to within a constant factor).*

Interval-Connectivity. The strategy of our optimal algorithm for this problem is in the same spirit as the one for T-Interval-Connectivity. However, it is more complex and corresponds to a walk in the two dimensions of the intersection hierarchy. It is best understood with reference to Fig. 4 which shows an example of the execution of the algorithm (see Algorithm 2 for details).

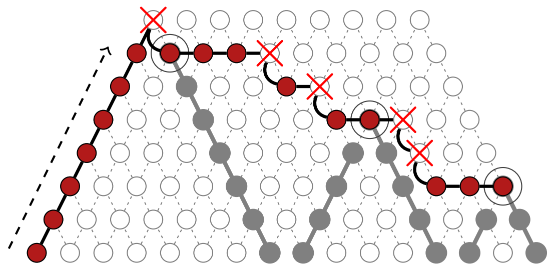


Figure 4: Example of the execution of the optimal algorithm for Interval-Connectivity. *(It is a coincidence that the rightmost ladder matches the outer face.)*

The walk starts at the bottom left graph $\mathcal{G}^1[1]$ and builds a right ladder incrementally until it encounters a disconnected graph. If $\mathcal{G}^\delta[1]$ is reached and is connected, then $\mathcal{G}$ is δ-interval connected and execution terminates returning δ. Otherwise, suppose that a disconnected graph is first found in row $k+1$. Then k is an upper bound on the connectivity of $\mathcal{G}$ and the walk drops down a level to $\mathcal{G}^k[2]$ which is the next graph in row k that needs to be checked. This requires the construction of a left ladder $\mathcal{L}^k[k+1]$ of length k ending at $\mathcal{G}^k[2]$. The walk proceeds rightward on row k using a similar traversal strategy to the algorithm for T-Interval-Connectivity. Here, however, every time that a disconnected graph is found, the walk drops down by one row. The dropping down operation, say, from some $\mathcal{G}^k[i]$, is made in two steps (curved line in Fig. 4). First it goes to $\mathcal{G}^{k-1}[i]$, which is necessarily connected because $\mathcal{G}^k[i-1]$ is connected (so a connectivity test is not needed here), and then it moves one unit right to $\mathcal{G}^{k-1}[i+1]$. If the walk eventually reaches the rightmost graph of some row and this graph is connected, then the algorithm terminates returning the corresponding row number as T. Otherwise the walk will terminate at a disconnected graph in row 1 and $\mathcal{G}$ is not T-interval connected for any T. In this case, the algorithm returns $T = 0$.

Similarly to the algorithm for T-Interval-Connectivity, the computations of the graphs in a walk by Algorithm 2 (for Interval-Connectivity) use binary intersections based on Lemmas 4 and 5. If the algorithm returns that $\mathcal{G}$ is T-interval connected, then each graph $\mathcal{G}^T[1], \mathcal{G}^T[2], \ldots, \mathcal{G}^T[\delta - T + 1]$ must be connected. The graphs that are on the walk are checked directly by the algorithm. For each graph $\mathcal{G}^T[i]$ on row T that is below the walk, there is a graph $\mathcal{G}^j[i]$ with $j > T$ that is on the walk and is connected and this implies that $\mathcal{G}^T[i]$ is connected.

```
k ← 1                                                   // current row
i ← 1                                                   // current index in the row
next ← 2                                                // trigger for next ladder construction

// builds a right ladder until a disconnected graph is found
while isConnected(\mathcal{G}^k[1]) do
    k ← k + 1
    if k > δ then
        return δ                                        // the graph is δ-interval connected
    else
        \mathcal{G}^k[1] ← \mathcal{G}^{k-1}[1] ∩ G_k    // "increment" the right ladder
if k = 1 then
    return 0                                            // the graph is 0-interval connected

k ← k − 1                                               // move down
i ← i + 1                                               // move right

// walk until stepping out of the hierarchy
while i ≤ δ − k + 1 do
    if i = next then
        next ← i + k
        computeFromRight(k, i, next)
    else
        computeFromIntersection(k, i, next)
        if ¬isConnected(\mathcal{G}^k[i]) then
            k ← k − 1
    if k = 0 then
        return 0
    i ← i + 1
return k
---------------------------------------------------------------------------------------------
function computeFromRight(k, i, next):                  // compute the left ladder \mathcal{L}^k[i]
    k' ← 1                                              // row of first increment
    i' ← next − 1                                       // index of first increment
    while k' < k do
        if ¬isConnected(\mathcal{G}^{k'}[i']) then
            k ← k' − 1                                  // move the original walk..
            i ← i' + 1                                  // ..below-right disconnected graph,
            return                                      // abort function
        k' ← k' + 1
        i' ← i' − 1
        \mathcal{G}^{k'}[i'] ← \mathcal{G}^{k'-1}[i' + 1] ∩ G_{i'}   // "increment" the ladder
---------------------------------------------------------------------------------------------
function computeFromIntersection(k, i, next):           (Same function as for Algorithm 1)
    k' ← k − next + i                                   // row of increment (right ladder)
    \mathcal{G}^{k'}[next] ← \mathcal{G}^{k'-1}[next] ∩ G_{next+k'-1}   // "increment" right ladder
    \mathcal{G}^k[i] ← \mathcal{G}^{next-i}[i] ∩ \mathcal{G}^{k'}[next] // compute intersection based on Lemma 5
```

Algorithm 2: Optimal algorithm for INTERVAL-CONNECTIVITY

The ranges of the indices covered by the left ladders that are constructed by this process are disjoint, so their total length is $\mathrm{O}(\delta)$. The first right ladder has length at most δ and each subsequent right ladder has length less than the left ladder that precedes it so the total length of the right ladders is also $\mathrm{O}(\delta)$. Therefore, this algorithm performs at most $\mathrm{O}(\delta)$ binary intersections and $\mathrm{O}(\delta)$ connectivity tests. This establishes the following result which matches the lower bound of Lemma 1.

Theorem 2. INTERVAL-CONNECTIVITY *can be solved with* $\Theta(\delta)$ *elementary operations, which is optimal (up to a constant factor).*

Online Algorithms. The optimal algorithms for T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY can be adapted to an online setting in which the sequence of graphs $G_1, G_2, G_3, \ldots$ of a dynamic graph $\mathcal{G}$ is processed in the order that the graphs are received. In the case of T-INTERVAL-CONNECTIVITY, the algorithm cannot provide an answer until at least T graphs have been received. When the T^{th} graph is received, the algorithm builds the first left ladder using $T-1$ binary intersections. It can then perform a connectivity test and answer whether or not the sequence is T-interval connected so far. After this initial period, a T-connectivity test can be performed for the T most recently received graphs (by performing a connectivity test on the corresponding graph in row T) after the receipt of each new graph. At no time does the number of intersections performed to build left ladders exceed the number of graphs received and the same is true for right ladders. Furthermore, each new graph after the first $T-1$ corresponds to a graph in row T which can be computed with one intersection by Lemma 5. In summary, the amortized cost is $O(1)$ elementary operations for each graph received and for each T-connectivity test after the initial period. The analysis for INTERVAL-CONNECTIVITY is similar except the algorithm can report the connectedness of the sequence so far starting with the first graph received.

Theorem 3. T-INTERVAL-CONNECTIVITY *and* INTERVAL-CONNECTIVITY *can be solved online with an amortized cost of* $\Theta(1)$ *elementary operations per graph received.*

5. Dynamic Online Interval Connectivity

The algorithms in this section are motivated by Internet protocols like TCP (Transmission Control Protocol) which adjust their behaviour dynamically in response to recent network events and conditions such as dropped packets and congestion. T-interval connectivity is a measure of the stability of a network. Generally, larger values of T indicate that communication is more reliable, so it is natural to consider a dynamic version of interval connectivity that is based only on the recent states of a network rather than the entire history of a network. We formalize this notion of recent history by introducing the concept of T*-stable* graphs. We then define the dynamic online versions of both T-INTERVAL-CONNECTIVITY and INTERVAL-CONNECTIVITY in terms of T-stable graphs.

Definition 6 (T-stable graph). A graph G_i, $i \geq T$, of a sequence $\mathcal{G} = (G_1, G_2, ..., G_\delta)$ is T*-stable* for a given T iff the subsequence $G_{i-(T-1)}, G_{i-(T-2)}, \ldots, G_{i-1}, G_i$ is T-interval connected.

Definition 7 (Testing T-Stability). The T-Stability problem for a given T is the problem of deciding for each received graph G_i, $i \geq T$, whether G_i is T-stable.

Definition 8 (Testing Stability). We use the term Stability to refer to the problem of finding $T_i = \max\{T : G_i \text{ is } T\text{-stable}\}$ for each received graph G_i.

As before, the first problem is a decision problem with true/false output, while the second is a maximization problem with integer output. Here, however, one such output is required after each graph in the sequence is received.

T-Stability. Our algorithm for T-Stability is similar to Algorithm 1 for T-Interval-Connectivity. The differences are that the algorithm for T-Stability produces an output after each graph of a sequence is received, and the algorithm does not terminate if a disconnected graph is found on row T of the hierarchy. Instead, it continues until the last graph in the sequence is received. The ladders constructed by the algorithm for T-Stability are the same as the ladders that would be constructed by Algorithm 1 for a dynamic graph that is T-interval connected (see Figure 3 for examples). Given a dynamic graph $\mathcal{G} = (G_1, G_2, ..., G_\delta)$, T-Stability is undefined for the graphs G_i with $i < T$, so the algorithm returns $\perp$ after each of the first $T-1$ graphs is received. When G_T is received, the algorithm builds a left ladder and returns `true` (resp. `false`) if the top graph of the ladder (i.e. $\mathcal{G}^T[1]$) is connected (resp. disconnected). Then the walk progresses rightward along row T every time that a graph is received, alternately building left and right ladders in such a way that the graph $\mathcal{G}^T[i-(T-1)]$ can always be computed from G_i with a single intersection (using Lemma 5). G_i is T-stable iff $\mathcal{G}^T[i-(T-1)]$ is connected and `true` or `false` is output as appropriate. By the same analysis as the analysis for the online version of Algorithm 1, the number of intersections performed to build left ladders never exceeds the number of graphs received, and the same is true for the number of intersections to build right ladders and the number of connectivity tests.

Theorem 4. T-Stability *can be solved online with an amortized cost of* $\Theta(1)$ *elementary operations per graph received.*

Stability. The algorithm for this problem must find $T_i = \max\{T : G_i \text{ is } T\text{-stable}\}$ for each received graph G_i. Our algorithm for Stability generalizes the strategy that we used in the algorithm for Interval-Connectivity by trying to climb as high as possible in the hierarchy, even after a disconnected intersection graph is found. This is necessary because the sequence of values $T_1, T_2, T_3, \ldots$ for Stability is not necessarily monotonic.

The algorithm for Stability uses right and left ladders to walk through the intersection hierarchy. The general idea is that the walk goes up when the current intersection graph is connected and down when it is disconnected (unless the walk is on the bottom level of

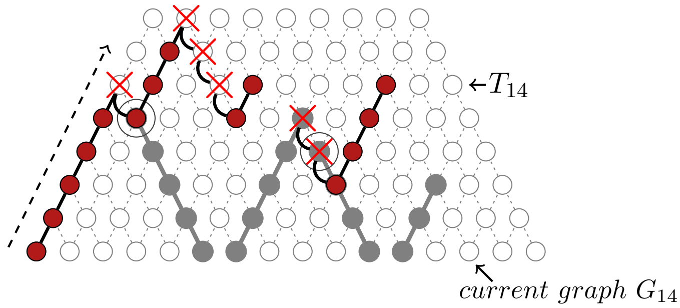


Figure 5: Example of the execution of the Stability algorithm.

the hierarchy in which case it goes right to the next graph). This is different from the algorithm for Interval-Connectivity which only goes up during the construction of the first right ladder and goes right or down in all other cases. We will describe the algorithm for Stability with reference to Fig. 5 which shows an example of the execution of the algorithm. See Algorithm 3 for complete details.

The walk begins by constructing a right ladder. In each step, if a computed intersection graph $\mathcal{G}^k[j]$ is connected, and G_i, $i = j + k - 1$, is the most recently received graph, then the value k is returned to indicate that G_i is k-stable. Then the walk climbs one row in the hierarchy to $\mathcal{G}^{k+1}[j]$ which takes into consideration the next graph G_{i+1}. If a computed intersection graph $\mathcal{G}^k[j]$, $k > 1$, is disconnected, then the walk descends to the next graph in the row below, i.e. to $\mathcal{G}^{k-1}[j+1]$. In this case no value is returned because the next graph in $\mathcal{G}$ has not yet been considered. If a graph $\mathcal{G}^1[j]$ is disconnected, then 0 is returned, and the walk moves right to the next graph.

As in the previous algorithms, the right ladders are constructed incrementally as the walk goes up, even though each graph $\mathcal{G}^k[j]$ can be computed from $\mathcal{G}^{k-1}[j] \cap G_{j+k-1}$, because this prepares the ladders needed to compute the intersection graphs if the walk goes down. This is illustrated by the second right ladder $\mathcal{R}^5[7]$ in Fig. 5. If a disconnected graph $\mathcal{G}^k[j]$ is found while building a right ladder, the walk jumps to the next graph in the row just below, i.e. to $\mathcal{G}^{k-1}[j+1]$, to avoid unnecessary computations. For example, in Fig. 5 the walk jumps from $\mathcal{G}^5[7]$ which is disconnected to $\mathcal{G}^4[8]$ without computing $\mathcal{G}^7[5]$ and $\mathcal{G}^6[6]$.

If a graph $\mathcal{G}^k[j]$ cannot be computed using the current ladders, then a complete new left ladder $\mathcal{L}^k[k+j-1]$ is constructed as high as possible until it reaches a previously computed graph or until it encounters a disconnected graph. The former case is illustrated by the left ladder $\mathcal{L}^5[6]$ in Fig. 5 which is built when the walk descends from $\mathcal{G}^6[1]$ to $\mathcal{G}^5[2]$. The latter case is illustrated by the left ladder $\mathcal{L}^4[11]$ which encounters the disconnected graph $\mathcal{G}^4[8]$. In this case the walk resumes from the previous graph in the ladder ($\mathcal{G}^3[9]$ in the example). In contrast, a new left ladder is not needed when the walk descends three times from $\mathcal{G}^8[2]$ to $\mathcal{G}^5[5]$ because the ladders $\mathcal{L}^5[6]$ and $\mathcal{R}^3[7]$ that exist at this point can be used to compute these intersections.

In the example in Fig. 5, the sequence of values $T_1, T_2, T_3, \ldots, T_{14}$ that the algorithm outputs is $1, 2, 3, 4, 5, 5, 6, 7, 5, 6, 3, 4, 5, 6$.

```
i ← 1                                                        // current index in the row
next ← 2                                                     // trigger for next ladder construction
output ← 0

while receiving graphs do
    while isConnected(𝒢^k[i]) do
        output ← k; k ← k + 1
        computeFromIntersection(k, i, next)                  // "increment" the right ladder
    while ¬isConnected(𝒢^k[i]) do
        if k = 1 then
            output ← 0; next ← i + 2
        else
            k ← k − 1
        i ← i + 1
        if i = next then
            next ← i + k; computeFromRight(k, i, next)
        else
            computeFromIntersection(k, i, next)

function computeFromRight(k, i, next):                       // compute the left ladder ℒ^k[i]
    k' ← 1                                                   // row of first increment
    i' ← next − 1                                            // index of first increment
    while k' < k do
        if ¬isConnected(𝒢^{k'}[i']) then
            k ← k' − 1                                       // move the original walk..
            i ← i' + 1                                       // ..below-right disconnected graph,
            return                                           // abort function
        k' ← k' + 1; i' ← i' − 1
        𝒢^{k'}[i'] ← 𝒢^{k'−1}[i' + 1] ∩ G_{i'}                // "increment" the ladder

function computeFromIntersection(k, i, next):
    if i = next − 1 then
        𝒢^k[i] ← 𝒢^{k−1}[i] ∩ G_{i+k−1}
    else
        k' ← k − next + i                                    // row of increment (right ladder)
        𝒢^{k'}[next] ← 𝒢^{k'−1}[next] ∩ G_{next+k'−1}         // "increment" right ladder
        if ¬isConnected(𝒢^{k'}[next]) then
            i ← next; k ← k'
        else
            𝒢^k[i] ← 𝒢^{next−i}[i] ∩ 𝒢^{k'}[next]            // compute intersection based on Lemma 5
```

Algorithm 3: Optimal algorithm for Stability

The complexity analysis of the algorithm is similar to the analysis of the online algorithm for Interval-Connectivity. The number of intersection graphs in right ladders never exceeds the number of graphs received and the same is true for left ladders. Each intersection graph in a ladder is computed using one binary intersection operation. Each time the walk climbs in the intersection hierarchy, one connectivity test is performed and a single graph is processed. When the walk descends, a new graph in $\mathcal{G}$ is not processed, but the number of descents cannot exceed the number of ascents, and each descent uses at most one connectivity test. This results in a constant amortized cost for each received graph.

Theorem 5. Stability *can be solved with an amortized cost of* $\Theta(1)$ *elementary operations per graph received.*

6. Conclusions

In this paper, we studied the problem of testing whether a given dynamic graph $\mathcal{G} = (G_1, G_2, ..., G_\delta)$ is T-interval connected. We also considered the related problem of finding the largest T for which a given $\mathcal{G}$ is T-interval connected. We assumed that the dynamic graph $\mathcal{G}$ is a sequence of *independent* graphs and we investigated algorithmic solutions that use two elementary operations, *binary intersection* and *connectivity testing*, to solve the problems. We developed efficient algorithms that use only $O(\delta)$ elementary operations, asymptotically matching the lower bound of $\Omega(\delta)$. We presented PRAM algorithms that show that both problems can be solved efficiently in parallel, and online algorithms that use $\Theta(1)$ elementary operations per graph received. We also presented dynamic versions of the online algorithms that report connectivity based on recent network history.

In our study, we focused on algorithms using only the two elementary operations *binary intersection* and *connectivity testing*. This approach is suitable for a high-level study of these problems when the details of changes between successive graphs in a sequence are arbitrary. If the evolution of the dynamic graph is constrained in some ways (e.g., bounded number of changes between graphs), then one could benefit from the use of more sophisticated data structures to lower the complexity of the problem. Another natural extension of our investigation of T-interval connectivity would be a similar study for other classes of dynamic graphs, as identified in [4].

Distributed algorithms for all of these problems, in which a node in the graph only sees its local neighbourhood, would also be of interest. For example, distributed versions of the dynamic algorithms in Section 5 could be used to supplement the information available to distributed Internet routing protocols such as OSPF (Open-Shortest Path First) which are used to construct routing tables. Our dynamic algorithms have $\Theta(1)$ amortized complexity, and distributed versions with $\Theta(1)$ amortized complexity could provide real-time information about network connectivity to OSPF.

[1] M. Barjon, A. Casteigts, S. Chaumette, C. Johnen, and Y. M. Neggaz. Testing temporal connectivity in sparse dynamic graphs. *CoRR*, abs/1404.7634:8p, 2014. a French version appeared in Proc. of ALGOTEL (2014).

[2] B. Bui-Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. of Foundations of Computer Science*, 14(2):267–285, April 2003.

[3] A. Casteigts, S. Chaumette, and A. Ferreira. Characterizing topological assumptions of distributed algorithms in dynamic networks. In *Proc. of SIROCCO*, pages 126–140, Piran, Slovenia, 2009. Springer. (Full version in *CoRR, abs/1102.5529*).

[4] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *Int. J. of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012. doi: 10.1080/17445760.2012.668546.

[5] A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1988. ISBN 978-0-521-38841-2.

[6] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *Proc. of SIGCOMM*, pages 145–158, 2004.

[7] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. ISBN 0-201-54856-9.

[8] F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *Proc. of STOC*, pages 513–522, Cambridge, USA, 2010. ACM.

[9] R. O'Dell and R. Wattenhofer. Information dissemination in highly dynamic graphs. In *Proc. of DIALM-POMC*, pages 104–110, Cologne, Germany, 2005. ACM. ISBN 1-59593-092-2.

[10] J. Whitbeck, M. Dias de Amorim, V. Conan, and J.-L. Guillaume. Temporal reachability graphs. In *Proc. of MOBICOM*, pages 377–388. ACM, 2012.