Efficiently Testing $T$-Interval Connectivity in Dynamic Graphs

Arnaud Casteigts$^1$, Ralf Klasing$^1$, Yessin M. Neggaz$^1$, and Joseph G. Peters$^2$

$^1$ LaBRI, CNRS, University of Bordeaux, France
$^2$ 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 static graphs $G = \{G_1, G_2, ..., G_\delta\}$ 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 $G$ is $T$-interval connected for a given $T$. We also consider the related problem of finding the largest $T$ for which a given $G$ 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.

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 [8] 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.

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

* Part of this work was done while Joseph G. 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).

** A short version of this paper appeared in the proceedings of CIAC 2015.
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 (or algorithms) can be successfully solved (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 static graphs $G = \{G_1, G_2, ..., G_\delta\}$, each of which corresponds to a given time in the discrete domain (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].

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, for every $T$ consecutive graphs in the sequence $G$, there exists a common connected spanning subgraph. 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 that of [9] since a graph that is $T$-interval connected is obviously 1-interval connected. Hence, $T$-interval connectivity is more specialized in that sense.

In this paper, we look at the problem of deciding whether a given sequence $G$ is $T$-interval connected for a given $T$. We also consider the related problem of finding the largest $T$ for which the given $G$ 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, which are 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 at stake. 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 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 $\text{NC}$ (i.e. Nick's class). Finally, 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.

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 $G = \{G_1, G_2, \ldots, G_\delta\}$ of static graphs such that $G_i = (V, E_i)$ accounts for the network topology at (discrete) time $i$. Observe that $V$ is non-varying; only the set of edges varies. We consider undirected edges throughout the paper, which is the setting in which $T$-interval connectivity was originally introduced. The parameter $\delta$ is called the length of the sequence $G$. It corresponds to the number of time steps this graph covers.

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 $G$, 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 $G$ 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 $\{G_t, G_{t+1}, \ldots, G_{t+T-1}\}$ share a common connected spanning subgraph.

Testing $T$-Interval Connectivity. We will use $T$-INTERVAL-CONNECTIVITY to refer to the problem of deciding whether a dynamic graph $G$ is $T$-interval connected for a given $T$, and INTERVAL-CONNECTIVITY to refer to the problem of finding $\max\{T : G$ is $T$-interval connected$\}$ for a given $G$. 
Let $G^T = \{G_{1,T}, G_{2,T+1}, \ldots, G_{(\delta-T+1,T)}\}$. We call $G^T$ the $T$th row in $G$’s intersection hierarchy, as depicted in Fig. 1. A particular case is $G^1 = G$. For any $1 \leq i \leq \delta - T + 1$, we define $G^T[i] = G_{(i,i+T-1)}$. We call $G^T[i]$ the $i$th element of row $G^T$ and $i$ is called the index of $G^T[i]$ in row $G^T$.

**Observation 1.** A sequence of graphs $G$ is $T$-interval connected if and only if all graphs in $G^T$ are connected.

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

**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 static graphs, compute their intersection) and *connectivity testing* (given a static 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 an undirected graph can also be done in linear time in the number of edges by building a DFS tree from an arbitrary node and testing whether all nodes were reached through it. Hence, both operations have similar costs. In what follows, we will refer to them as elementary operations. One advantage of using them is that the high-level logic of the algorithm 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. Whenever a disconnected graph is found in row \(k\), the answer is \(k-1\). If all graphs are connected up to row \(\delta\), then \(\delta\) 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 \(\mathcal{G}\) 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 applied 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 over 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 \(\text{NC}\) (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 $G^k$ is already computed, then any row $G^\ell$ for $k + 1 \leq \ell \leq 2k$ can be computed with $O(\delta)$ elementary operations.

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

$T$-Interval-Connectivity. Using Lemma 2 we can incrementally compute rows $G^{2i}$ ("power rows") for all $i$ from 1 to $\lceil \log_2 T \rceil - 1$ without computing the intermediate rows. Then, we compute row $G^T$ directly from row $G^{2\lceil \log_2 T \rceil - 1}$ (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 $G^{2i}$ 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 $G^{2i + 1}$ is the first power row that contains a disconnected graph, and that $G^{2i}$ is the row computed before $G^{2i + 1}$. Next, we do a binary search among the rows between $G^{2i}$ and $G^{2i + 1}$ to find the row $G^T$ 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 $G^{2i}$ 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.

Fig. 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 $G^{16}$ which contains a disconnected graph ($\times$). A binary search between rows $G^9$ and $G^{16}$ is then used to find $G^{11}$, the highest row where all graphs are connected.

Now we establish that these problems are in $\text{NC}$ by showing that our algorithms are efficiently parallelizable.

Lemma 3. If some row $G^k$ is already computed, then any row between $G^{k+1}$ and $G^{2k}$ can be computed in $O(1)$ time on an EREW PRAM with $O(\delta)$ processors.
Proof. Assume that row $G^k$ is already computed, and that one wants to compute row $G^l$, consisting of the entries $G^l[1], \ldots, G^l[\delta - \ell + 1]$, for some $k + 1 \leq \ell \leq 2k$. Since $G^l[i] = G^k[i] \cap G^k[i + \ell - k]$ for $1 \leq i \leq \delta - \ell + 1$, the computation of row $G^l$ 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 $G^l[i]$. In the first round $P_i$ reads $G^k[i]$, and in the second round $P_i$ reads $G^k[i + \ell - k]$. This guarantees that each $P_i$ has exclusive access to the entries of row $G^k$ that it needs for its computation. Hence, row $G^l$ can be computed in $O(1)$ time on an EREW PRAM using $O(\delta)$ processors. □

$T$-Interval-Connectivity on an EREW PRAM. We compute the same rows in the same order as the sequential algorithm. We can use $O(\delta)$ processors in parallel to compute row $G^{2\ell+1}$ from row $G^{2\ell}$ in $O(1)$ time on an EREW PRAM by Lemma 3. The (potentially missing) row $G^T$ can also be computed in $O(1)$ time using $O(\delta)$ processors. Overall, we compute $O(\log \delta)$ rows, and all necessary 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 $G^T$ can be done in $O(1)$ time with $O(\delta)$ processors and then the processors can establish whether or not all graphs in row $G^T$ are connected in time $O(\log \delta)$. 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. 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. 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. The total time is $O(\log^2 \delta)$ on an EREW PRAM with $O(\delta)$ processors.

CRCW PRAM. Using standard techniques and results for computing the logical AND on a CRCW PRAM (see [5,7]), our results for an EREW PRAM can be improved to $O(1)$ time on a CRCW PRAM using $O(\delta^2)$ 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 $\delta$ of $G$), 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 3. The right ladder of length $l$ at index $i$, denoted by $R^l[i]$, is the sequence of intersection graphs $\{G^k[i], k = 1, 2, \ldots, l\}$. The left ladder of length $l$ at index $i$, denoted by $L^l[i]$, is the sequence $\{G^k[i - k + 1], k = 1, 2, \ldots, l\}$. 

Efficiently Testing $T$-Interval Connectivity in Dynamic Graphs  

7
A right (resp. left) ladder of length \( l - 1 \) at index \( i \) is said to be incremented when graph \( G^i[i] \) (resp. \( G^i[i + l - 1] \)) 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 (a symmetrical argument holds for left ladders). For any \( k \) in \([2, l]\) it holds that \( G^k[i] = G^{k-1}[i] \cap G_{i+k-1} \). Indeed, by definition, \( G^{k-1}[i] = \cap\{G_i, G_{i+1}, ..., G_{i+k-2}\} \). The ladder can thus be built bottom-up using a single new intersection at each level.

**Lemma 5.** Given a left ladder of length \( l_\ell \) at index \( i_\ell \) and a right ladder of length \( l_r \) at index \( i_r = i_\ell + 1 \). For any pair \((i, k)\) such that \( i_\ell - l_\ell \leq i < i_r \) and \( i_r - i < k \leq i_r - i + l_r \), \( G^k[i] \) can be computed by a single binary intersection, namely \( G^k[i] = G^{i_\ell - 1}[i] \cap G^{i_r - l_r}[i_r] \) (see example below).

**Proof.** By definition, \( G^k[i] = \cap\{G_i, G_{i+1}, ..., G_{i+k-1}\} \) and \( G^{i_\ell - 1}[i] \cap G^{i_r - l_r}[i_r] \). It follows that \( G^k[i] = G^{i_\ell - 1}[i] \cap G^{i_r - l_r}[i_r] \).

Informally, the constraints \( i_\ell - l_\ell \leq i < i_r \) and \( i_r - i < k \leq i_r - i + l_r \) 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 execution of the algorithm. (see details on Algorithm 1). The algorithm traverses the \( T \)th row in the intersection hierarchy from left to right, starting at \( G^T[1] \). If a disconnected graph is found, the algorithm returns \texttt{false} and terminates. If the algorithm reaches the last graph in the row, i.e. \( G^T[\delta - T + 1] \), and no disconnected graph was found, then it returns \texttt{true}. The graphs \( G^T[1], G^T[2], \ldots, G^T[\delta - T + 1] \) are computed based on the set of ladders \( S = \{L^T[T], R^T[T - 1][T + 1], L^T[2T], R^T[T - 1][2T + 1], \ldots\} \), which are constructed as follows. Each left ladder is built entirely (from bottom to top) when the traversal arrives at its top location in row \( T \) (i.e. where the last increment is to take place). For instance, \( L^T[T] \) is built when the walk is at index 1 in row \( T \), \( L^T[2T] \) is built at index \( T + 1 \), and so on. If a disconnected graph is found in the process, the execution terminates returning \texttt{false}. 

---

\( G \)
Algorithm 1: Optimal algorithm for $T$-INTERVAL-CONNECTIVITY

```
1 $k \leftarrow T$  // current row (non-changing)
2 $i \leftarrow 1$  // current index in the row
3 $next \leftarrow 1$  // trigger for next ladder construction

4 // walk until stepping out of the intersection hierarchy
5 while $i \leq \delta - k + 1$ do
6     if $i = next$ then
7         next $\leftarrow i + k$
8         if $\neg \text{computeFromRight}(k, i, next)$ then
9             return false
10        else
11            $i \leftarrow i + 1$
12        else
13            computeFromIntersection($k, i, next$)
14            $i \leftarrow i + 1$
15        if $\neg \text{isConnected}(G^k[i])$ then
16            return false
17    return true

18 function computeFromRight($k, i, next$):  // compute the left ladder $L^k[i]$
19     $k' \leftarrow 1$  // row of first increment
20     $i' \leftarrow next - 1$  // index of first increment
21     while $k' < k$ do
22         if $\neg \text{isConnected}(G^{k'}[i'])$ then
23             return false  // a disconnected graph was found
24         $k' \leftarrow k' + 1$
25         $i' \leftarrow i' - 1$
26     $G^{k'}[i'] \leftarrow G^{k'-1}[i' + 1] \cap G_{i'}$  // “increment” the ladder

27 function computeFromIntersection($k, i, next$):  // “increment” the right ladder
28     $k' \leftarrow k - next + i$  // row of increment (right ladder)
29     $G^{k'}[next] \leftarrow G^{k'-1}[next] \cap G_{next+k'-1}$  // “increment right ladder
30     $G^k[i] \leftarrow G^{next-i}[i] \cap G^{k'}[next]$  // compute intersection based on

Lemma[8]
```
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 will be built.

The set \( S \) of ladders constructed by this process includes at most \( \delta/T \) left ladders and \( \delta/T \) right ladders, each of length at most \( T \). By Lemma 4 the set of ladders \( S \) can be computed using less than \( 2\delta \) binary intersections. Based on Lemma 5 each of the \( \delta-T+1 \) graphs \( G^T[i] \) in row \( T \) can be computed at the cost of a single intersection of two graphs in \( S \). 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 \( O(\delta) \) elementary operations, which is optimal (up to 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)

The walk starts at the bottom left graph \( G^1[1] \) and builds a right ladder incrementally until it encounters a disconnected graph. If \( G^\delta[1] \) is reached and is connected, then \( \mathcal{G} \) is \( \delta \)-interval connected and execution terminates returning \( \delta \). 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 \( G^k[2] \) which is the next graph in row \( k \) that needs to be checked. This requires the construction of a left ladder \( L^k[k+1] \) of length \( k \) ending at \( G^k[2] \). 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. If the walk eventually reaches the rightmost graph of some row and this graph is connected, then the algorithm
Efficiently Testing $T$-Interval Connectivity in Dynamic Graphs

1 $k \leftarrow 1$ // current row
2 $i \leftarrow 1$ // current index in the row
3 $next \leftarrow 2$ // trigger for next ladder construction

4 // builds a right ladder until a disconnected graph is found
5 while isConnected($G^k[1]$) do
6     $k \leftarrow k + 1$
7     if $k > \delta$ then
8         return $\delta$ // the graph is $\delta$-interval connected
9     else
10        $G^k[1] \leftarrow G^{k-1}[1] \cap G_k$ // "increment" the right ladder
11    if $k = 1$ then
12        return 0 // the graph is 0-interval connected
13    $k \leftarrow k - 1$ // move down
14    $i \leftarrow i + 1$ // move right
15 // walk until stepping out of the hierarchy
16 while $i \leq \delta - k + 1$ do
17     if $i = next$ then
18         next $\leftarrow i + k$
19         computeFromRight($k, i, next$)
20     else
21         computeFromIntersection($k, i, next$)
22     if $\neg$isConnected($G^k[i]$) then
23         $k \leftarrow k - 1$
24     if $k = 0$ then
25         return 0
26     $i \leftarrow i + 1$
27 return $k$

28 function computeFromRight($k, i, next$): // compute the left ladder $L^k[i]$
29     $k' \leftarrow 1$ // row of first increment
30     $i' \leftarrow next - 1$ // index of first increment
31     while $k' < k$ do
32         if $\neg$isConnected($G^{k'}[i']$) then
33             $k \leftarrow k' - 1$ // move the original walk...
34             $i \leftarrow i' + 1$ // ...below-right disconnected graph,
35             return // abort function
36         $k' \leftarrow k' + 1$
37         $i' \leftarrow i' - 1$
38         $G^{k'}[i'] \leftarrow G^{k'-1}[i'+1] \cap G_{i'}$ // "increment" the ladder
39 function computeFromIntersection($k, i, next$): // Same function as for Algorithm 1
40     $k' \leftarrow k - next + i$ // row of increment (right ladder)
41     $G^{k'}[next] \leftarrow G^{k'-1}[next] \cap G_{next+k'-1}$ // "increment" right ladder
42     $G^k[i] \leftarrow G^{next-i}[i] \cap G^{k'}[next]$ // compute intersection based on Lemma 5

Algorithm 2: Optimal algorithm for INTERVAL-CONNECTIVITY
terminates returning the corresponding row number as $T$. Otherwise the walk will terminate at a disconnected graph in row 1 and $G$ is not $T$-interval connected for any $T$. In this case, the algorithm returns $T = 0$.

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

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

**Theorem 2.** Interval-Connectivity can be solved with $O(\delta)$ 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 static graphs $G_1, G_2, G_3, \ldots$ of a dynamic graph $G$ 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 $T$ static 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 static graphs (corresponding to a graph in row $T$) after the receipt of

\[ \text{Fig. 4. Example of the execution of the optimal algorithm for Interval-Connectivity. (It is a coincidence that the rightmost ladder matches the outer face.)} \]
The number of intersections performed to build left ladders exceed the number of static graphs received and the same is true for right ladders. Furthermore, each new static 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 static 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 $O(1)$ elementary operations per static graph received.*

## 5 Conclusions

In this paper, we studied the problem of testing whether a given dynamic graph $G = \{G_1, G_2, \ldots, G_δ\}$ is $T$-interval connected. We also considered the related problem of finding the largest $T$ for which a given $G$ is $T$-interval connected. We assumed that the dynamic graph $G$ is a sequence of independent static 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 $O(δ)$ elementary operations, asymptotically matching the lower bound of $Ω(δ)$. We also presented PRAM algorithms that show that both problems can be solved efficiently in parallel and online algorithms that use $O(1)$ elementary operations per static graph received.

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 is constrained in some ways (e.g. bounding the number of changes in-between graphs), then one could benefit from the use of more sophisticated data structures to lower the complexity of the problem.

A 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 vertex in the graph only sees its local neighbourhood, would also be of high interest.

## References

1. Barjon, M., Casteigts, A., Chaumette, S., Johnen, C., Neggaz, Y.M.: Testing temporal connectivity in sparse dynamic graphs. CoRR abs/1404.7634, 8p (2014), a French version appeared in Proc. of ALGOTEL (2014).
2. Bui-Xuan, B., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. Int. J. of Foundations of Computer Science 14(2), 267–285 (April 2003)
3. Casteigts, A., Chaumette, S., Ferreira, A.: Characterizing topological assumptions of distributed algorithms in dynamic networks. In: Proc. of SIROCCO. pp. 126–140. Springer, Piran, Slovenia (2009), (Full version in CoRR, abs/1102.5529).
4. Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-varying graphs and dynamic networks. Int. J. of Parallel, Emergent and Distributed Systems 27(5), 387–408 (2012)
5. Gibbons, A., Rytter, W.: Efficient parallel algorithms. Cambridge University Press (1988)
6. Jain, S., Fall, K., Patra, R.: Routing in a delay tolerant network. In: Proc. of SIGCOMM. pp. 145–158 (2004)
7. JáJá, J.: An Introduction to Parallel Algorithms. Addison-Wesley (1992)
8. Kuhn, F., Lynch, N., Oshman, R.: Distributed computation in dynamic networks. In: Proc. of STOC. pp. 513–522. ACM, Cambridge, USA (2010)
9. O’Dell, R., Wattenhofer, R.: Information dissemination in highly dynamic graphs. In: Proc. of DIALM-POMC. pp. 104–110. ACM, Cologne, Germany (2005)
10. Whitbeck, J., Dias de Amorim, M., Conan, V., Guillaume, J.L.: Temporal reachability graphs. In: Proc. of MOBICOM. pp. 377–388. ACM (2012)