Dynamic Compact Data Structure for Temporal Reachability with Unsorted Contact Insertions

LUIZ FERNANDO AFRA BRITO[1], MARCELO KEESE ALBERTINI[1], BRUNO AUGUSTO NASSIF TRAVENÇOLO[1] AND GONZALO NAVARRO[2]

[1] *Faculty of Computer Science (FACOM), Federal University of Uberlândia, Uberlândia, Brazil*
[2] *IMFD & Department of Computer Science (DCC), University of Chile, Santiago, Chile*
Email: luiz.brito@ufu.br

Temporal graphs represent interactions between entities over time. Deciding whether entities can reach each other through temporal paths is useful for various applications such as in communication networks and epidemiology. Previous works have studied the scenario in which addition of new interactions can happen at any point in time. A known strategy maintains, incrementally, a Timed Transitive Closure by using a dynamic data structure composed of $O(n^2)$ binary search trees containing non-nested time intervals. However, space usage for storing these trees grows rapidly as more interactions are inserted. In this paper, we present a compact data structure that represents each tree as two dynamic bit-vectors. In our experiments, we observed that our data structure improves space usage while having similar time performance for incremental updates when comparing with the previous strategy in temporally dense temporal graphs.

Keywords: Temporal Graph; Temporal Transitive Closure; Compact; Dynamic; Incremental

1. INTRODUCTION

Temporal graphs represent interactions between entities over time. These interactions often appear in the form of contacts at specific timestamps. Moreover, entities can also interact indirectly with each other by chaining several contacts over time. For example, in a communication network, devices that are physically connected can send new messages or propagate received ones; thus, by first sending a new message and, then, repeatedly propagating messages over time, remote entities can communicate indirectly. Time-respecting paths in temporal graphs are known as temporal paths, or simply journeys, and when a journey exists from one entity to another, we say that the first can reach the second.

In a computational environment, it is often useful to check whether entities can reach each other while using low space. Investigations on temporal reachability have been used, for instance, for characterizing mobile and social networks [1, 2], and for validating protocols and better understanding communication networks [3, 4]. Some other applications require the ability to reconstruct a concrete journey if one exists. Journey reconstruction has been used in applications such as finding and visualizing detailed trajectories in transportation networks [5, 6, 7], and matching temporal patterns in temporal graph databases [8, 9]. In all these applications, low space usage is important because it allows the maintenance of larger temporal graphs in primary memory.

In [10, 4], the authors considered updating reachability information given a chronologically sorted sequence of contacts. In this problem, a standard Transitive Closure (TC) is maintained as new contacts arrive. Differently, in [11, 12], the authors studied the problem in which sequences of contacts may be chronologically unsorted and queries may be intermixed with update operations. For instance, during scenarios of epidemics, outdated information containing interaction details among infected and non-infected individuals are reported in arbitrary order, and the dissemination process is continually queried in order to take appropriate measures against contamination [13, 14, 15, 16].

Particularly to our interest, the data structure proposed by [11] maintains a Timed Transitive Closure (TTC), a generalization of a TC that takes time into consideration. It maintains well-chosen sets of time intervals describing departure and arrival timestamps of journeys in order to provide time related queries and enable incremental updates on the data structure. The key idea is that, each set associated with a pair of vertices only contains non-nested time intervals and it is sufficient to implement all the TTC operations. In a temporal graph with n entities interacting over τ time units, this data structure maintains only $O(n^2\tau)$ intervals (as opposed to $O(n^2\tau^2)$) using $O(n^2)$ dynamic Binary Search Trees (BSTs). Although the reduction of intervals is interesting, the space to maintain $O(n^2)$ BSTs containing $O(\tau)$ intervals each can still be prohibitive for large temporal graphs.

In this paper, we propose a compact data structure to represent TTCs incrementally while answering reachability queries. Our new data structure maintains each set of non-nested time intervals as two dynamic bit-vectors, one for departure and the other for arrival timestamps. Each dynamic bit-vector uses the same data layout introduced in [17], which resembles a B$^+$-tree [18] with static bit-vectors as leaf nodes. Furthermore, we studied the usage of two variants of this dynamic bit-vector. One variant represents leaves of dynamic bit-vectors as raw bit sequences, and the other as the encoded distances between consecutive 1's present in raw bit sequences.

We proved that our novel data structure handles the insertion of new contacts in $O(n^2 d \log \tau)$ time, where d is the total number of intervals removed from any of the $O(n^2)$ BSTs; and answers basic reachability queries in time $O(\log \tau)$. As a comparison, the time complexities introduced in [11] offer $O(n^2 \log \tau)$ amortized time for insertion, and $O(\log \tau)$ time for basic queries. Also, we introduced a specialized insertion algorithm for the second variant, the one which represents leaves of dynamic bit-vector as encoded distances of consecutive 1's, that has the same time complexity of the data structure described in [11].

Even though the time complexity for inserting new contacts in a TTC using our general algorithm has an additional factor in the number of removed intervals, *i.e.*, $O(n^2 d \log(\tau))$, we show empirically that the average wall-clock times do not differ much from the previous approach [11]. Besides that, when using the first variant on temporally-denser temporal graphs, the space to construct random TTCs is much smaller. We also show that, for the second variant, the space needed to represent TTCs is generally smaller than the data struture introduced in [11], specially for temporally-sparser temporal graphs; however, the wall-clock time overhead on all operations is higher due to additional encoding/decoding steps.

1.1. Organization of the document

This paper is organized as follows. In Section 2, we briefly review the Timed Transitive Closure, the data structure introduced in [11], and the dynamic bit-vector proposed by [17]. In Section 3, we describe our data structure along with the algorithms for each operation. In Section 4, we conduct some experiments comparing our data structure with the previous work [11]. Finally, Section 5 concludes with some remarks and open questions such as the usage of an encoding or packing techniques for temporal very sparse temporal graphs.

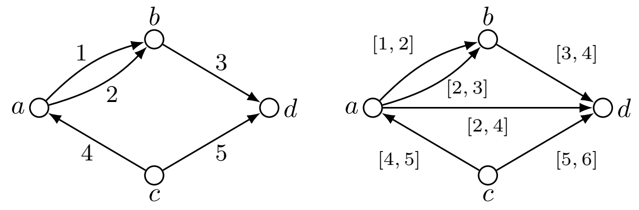


FIGURE 1. On the left, a temporal graph $\mathcal{G}$ on four vertices $V = \{a, b, c, d\}$, where the presence times of edges are depicted by labels. For $\delta = 1$, this temporal graph has only two non-trivial journeys, *i.e.* journeys with more than one contact, namely $\mathcal{J}_1 = \langle (a, b, 1), (b, d, 4) \rangle$ and $\mathcal{J}_2 = \langle (a, b, 2), (b, d, 4) \rangle$. On the right, the corresponding Timed Transitive Closure (TTC). Note that only the interval $\mathcal{I}_2 = [2, 4]$, regarding $\mathcal{J}_2$, is depicted on the edge from a to d because the other possibility, $\mathcal{I}_1 = [1, 4]$, regarding $\mathcal{J}_1$, encloses I_2. A query to check whether a reaches d within the time interval $\mathcal{I}_1$ can also be satisfied by using $\mathcal{I}_2$.

2. BACKGROUND

2.1. Timed Transitive Closure

Following the formalism in [19], a temporal graph is represented by a tuple $\mathcal{G} = (V, E, \mathcal{T}, \rho, \zeta)$ where: V is a set of vertices; $E \subseteq V \times V$ is a set of edges; $\mathcal{T}$ is the time interval over which the temporal graph exists (lifetime); $\rho : E \times \mathcal{T} \to \{0, 1\}$ is a function that expresses whether a given edge is present at a given time instant; and $\zeta : E \times \mathcal{T} \mapsto \mathbb{T}$ is function that expresses the duration of an interaction for a given edge at a given time, where $\mathbb{T}$ is the time domain. In this paper, we consider a setting where E is a set of directed edges, $\mathbb{T}$ is discrete such that $\mathcal{T} = [1, \tau] \subseteq \mathbb{T}$ is the lifetime containing τ timestamps, and $\zeta = \delta$, where δ is any fixed positive integer. Additionally, we call (u, v, t) a contact in $\mathcal{G}$ if $\rho((u, v), t) = 1$.

Reachability in temporal graphs can be defined in terms of journeys. A journey from u to v in $\mathcal{G}$ is a sequence of contacts $\mathcal{J} = \langle c_1, c_2, \ldots, c_k \rangle$, whose sequence of underlying edges form a valid (u, v)-path in the underlying graph G and, for each contact $c_i = (u_i, v_i, t_i)$, it holds that $t_{i+1} \geq t_i + \delta$ for $i \in [1, k-1]$. Throughout this article we use $departure(\mathcal{J}) = t_1$, and $arrival(\mathcal{J}) = t_k + \delta$. Thus, a vertex u can reach a vertex v within time interval $[t^-, t^+]$ iff there exists a journey $\mathcal{J}$ from u to v such that $t^- \leq departure(\mathcal{J}) \leq arrival(\mathcal{J}) \leq t^+$.

In [11], the authors introduced the Timed Transitive Closure (TTC), a transitive closure that captures the reachability information of a temporal graph within all possible time intervals. Informally, the TTC of a temporal graph $\mathcal{G}$ is a multigraph with time interval labels on edges. Each time interval expresses the $departure(\mathcal{J})$ and $arrival(\mathcal{J})$ timestamps of a journey $\mathcal{J}$ in $\mathcal{G}$ as its left and right boundaries, respectively. This additional information allows answering reachability queries parametrized by time intervals and also deciding if a new contact occurring anywhere in history can be composed with existing journeys. Furthermore, a TTC needs at most τ edges (in the same direction) between two vertices instead of τ^2 to perform basic operations. The key idea is that each set of intervals from these edge labels can be reduced to a set containing only non-nested time intervals. For instance, in the contrived example shown in Figure 1, we can see that, even though the information of an existing journey in the temporal graph was discarded in the corresponding TTC, a reachability query that could be satisfied by a "larger" interval can also be satisfied by a "smaller" nested interval.

Their data structure encodes TTCs as $n \times n$ matrices, in which every entry (u, v) points to a self-balanced Binary Search Tree (BST) denoted by $T(u, v)$. Each tree $T(u, v)$ contain up to τ intervals corresponding to the reduced edge labels from vertex u to vertex v in the TTC. As all these intervals are non-nested, one can use any of their boundaries (departure or arrival) as sorting key. This data structure supports the following operations: `add_contact(`u, v, t`)`, which updates information based on a new contact (u, v, t); `can_reach(`u, v, t^-, t^+`)`, which returns true if u can reach v within the interval $[t^-, t^+]$; `is_connected(`t^-, t^+`)`, which returns true if $\mathcal{G}$, restricted to the interval $[t^-, t^+]$, is temporally connected, *i.e.*, all vertices can reach each other within $[t^-, t^+]$; and `reconstruct_journey(`u, v, t^-, t^+`)`, which returns a journey (if one exists) from u to v occurring within the interval $[t^-, t^+]$. All these operations can be implemented using the following BST primitives, where $T_{(u,v)}$ is a BST containing reachability information regarding journeys from u to v:

- FIND_NEXT($T_{(u,v)}$, t), which retrieves from $T_{(u,v)}$ the earliest interval $[t^-, t^+]$ such that $t^- \geq t$, if any, and nil otherwise;
- FIND_PREV($T_{(u,v)}$, t), which retrieves from $T_{(u,v)}$ the latest interval $[t^-, t^+]$ such that $t^+ \leq t$, if any, and nil otherwise; and
- INSERT($T_{(u,v)}$, t^-, t^+), which inserts into $T_{(u,v)}$ a new interval $\mathcal{I} = [t^-, t^+]$ if no other interval $\mathcal{I}'$ such that $\mathcal{I}' \subseteq \mathcal{I}$ exists while removing all intervals $\mathcal{I}''$ such that $\mathcal{I} \subseteq \mathcal{I}''$.

The algorithm for `add_contact(`u, v, t`)` manages the insertion of a new contact (u, v, t) as follows. As shown in Algorithm 1, first, at line 1, the interval $[t, t+\delta]$, corresponding to the trivial journey $\mathcal{J}$ from u to v with $departure(\mathcal{J}) = t$ and $arrival(\mathcal{J}) = t + \delta$, is inserted in $T_{(u,v)}$ using the INSERT primitive, which runs in time $O(\log \tau)$. Then, the core of the algorithm consists of computing the indirect consequences of this insertion for the other vertices. Their algorithm consists of enumerating these compositions with the help of the FIND_PREV and FIND_NEXT primitives, which runs in time $O(\log \tau)$, and inserting them into the TTC using INSERT. From line 2 to 7, for every vertex $w^- \in \mathcal{N}^*_{in}(u)$, where $\mathcal{N}^*_{in}(u)$ retrieves all incoming neighbors of u in the TTC, it finds the latest interval $[t^-, \_]$ in $T_{(w^-,u)}$ that arrives before time t (inclusive) and inserts the composition $[t^-, t+\delta]$ into $T_{(w^-,v)}$. Similarly, from line 8 to 11, for every vertex $w^+ \in \mathcal{N}^*_{out}(v)$, where $\mathcal{N}^*_{out}(v)$ retrieves all outgoing neighbors of v in the TTC, the algorithm finds the earliest interval $[\_, t^+]$ in $T_{(v,w^+)}$ that leaves v after time $t + \delta$ (inclusive), and inserts the composition $[t, t^+]$ into $T_{(u,w^+)}$. Finally, from line 12 to 14, for all $w^- \in \mathcal{N}^*_{in}(u)$ and $w^+ \in \mathcal{N}^*_{out}(v)$, it inserts the composition $[t^-, t^+]$ into $T_{(w^-,w^+)}$ when appropriate. As there can only be one new interval for each pair of vertices, the algorithm takes $O(n^2 \log \tau)$ time.

Algorithm 1 `add_contact(`u, v, t`)`

1: INSERT($T_{(u,v)}$, t, $t+\delta$)
2: $D \leftarrow \{\}$
3: **for all** $w^- \in \mathcal{N}^*_{in}(u)$ **do**
4:     $[t^-, \_] \leftarrow$ FIND_PREV($T_{(w^-,u)}$, t)
5:     **if** $t^- \neq nil$ **then**
6:         INSERT($T_{(w^-,v)}$, t^-, $t+\delta$)
7:         $D \leftarrow D \cup (w^-, t^-)$
8: **for all** $w^+ \in \mathcal{N}^*_{out}(v)$ **do**
9:     $[\_, t^+] \leftarrow$ FIND_NEXT($T_{(v,w^+)}$, $t+\delta$)
10:     **if** $t^+ \neq nil$ **then**
11:         INSERT($T_{(u,w^+)}$, t, t^+)
12:         **for all** $(w^-, t^-) \in D$ **do**
13:             **if** $w^- \neq w^+$ **then**
14:                 INSERT($T_{(w^-,w^+)}$, t^-, t^+)

The algorithm for `can_reach(`u, v, t^-, t^+`)` consists of testing whether $T_{(u,v)}$ contains at least one interval included in $[t^-, t^+]$. The cost of this algorithm reduces essentially to calling FIND_NEXT($T_{(u,v)}$, t^-) once, which takes $O(\log \tau)$ time. The algorithm for `is_connected(`t^-, t^+`)` simply calls `can_reach(`u, v, t^-, t^+`)` for every pair of vertices; therefore, it takes $O(n^2 \log \tau)$ time.

For `reconstruct_journey(`u, v, t^-, t^+`)`, the `add_contact(`u, v, t`)` algorithm must include an additional field along every time interval added to BSTs indicating which vertex comes next in (at least one of) the possible journeys. This modification can be trivially implemented and do not change the time complexity of `add_contact(`u, v, t`)`. The algorithm consists of unfolding intervals and successors, one pair at a time using the FIND_NEXT primitive, which must also consider the new successor field added to intervals, until the completion of the resulting journey of length k; therefore, it takes $O(k \log \tau)$ time in total.

In this paper, we develop a new compact data structure that supports `add_contact(`u, v, t`)` and `can_reach(`u, v, t^-, t^+`)`, and, consequently, `is_connected(`t^-, t^+`)`. Our new data structure do not account for the modification needed to answer `reconstruct_journey(`u, v, t^-, t^+`)`, although it should be possible using a different compact data structure to store successors.

2.2. Dynamic bit-vectors

A bit-vector B is a data structure that holds a sequence of bits and provides the following operations: $\text{ACCESS}(B, i)$, which accesses the bit at position i; $\text{RANK}_b(B, i)$, which counts the number of b's until (and including) position i; and $\text{SELECT}_b(B, j)$, which finds the position of the j-th bit with value b. In this paper, we consider that $\text{SELECT}_b(B, j) = len(B) + 1$ when $j > \text{RANK}_b(B, len(B))$, where $len(B)$ is the length of the bit-vector B. It is a fundamental data structure to design more complex data structures such as compact sequence of integers, text, trees, and graphs [20, 21]. Usually, bit-vectors are static, meaning that we first construct the data structure from an already known sequence of bits in order to take advantage of these query operations.

Additionally, a dynamic bit-vector allows changes on the underlying bits. Although many operations to update a dynamic bit-vector has been proposed, the following are the most commonly used: $\text{INSERT}_b(B, i)$, which inserts a bit b at position i; $\text{UPDATE}_b(B, i)$, which writes the new bit b to position i; and $\text{REMOVE}(B, i)$, which removes the bit at position i. Apart from these operations, there are others such as $\text{INSERT\_WORD}_w(B, i)$, which inserts a word w at position i, and $\text{REMOVE\_WORD}_n(B, i)$, which removes a word of n bits from position i.

In [17], the authors proposed a dynamic data structure for bit-vectors with a layout similar to B^+-trees [18]. Leaves wrap static bit-vectors of maximum length l and internal nodes contain at most m pointers to children along with the number of 1's and the total number of bits in each subtree. With exception to the root node, static bit-vectors have a minimum length of $\lceil l/2 \rceil$ and internal nodes have at least $\lceil m/2 \rceil$ pointers to children. These parameters serve as rules to balance out tree nodes during insertion and removal of bits. Figure 2 illustrates the overall layout of this data structure.

Any static bit-vector representation can be used as leaves, the simplest one being arrays of words representing bits explicitly. In this case, the maximum length could be set to $l = \Theta(w^2)$, where w is the integer word size, and, therefore, processing leaves would cost $O(w) = O(\log n)$ time, *i.e.* the time to process at most w words, where n is the size of the dynamic bit-vector. Another possibility is to represent bit-vectors sparsely by computing the distances between consecutive 1's and then encoding them using an integer compressor such as Elias-Delta [22], or simply packing them using binary packing [23] to reduce the amount of unused bits. In this case, we can instead use as parameter the maximum number of 1's encoded by static bit-vectors to balance out leaves. By setting the maximum numbers of 1's in leaves to w, the time to process leaves would also cost $O(w)$ since encoding/decoding a single distance using such compressors takes $O(1)$ time.

The data structure introduced in [17] supports all main dynamic bit-vector operations in time $O(\log n)$ (using the mentioned parameters) since the costs are dominated by the processing of leaves. An $\text{ACCESS}(B, i)$ operation is done by traversing the tree starting at the root node. In each node the algorithm searches from left to right for the branch that has the i-th bit and subtracts from i the number of bits in previous subtrees. After traversing to the corresponding child node, the new i is local to that subtree and the search continues until reaching the leaf containing the i-th bit. At a leaf node, the algorithm simply accesses and returns the i-th local bit in the corresponding static bit-vector. If bits in static bit-vectors are encoded sparsely, for instance, an additional decoding step is necessary.

The $\text{RANK}_b(B, i)$ and $\text{SELECT}_b(B, j)$ operations are similar to $\text{ACCESS}(B, i)$. For $\text{RANK}_b(B, i)$, the algorithm also sums the number of 1's in previous subtrees when traversing the tree. At a leaf, it finally sums the number of 1's in the corresponding static bit-vector up to the i-th local bit using `popcount` operations, which counts the number of 1's in a word, and return the resulting value. For $\text{SELECT}_b(B, j)$, the algorithm instead uses the number of 1's in each subtree to guide the search. Thus, when traversing down, it subtracts the number of 1's in previous subtrees from j, and sums the total number of bits. At a leaf, it searches for the position of the j-th local set bit using `clz` or `ctz` operations, which counts, respectively, the number of leading and trailing zeros in a word; sums it, and returns the resulting value.

The algorithm for $\text{INSERT}_b(B, i)$ first locates the leaf that contains the static bit-vector with the i-th bit. During this top-down traversal, it increments the total number of bits and the number of 1's, whether $b = 1$, in each internal node key associated with the child it descends. Then, it reconstructs the leaf while including the new bit b. If the leaf becomes full, the algorithm splits its content into two bit-vectors and updates its parent accordingly while adding a new key and a pointer to the new leaf. After this step, the parent node can also become full and, in this case, it must also be split into two nodes. Therefore, the algorithm must traverse back, up to the root node, balancing any node that becomes full. If the root node becomes full, then it creates a new root containing pointers to the split nodes along with the keys associated with both subtrees.

The algorithm for $\text{REMOVE}(B, i)$ also has a top-down traversal to locate and reconstruct the appropriate leaf, and a bottom-up phase to rebalance tree nodes. However, internal node keys associated with the child it descends must be updated during the bottom-up phase since the i-bit is only known after reaching the corresponding leaf. Moreover, a node can become empty when it has less than half the maximum number of entries. In this case, first, the algorithm tries to share the content of siblings with the current node while updating parent keys. If sharing is not possible, it merges a sibling into

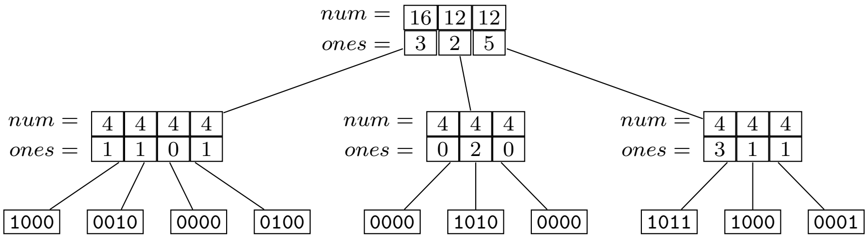


FIGURE 2. A dynamic bit-vector using the data structure introduced in [17]. Leaves wrap static bit-vectors and internal nodes contain pointers to children along with the number of 1's and the total number of bits in each of them. The maximum number of pointers in each internal node m and the length of each static bit-vector n in this example is 4.

the current node and updates its parent while removing the key and pointer previously related to the merged node. If the root node becomes empty, the algorithm removes the old root and makes its single child the new root.

The $\textsc{update}_b(B,\ i)$ operation can be implemented by calling $\textsc{remove}(B,\ i)$ then $\textsc{insert}_b(B,\ i)$, or by using a similar strategy with a single traversal.

3. DYNAMIC COMPACT DATA STRUCTURE FOR TEMPORAL REACHABILITY

Our new data structure uses roughly the same strategy as in the previous work [11]. The main difference is the usage of a compact dynamic data structure to maintain a set of non-nested intervals instead of Binary Search Trees (BSTs). This compact representation provides all BST primitives in order to incrementally maintain Temporal Transitive Closures (TTCs) and answer reachability queries. In [11], the authors defined them as follows, where $T_{(u,v)}$ represents a BST holding a set of non-nested intervals associated with the pair of vertices (u, v). (1) $\textsc{find\_next}(T_{(u,v)},\ t)$ returns the earliest interval $[t^-, t^+]$ in $T_{(u,v)}$ such that $t^- \geq t$, if any, and nil otherwise; (2) $\textsc{find\_prev}(T_{(u,v)},\ t)$ returns the latest interval $[t^-, t^+]$ in $T_{(u,v)}$ such that $t^+ \leq t$, if any, and nil otherwise; and (3) $\textsc{insert}(T_{(u,v)},\ t^-,\ t^+)$ inserts the interval $[t^-, t^+]$ in $T_{(u,v)}$ and performs some operations for maintaining the property that all intervals in $T_{(u,v)}$ are minimal.

For our new compact data structure, we take advantage that every set of intervals only contains non-nested intervals, thus we do not need to consider other possible intervals. For instance, if there is an interval $\mathcal{I} = [4, 6]$ in a set, no other interval starting at timestamp 4 or ending at 6 is possible, otherwise, there would be some interval $\mathcal{I}'$ such that $\mathcal{I}' \subseteq \mathcal{I}$ or $\mathcal{I} \subseteq \mathcal{I}'$. Therefore, we can represent each set of intervals as a pair of dynamic bit-vectors D and A, one for departure and the other for arrival timestamps. Both bit-vectors must provide the following low-level operations: $\textsc{access}(B,\ i)$, $\textsc{rank}_b(B,\ i)$, $\textsc{select}_b(B,\ j)$, $\textsc{insert}_b(B,\ i)$, and $\textsc{update}_b(B,\ i)$.

By using these simple bit-vectors operations, we first introduce algorithms for the primitives $\textsc{find\_next}((D, A)_{(u,v)},\ t)$, $\textsc{find\_prev}((D, A)_{(u,v)},\ t)$ and $\textsc{insert}((D, A)_{(u,v)},\ t^-,\ t^+)$ that runs, respectively,

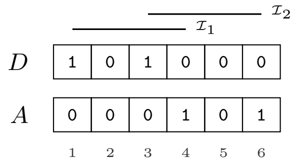


FIGURE 3. Representation of a set of non-nested interval using two bit-vectors, one for departures and the other for arrival timestamps. In this example, a set containing the intervals $[1, 4]$ and $[3, 6]$ is represented by the first bit-vector containing 1's at position 1 and 3, and the second bit-vector containing 1's at positions 4 and 6. Note that both bit-vectors must have the same number of 1's, otherwise, there would be an interval with missing values for departure or arrival.

in time $O(\log \tau)$, $O(\log \tau)$ and $O(d \log \tau)$, where d is the number of intervals removed during an interval insertion. Note that, now, these operations receive as first argument a pair containing two bit-vectors D and A associated with the pair of vertices (u, v) instead of a BST $T_{(u,v)}$. If the context is clear, we will simply use the notation (D, A) instead of $(D, A)_{(u,v)}$.

Then, in order to improve the time complexity of $\textsc{insert}((D, A)_{(u,v)},\ t^-,\ t^+)$ to $O(\log \tau + d)$, we propose a new bit-vector operation: $\textsc{unset\_one\_range}(B,\ j_1,\ j_2)$, which clears all bits in the range $[\textsc{select}_1(B,\ j_1), \textsc{select}_1(B,\ j_2)]$.

3.1. Compact representation of non-nested intervals

Each set of non-nested intervals is represented as a pair of dynamic bit-vectors D and A, one storing departure timestamps and the other arrival timestamps. Given a set of non-nested intervals $\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_k$, where $I_i = [d_i, a_i]$, D contains 1's at every position d_i, and A contains 1's at every position a_i. Figure 3 depicts this representation.

3.2. Query algorithms

Algorithms 2 and 3 answer the primitives $\textsc{find\_prev}((D, A),\ t)$ and $\textsc{find\_next}((D, A),\ t)$, respectively. In order to find a previous interval, at line 1, Algorithm 2 first counts in j how many 1's exist up to position t in A. If $j = 0$, then there is no interval $I = [t^-, t^+]$ such that $t^+ \leq t$, therefore, it returns nil. Otherwise, at lines 4 and 5, the algorithm computes the positions of the j-th 1's in D and A to compose the resulting intervals. In order to find a next interval, at line 1, Algorithm 3 first counts in j' how many 1's exist up to time $t-1$ in D. If $j' = \text{RANK}_1(D,\ len(D))$, then there is no interval $I' = [t'^-, t'^+]$ such that $t' \leq t^-$, therefore, it returns nil. Otherwise, at lines 4 and 5, the algorithm computes the positions of the $(j'+1)$-th 1's in D and A to compose the resulting interval.

Algorithm 2 FIND_PREV((D, A), t)

1: $j \leftarrow \text{RANK}_1(A,\ t)$
2: **if** $j = 0$ **then**
3:     **return** nil
4: $t^- \leftarrow \text{SELECT}_1(D,\ j)$
5: $t^+ \leftarrow \text{SELECT}_1(A,\ j)$
6: **return** $[t^-, t^+]$

Algorithm 3 FIND_NEXT((D, A), t)

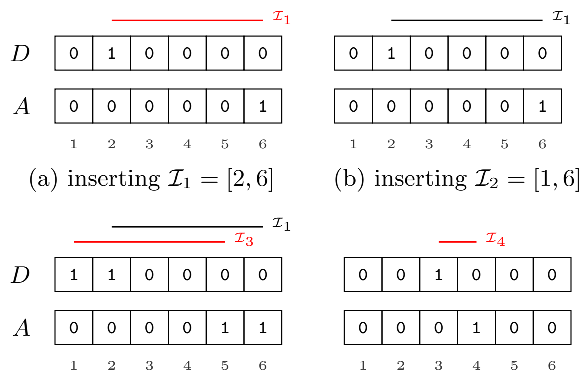
1: $j \leftarrow \text{RANK}_1(D,\ t-1)$
2: **if** $j = \text{RANK}_1(D,\ len(D))$ **then**
3:     **return** nil
4: $t^- \leftarrow \text{SELECT}_1(D,\ j+1)$
5: $t^+ \leftarrow \text{SELECT}_1(A,\ j+1)$
6: **return** $[t^-, t^+]$

As $\text{RANK}_1(B,\ i)$ and $\text{SELECT}_1(B,\ j)$ on dynamic bit-vectors have time complexity $O(\log \tau)$ using the data structure proposed by [17], FIND_PREV((D, A), t) and FIND_NEXT((D, A), t) have both time complexity $O(\log \tau)$.

3.2.1. Interval insertion

Due to the property of non-containment of intervals, given a new interval $\mathcal{I} = [t^-, t^+]$, we must first assure that there is no other interval $\mathcal{I}'$ in the data structure such that $\mathcal{I}' \subseteq \mathcal{I}$, otherwise, $\mathcal{I}$ cannot be present in the set. Then, we must find and remove all intervals $\mathcal{I}''$ in the data structure such that $\mathcal{I} \subseteq \mathcal{I}''$. Finally, we insert $\mathcal{I}$ by setting the t^--th bit of bit-vector D and the t^+-th bit of A. Figure 4 illustrates the process of inserting new intervals.

Algorithm 4 describes a simple process for the primitive INSERT((A, D), t^-, t^+) in order to insert a new interval $\mathcal{I} = [t^-, t^+]$ into a set of non-nested intervals encoded as two bit-vectors D and A. At line 1, it computes how many 1's exist in D prior to position t^- by calling $r_d = \text{RANK}_1(D,\ t^- - 1)$ and access the t^--th bit in D by calling $bit_d = \text{ACCESS}(D,\ t^-)$. At line 2, it computes the same information with respect to the bit-vector A and timestamp t^+ by calling $r_a = \text{RANK}_1(A,\ t^+ - 1)$ and $bit_a = \text{ACCESS}(A,\ t^+)$. We note that the operations $\text{RANK}_1(B,\ i)$ and $\text{ACCESS}(B,\ i)$ can be processed in a single tree traversal using the dynamic



(a) inserting $\mathcal{I}_1 = [2, 6]$ (b) inserting $\mathcal{I}_2 = [1, 6]$

(c) inserting $\mathcal{I}_3 = [1, 5]$ (d) inserting $\mathcal{I}_4 = [3, 4]$

FIGURE 4. Sequence of insertions using our data structure based on bit-vectors D and A. In (a), our data structure is empty, thus, the insertion of interval $\mathcal{I}_1 = [2, 6]$ results in setting the position 2 in D and 6 in A. Then, in (b), the new interval $\mathcal{I}_2 = [1, 6]$ encloses $\mathcal{I}_1$, therefore, the insertion is skipped. Next, in (c), no interval encloses or is enclosed by the new interval $\mathcal{I}_3 = [1, 5]$, thus, it suffices to set the position 1 in D and 5 in A. Finally, in (d), the new interval $\mathcal{I}_4 = [3, 4]$ is enclosed by $\mathcal{I}_1$ and $\mathcal{I}_3$, thus both of them are removed by clearing the corresponding bits and then $\mathcal{I}_4$ is inserted by setting the position 3 in D and 4 in A.

bit-vector described in [17]. If r_d is less than $r_a + bit_a$, then there are more intervals closing up to timestamp t^+ than intervals opening before t^-, therefore, there is some interval $\mathcal{I}' = [d', a']$ such that $t^- \leq d' \leq a' \leq t^+$, *i.e.*, $\mathcal{I}' \subseteq \mathcal{I}$. In this case, the algorithm stops, otherwise, it proceeds with the insertion. When proceeding, if $r_d + bit_d$ is greater than r_a, then there are more intervals opening up to t^- than intervals closing before t^+, therefore, there are $d = (r_d + bit_d) - r_a$ intervals $\mathcal{I}''_i = [d''_i, a''_i]$, such that $d''_i \leq t^- \leq t^+ \leq a''_i$, *i.e.*, $\mathcal{I} \subseteq \mathcal{I}''_i$, that must be removed. From lines 5 to 9, the algorithm removes the d intervals that contain $\mathcal{I}$ by iteratively unsetting their corresponding bits in D and A. In order to unset the j-th 1 in a bit-vector B, we first search for its position by calling $i = \text{SELECT}_1(B,\ j)$, then update $B[i] = 0$ by calling $\text{UPDATE}_0(B,\ i)$. Thus, the algorithm calls $\text{UPDATE}_0(D,\ \text{SELECT}_1(D,\ r_a + 1))$ and $\text{UPDATE}_0(A,\ \text{SELECT}_1(A,\ r_a + 1))$ d times to remove the d intervals that close after (and including) t^+. Finally, at lines 10 and 11, the algorithm inserts $\mathcal{I}$ by calling $\text{UPDATE}_1(D,\ t^-)$ and $\text{UPDATE}_1(A,\ t^+)$. Note that both bit-vectors can grow with new insertions, thus we need to assure that both bit-vectors are large enough to accommodate the new 1's. That is why the algorithm calls *ensureCapacity* before setting the corresponding bits. The *ensureCapacity* implementation may call $\text{INSERT}_0(B,\ len(B))$ or $\text{INSERT\_WORD}_0(B,\ len(B))$ until B has enough space. Moreover, $\text{RANK}_1(B,\ i)$ and $\text{ACCESS}(B,\ i)$ operations can also receive positions that are larger than the actual length of B. In such cases, these operations must instead return $\text{RANK}_1(B,\ len(B))$ and 0, respectively.

Algorithm 4 $\textsc{insert}((D, A),\ t^-,\ t^+)$

```
1: r_d ← RANK_1(D, t⁻ − 1); bit_d ← ACCESS(D, t⁻)
2: r_a ← RANK_1(A, t⁺ − 1); bit_a ← ACCESS(A, t⁺)
3: if r_d ≥ r_a + bit_a then
4:     if r_d + bit_d > r_a then
5:         r_d⁺ ← r_d + bit_d
6:         while r_d⁺ > r_a do
7:             UPDATE_0(D, SELECT_1(D, r_a + 1))
8:             UPDATE_0(A, SELECT_1(A, r_a + 1))
9:             r_d⁺ ← r_d⁺ − 1
10:    ensureCapacity(D, t⁻); UPDATE_1(D, t⁻)
11:    ensureCapacity(A, t⁺); UPDATE_1(A, t⁺)
```

Theorem 3.1. *The update operation has worst-case time complexity $O(d \log \tau)$, where d is the number of intervals removed.*

Proof. All operations on dynamic bit-vectors have time complexity $O(\log \tau)$ using the data structure proposed by [17]. As the maximum length of each bit-vector is τ, the cost of *ensureCapacity* is amortized to $O(1)$ during a sequence of insertions. Therefore, the time complexity of $\textsc{insert}((D, A),\ t^-,\ t^+)$ is $O(d \log \tau)$ since in the worst case Algorithm 4 removes d intervals from line 6 to 9 before inserting the new one at lines 10 and 11. □

This simple strategy has a multiplicative factor on the number of removed intervals. In general, as more intervals in $[1, \tau]$ are inserted, the number of intervals d to be removed decreases, thus, in the long run, the runtime of this naïve solution is acceptable. However, when static bit-vectors are encoded sparsely as distances between consecutive 1's, the algorithm needs to decode/encode leaves d times and thus runtime may degrade severely. In the next section, we propose a new operation for dynamic bit-vectors using sparse static bit-vectors as leaves, $\textsc{unset\_one\_range}(B,\ j_1,\ j_2)$, to replace this iterative approach and improve the time complexity of $\textsc{insert}((D, A),\ t^-,\ t^+)$ to $O(\log \tau)$.

Corollary 3.1. *The* `add_contact(u, v, t)` *operation has worst-case time complexity $O(n^2 d \log \tau)$, where d is the number of intervals removed.*

Using Theorem 3.1, it trivially follows that an algorithm for `add_contact(u, v, t)`, *i.e.*, an algorithm that inserts a new contact into a TTC, has time complexity $O(n^2 d \log \tau)$ since it calls Algorithm 4 at most n^2 times in order to update, in the worst case, all BSTs.

3.3. New dynamic bit-vector operation to improve interval insertion

In this section, we propose a new operation $\textsc{unset\_one\_range}(B,\ j_1,\ j_2)$ for dynamic bit-vectors using sparse static bit-vectors as leaves to improve the time complexity of $\textsc{insert}((D, A),\ t^-,\ t^+)$. This new operation clears all bits starting from the j_1-th 1 up to the j_2-th 1 in time $O(\log \tau)$. Our algorithm for $\textsc{unset\_one\_range}(B,\ j_1,\ j_2)$, based on the split/join strategy commonly used in parallel programs [24], uses two internal functions $\textsc{split\_at\_jth\_one}(N,\ j)$ and $\textsc{join}(N_1,\ N_2)$. The $\textsc{split\_at\_jth\_one}(N,\ j)$ function takes a root node N representing a dynamic bit-vector B and splits its bits into two nodes N_1 and N_2 representing bit-vectors B_1 and B_2 containing, respectively, the bits in range $[1, \textsc{select}_1(B,\ j) - 1]$ and $[\textsc{select}_1(B,\ j), len(B)]$. Note that $[1, \textsc{select}_1(B,\ j) - 1]$ is invalid when $\textsc{select}_1(B,\ j) = 1$ and $[\textsc{select}_1(B,\ j), len(B)]$ is invalid when $\textsc{select}_1(B,\ j) = len(B)+1$. In such cases, the respective output node represents an empty bit-vector. The $\textsc{join}(N_1,\ N_2)$ function takes two root nodes N_1 and N_2, representing two bit-vectors B_1 and B_2 and constructs a new tree with root node N representing a bit-vector B containing all bits from B_1 followed by all bits from B_2. The resulting trees for both functions must preserve the balancing properties of dynamic bit-vectors [17].

Thus, given a dynamic bit-vector B represented as a tree with root N, our algorithm for $\textsc{unset\_one\_range}(B,\ j_1,\ j_2)$ is described as follows. First, the algorithm calls $\textsc{split\_at\_jth\_one}(N,\ j_1)$ in order to split the bits in B into two nodes N_{left} and N_{tmp} representing two bit-vectors containing, respectively, the bits in range $[1, \textsc{select}_1(B,\ j_1) - 1]$ and in range $[\textsc{select}_1(B,\ j_1), len(B)]$. Then, it calls $\textsc{split\_at\_jth\_one}(N_{tmp},\ \ j_2 - j_1 + 2)$ to split N_{tmp} further into two nodes N_{ones} and N_{right} containing, respectively the bits in range $[\textsc{select}_1(B,\ j_1), \textsc{select}_1(B,\ j_2 + 1) - 1]$, and $[\textsc{select}_1(B,\ j_2 + 1), len(B)]$. The tree with root node N_{ones} contains all 1's previously in the original dynamic bit-vector B that should be cleared. In the next step, the algorithm creates a new tree with root node N_{zeros} containing $len(N_{ones})$ 0's to replace N_{ones}. Finally, it calls $\textsc{join}(\textsc{join}(N_{left},\ N_{zeros}),\ N_{right})$ to join the trees with root nodes N_{left}, N_{zeros}, and N_{right} into a final tree representing the original bit-vector B with the corresponding 1's cleared.

Note that the tree with root N_{ones} is still in memory, thus it needs some sort of cleaning. The cost of immediately cleaning this tree would increase proportionally to the total number of nodes in N_{ones} tree. Instead, we keep N_{ones} in memory and reuse its children lazily in other operations that request node allocations so that the cost of cleaning is amortized.

Moreover, even though we need to create a new bit-vector filled with zeros, this operation is performed in $O(1)$ time with a sparse implementation. Using the dynamic bit-vector introduced in [17], a new bit-vector filled with k zeros is constructed as follows when leaves are represented sparsely. Create a root node containing a key composed of the total number of bits equals to k and the total number of 1's equals to 0, and associate it with a pointer to a static bit-vector. Then, initialize the static bit-vector with the encodings of the distances between consecutive 1's. In this case, as there is no bit set, the static bit-vector is initialized with no encoding. However, we note that an implementation could encode the bit sequence in leaves as if there was an additional 1 appended to the end of the bit sequence in order to avoid edge cases in other operations [20]. Extraneous 1's will not change the results of other operations because the correct lengths of leaves are stored in their parents and, therefore, traversals would not descend to leaves looking for extraneous 1's. When doing so, the leaf would consist of a single encoding of the number k.

Next we describe JOIN(N_1, N_2) and SPLIT_AT_JTH_ONE(N, j). The idea of JOIN(N_1, N_2) is to merge the root of the smallest tree with the correct node of the highest tree and rebalance the resulting tree recursively.

Algorithm 5 JOIN(N_1, N_2)

1: **if** $height(N_1) = height(N_2)$ **then**
2:     **return** $mergeOrGrow(N_1, N_2)$
3: **else if** $height(N_1) > height(N_2)$ **then**
4:     $R \leftarrow$ JOIN($extractRightmostChild(N_1)$, N_2)
5:     **if** $height(R) = height(N_1)$ **then**
6:         **return** $mergeOrGrow(N_1, R)$
7:     $insertRightmostChild(N_1, R)$
8:     **return** N_1
9: **else**
10:     $R' \leftarrow$ JOIN(N_1, $extractLeftmostChild(N_2)$)
11:     **if** $height(R') = height(N_2)$ **then**
12:         **return** $mergeOrGrow(R', N_2)$
13:     $insertLeftmostChild(N_2, R')$
14:     **return** N_2

Algorithm 5 details the JOIN(N_1, N_2) recursive function. If $height(N_1) = height(N_2)$, at line 2, the algorithm tries to merge keys and pointers present in N_1 and N_2 if possible, or distributes their content evenly and grow the resulting tree by one level. This process is done by calling $mergeOrGrow(N_1, N_2)$, which returns the root node of the resulting tree. Instead, if $height(N_1) > height(N_2)$, at line 4, the algorithm first extracts the rightmost child from N_1, by calling $extractRightmostChild(N_1)$, and then recurses further passing the rightmost child instead. The next recursive call might perform a merge operation or grow the resulting subtree one level; therefore, the output node R may have, respectively, height equal to $height(N_1) - 1$ or $height(N_1)$. If the resulting tree grew, *i.e.*, $height(R) = height(N_1)$, then, at line 6, the algorithm returns the result of $mergeOrGrow(N_1, R)$. Otherwise, if a merge operation was performed, *i.e.*, $height(R) = height(N_1) - 1$, then, at line 7, it inserts R into N_1 as its new rightmost child, and returns N_1. Finally, if $height(N_1) < height(N_2)$, at line 10, the algorithm extracts the leftmost child from N_2 by calling $extractLeftmostChild(N_2)$ and recurses further passing the leftmost child instead. Similarly, the root R' resulted from the next recursive call might have height equal to $height(N_2) - 1$ or $height(N_2)$. If $height(R') = height(N_2)$, then, at line 12, the algorithm returns the result of calling $mergeOrGrow(R', N_2)$, otherwise, if $height(R') = height(N_2) - 1$, then, at line 13, it inserts R' into N_2 as its new leftmost child, and returns N_2. Note that all subroutines must properly update keys describing the length and number of 1's of the bit-vector represented by the corresponding child subtree. For instance, a call to $rightmost = extractRightmostChild(N)$ must decrement from the key associated with N the length and number of 1's in the bit-vector represented by $rightmost$.

Lemma 3.1. *The operation* JOIN(N_1, N_2) *has time complexity* $O(|height(N_1) - height(N_2)|)$.

Proof. Algorithm 5 descends at most $|height(N_1) - height(N_2)|$ levels starting from the root of the highest tree. At each level, in the worst case, it updates a node doing a constant amount of work equals to the branching factor of the tree. Therefore, the cost of JOIN(N_1, N_2) is $O(|height(N_1) - height(N_2)|)$. □

The idea of SPLIT_AT_JTH_ONE(N, j) is to traverse N recursively while partitioning and joining its content properly until it reaches the node containing the j-th 1 at position SELECT$_1$(B, j). During the forward traversal, it partitions the current subtree in two nodes N_1 and N_2, excluding the entry associated with the child to descend. Then, during the backward traversal, it joins N_1 and N_2, respectively, with the left and right nodes resulting from the recursive call.

Algorithm 6 SPLIT_AT_JTH_ONE(N, j)

1: **if** N is leaf **then**
2:     $(N_1, N_2) \leftarrow partitionLeaf(N, j)$
3:     **return** (N_1, N_2)
4: $(N_1, child, N_2) \leftarrow partitionNode(N, j)$
5: $(N_1', N_2') \leftarrow$ SPLIT_AT_JTH_ONE($child$, $j - ones(N_1)$)
6: **return** (JOIN(N_1, N_1'), JOIN(N_2', N_2))

The details of this function is shown in Algorithm 6. From lines 1 to 3, the algorithm checks whether the root is a leaf. If it is the case, it partitions the current bit-vector $B_1 \cdot b \cdot B_2$, where b is the j-th 1, and returns two nodes containing, respectively, B_1 and $b \cdot B_2$. Otherwise, from lines 4 to 6, the algorithm first finds the i-th child that contains the j-th 1 using a linear search and partitions the current node into three other nodes: N_1, containing the partition with all keys and children in range $[1, i-1]$; $child$, which is the child node associated with position i; and N_2, containing the partition with all keys and children in range $[i+1, \ldots]$. Then, at line 5, it recursively calls SPLIT_AT_JTH_ONE($child$, $j - ones(N_1)$) to retrieve the partial results N_1' containing bits from *child* up to the j-th 1; and N_2' containing bits from *child* starting at the j-th 1 and forward. Note that the next recursive call expects an input j that is local to the root node *child*. Finally, at line 6 it joins N_1 with N_1' and N_2' with N_2, and returns the resulting trees.

Lemma 3.2. *The operation* split_at_jth_one$(N,\ j)$ *has time complexity* $O(\log \tau)$.

Proof. As join$(N_1,\ N_2)$ has cost $O(|height(N_1) - height(N_2)|)$ and the sum of height differences for every level cannot be higher than the resulting tree height containing $n < \tau$ nodes, the time complexity of split_at_jth_one$(N,\ j)$ is $O(\log \tau)$. □

Furthermore, since join$(N_1,\ N_2)$ outputs a balanced tree when concatenating two already balanced trees, both trees resulting from the split_at_jth_one$(N,\ j)$ calls are also balanced.

Lemma 3.3. *The operation* unset_one_range$(B,\ j_1,\ j_2)$ *has time complexity* $O(\log \tau)$ *when* B *encodes leaves sparsely.*

Proof. The unset_one_range$(B,\ j_1,\ j_2)$ operation calls split_at_jth_one and join twice. It must also create a new tree containing select$_1(B,\ j_2 - 1)$ − select$_1(B,\ j_1)$ 0's to replace the subtree containing $j_2 - j_1$ 1's. If leaves of B are represented sparsely, then the creation of a new tree filled with 0's costs $O(1)$ since the resulting tree only has a root node, with its only key having the current length (select$_1(B,\ j_2 - 1)$ - select$_1(B,\ j_1)$), and an empty leaf. Therefore, as the cost of split_at_jth_one$(N,\ j)$, $O(\log \tau)$, dominates the cost of join$(N_1,\ N_2)$, the time complexity of unset_one_range$(B,\ j_1,\ j_2)$ is $O(\log \tau)$. □

By using the operation unset_one_range$(B,\ j_1,\ j_2)$, we can implement a new algorithm for the primitive insert$((D, A),\ t^-,\ t^+)$ when the dynamic bit-vectors D and A represent their leaves sparsely. Our previous general algorithm, Algorithm 4, first computes r_d = rank$_1(D,\ t^- - 1)$, bit_d = access$(D,\ t^-)$, r_a = rank$_1(A,\ t^+ - 1)$ and bit_a = access$(A,\ t^+)$. This information is used in order to check whether $[t^-, t^+]$ must be inserted into the set of intervals, and, if so, whether there is any interval already present in the set that needs to be removed. When the interval $[t^-, t^+]$ must be inserted in the set, $r_d + bit_d$ refers to how many intervals open before (and including) t^- and r_a to how many intervals close before t^+. If $r_d + bit_d < r_a$ then there are $d = (r_d + bit_b) - r_a$ intervals that close after (and including) t^+ to be removed, whose first interval is associated with the $(r_a + 1)$-th bit set in both bit-vectors D and A. Therefore, a new algorithm for the primitive insert$((D, A),\ t^-,\ t^+)$, should replace the loop in Algorithm 4, from lines 6 to 9, with the calls unset_one_range$(D,\ r_a + 1,\ r_a + d)$ and unset_one_range$(A,\ r_a + 1,\ r_a + d)$.

Theorem 3.2. *The primitive* insert$((D, A),\ t^-,\ t^+)$ *has time complexity* $O(\log \tau)$ *when* D *and* A *encode leaves sparsely.*

Proof. Following from Theorem 3.1 and Lemma 3.3, the loop in Algorithm 4 that iteratively unset d bit-vector bits can be substituted by the calls unset_one_range$(D,\ r_a + 1,\ r_a + d)$ and unset_one_range$(A,\ r_a + 1,\ r_a + d)$, where $d = (r_d + bit_b) - r_a$. Note that r_a, r_d and bit_b are already computed at lines 1 and 2 in Algorithm 4. As the cost of Algorithm 4 is dominated by this loop, its time complexity reduces to $O(\log \tau)$. □

Corollary 3.2. *The* `add_contact(u, v, t)` *operation has worst-case time complexity* $O(n^2 \log \tau)$ *when all BSTs, of the form* (D, A)*, uses dynamic bit-vectors* D *and* A *such that leaves are encoded sparsely.*

4. EXPERIMENTS

In this section, we conduct experiments to analyze the time performance and space efficiency of data structures when adding new information from synthetic datasets. The main motivation is to better understand, empirically, the trade-offs between the data structure described in [11] and two variants of the compact data structures described in this paper. All three data structure, including the one described in [11], maintains a set o

.

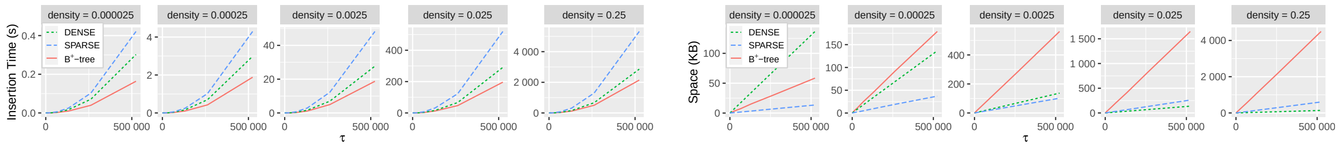
In Section 4.1, we compare our compact data structure that maintain a set of non-nested intervals directly with an in-memory B$^+$-tree implementation storing intervals as keys. For our compact data structure, we used dynamic bit-vectors [17] with leaves storing bits explicitly as arrays of integer words with words being 64 bits long. Internal nodes have a maximum number of pointers to children $m = 32$ and leaf nodes have static bit-vectors with maximum length $l = 4096$. For the B$^+$-tree implementation we used $m = 32$ for all nodes. In Section 4.2, we compare the overall Temporal Transitive Closure (TTC) data structure using our new compact data structure with the TTC using the B$^+$-tree implementation for each pair of vertices. All code is available at `https://bitbucket.org/luizufu/zig-ttc/src/master/`.

All experiments were execute in a AMD Ryzen 9 5950X 16-Core Processor with 64GB of RAM.

4.1. Comparison of data structures for sets of non-nested intervals

For this experiment, we created datasets containing all $O(\tau^2)$ possible intervals in $[1, \tau]$ for $\tau \in [2^3, 2^{14}]$. Then, for each dataset, we executed 10 times a program that shuffles all intervals at random, and inserts them into the tested data structure while gathering the wall-clock time and memory space usage after every insertion.

Figure 5(a) shows the average wall-clock time to insert all intervals into the both data structures as



(a) Total wall-clock time (b) Space after insertion

FIGURE 5. Comparison of incremental data structures to represent a set of non-nested intervals. In (a), the total wall-clock time to insert all intervals randomly shuffled into data structures. In (b), the total space used by the data structure after the insertion of all intervals.

τ increases. Figure 5(b) shows the cumulative wall-clock time to insert all intervals and the memory usage throughout the lifetime of a single program execution with $\tau = 2^{14}$. As shown in Figure 5(a), our new data structure slightly underperforms when compared with the B$^+$-tree implementation. However, as shown in Figure 5(b), the wall-clock time has a higher overhead at the beginning of the execution (first quartile) and, after that, the difference between both data structures remains almost constant. This overhead might be due to insertions of 0's at the end of the bit-vectors in order to make enough space to accommodate the rightmost interval inserted so far. We can also see in Figure 5(b) that the space usage of our new data structure is much smaller than the B$^+$-tree implementation. It is worth noting that, if the set of intervals is very sparse, maybe the use of sparse bit-vector as leaves could decrease the space since it does not need to preallocate most of the tree nodes, however, the wall-clock time could increase since at every operation leaves need to be decoded/unpacked and encoded/packed.

4.2. Comparison of data structures for Time Transitive Closures

For this experiment, we created datasets containing all $O(n^2\tau)$ possible contacts fixing the number of vertices $n = 32$ and the latency to traverse an edge $\delta = 1$ while varying $\tau = [2^3, 2^{14}]$. Then, for each dataset, we executed 10 times a program that shuffles all contacts at random, and inserts them into the tested TTC data structure while gathering the wall-clock time and memory space usage after every insertion.

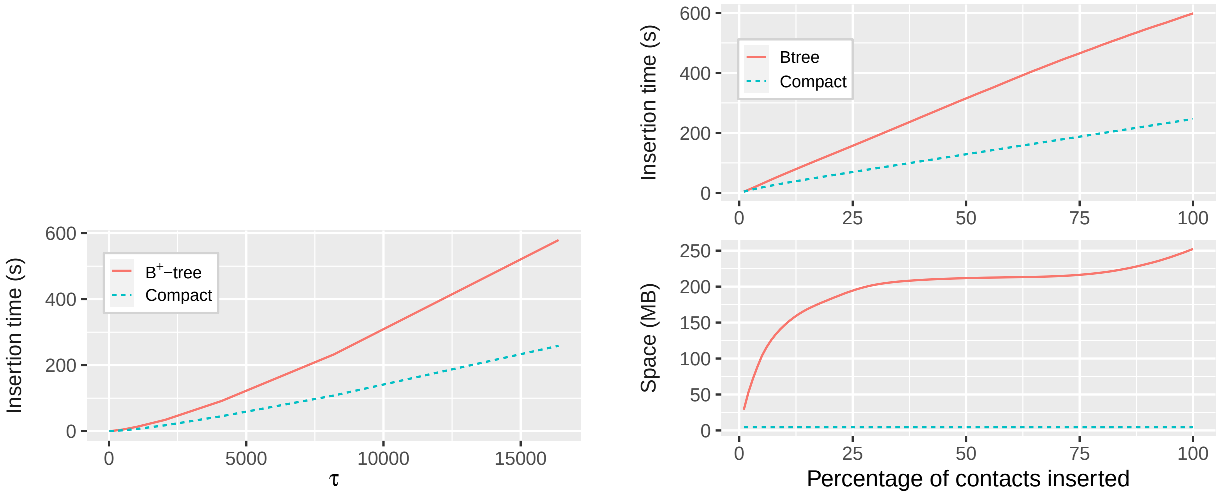
Figure 6(a) shows the average wall-clock time to insert all contacts into the TTCs using both data structures as τ increases. Figure 6(b) shows the cumulative wall-clock time to insert all contacts and the memory usage throughout the lifetime of a single program execution with $n = 32$ and $\tau = 2^{14}$. As shown in Figure 6(a), the TTC version that uses our compact data structure in fact outperforms when compared with the TTC that uses the B$^+$-tree implementation for large values of τ. In Figure 6(b), we can see that the time to insert a contact into the TTC using our new data structure is lower during almost all lifetime, and the space usage followed the previous experiment comparing data structures in isolation.

5. CONCLUSION AND OPEN QUESTIONS

We presented in this paper an incremental compact data structure to represent a set of non-nested time intervals. This new data structure is composed by two dynamic bit-vectors and works well using common operations on dynamic bit-vectors. Among the operations of our new data structures are: FIND_PREV$((A, D), t)$, which retrieves the previous interval $[t^-, t^+]$ such that $t^+ \leq t$ in time $O(\log \tau)$; FIND_NEXT$((A, D), t)$, which retrieves the next interval $[t^-, t^+]$ such that $t^- \geq t$ also in time $O(\log \tau)$; and INSERT$((A, D), t^-, t^+)$, which inserts a new interval $\mathcal{I} = [t^-, t^+]$ if no other interval $\mathcal{I}'$ such that $\mathcal{I}' \subseteq \mathcal{I}$ exists while removing all intervals $\mathcal{I}''$ such that $\mathcal{I} \subseteq \mathcal{I}''$ in time $O(d \log \tau)$, where d is the number of intervals removed. Moreover, we introduced a new operation UNSET_ONE_RANGE(B, j_1, j_2) for dynamic bit-vectors that encode leaves sparsely, which we used to improve the time complexity of our insert algorithm to $O(\log \tau)$.

Additionally, we used our new data structure to incrementally maintain Temporal Transitive Closures (TTCs) using much less space. We used the same strategy as described in [11], however, instead of using Binary Search Trees (BSTs), we used our new compact data structure. The time complexities of our algorithms for the new data structure are the same as those for BSTs. However, as we showed in our experiments, using our new data structure greatly reduced the space usage for TTCs in several cases and, as they suggest, the wall-clock time to insert new contacts also improves as τ increases.

For future investigations, we conjecture that our compact data structure can be simplified further so that the content of both its bit-vectors are merged into a single data structure. Our current insertion algorithm duplicates most operations in order to update both bit-vectors. Furthermore, each of these operations traverse a tree-like data structure from top to bottom. With a single tree-like data structure, our insertion algorithm could halve the number of traversals and, maybe, benefit from a better spatial locality. In another direction, our algorithm for INSERT$((A, D), t^-, t^+)$ only has time complexity $O(\log \tau)$ when both A and D encode leaves



(a) Overall

(b) Execution for $n = 32$ and $\tau = 2^{14}$

FIGURE 6. Comparison of Temporal Transitive Closures (TTCs) using incremental data structures to represent sets of non-nested intervals for each pair of vertices. In (a), the overall average wall-clock time to insert all possible $O(n^2\tau)$ contacts randomly shuffled into data structures. In (b), the cumulative wall-clock time and the memory space usage to insert all possible $O(n^2\tau)$ contacts randomly shuffled throughout a single execution. Note that the final wall-clock time of the execution described in (b) was one of the 10 executions with $\tau = 2^{14}$ used to construct (a).

sparsely. Perhaps, a dynamic bit-vector data structure that holds a mix of leaves represented densely or sparsely can be employed to retain the $O(\log\tau)$ complexity while improving the overall runtime for other operations. Lastly, we expect soon to evaluate our new compact data structure on larger datasets and under other scenarios; for instance, in very sparse and real temporal graphs.

ACKNOWLEDGEMENTS

This study was financed in part by Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001* - under the "CAPES PrInt program" awarded to the Computer Science Post-graduate Program of the Federal University of Uberlândia, as well as the Conselho Nacional de Desenvolvimento Científico e Tecnológico (grants 402543/2021-1, 306436/2022-1, 406418/2021-7, and 306795/2022-1). GN was supported by ANID – Millennium Science Initiative Program – Code ICN17_002 and Fondecyt grant 1-230755, Chile.

REFERENCES

[1] Tang, J., Musolesi, M., Mascolo, C., and Latora, V. (2010) Characterising temporal distance and reachability in mobile and online social networks. *SIGCOMM Computer Communication Review*, **40**, 118—124.

[2] Linhares, C. D. G., Ponciano, J. R., Paiva, J. G. S., Travençolo, B. A. N., and Rocha, L. E. C. (2019) Visualisation of structure and processes on temporal networks. In Holme, P. and Saramäki, J. (eds.), *Computational Social Sciences*. Springer International Publishing, Cham.

[3] Cacciari, L. and Rafiq, O. (1996) A temporal reachability analysis. *Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6. 1 International Symposium on Protocol Specification, Testing and Verification*, Warsaw, Poland, June 1995, pp. 35–49. Springer US, Boston, MA.

[4] Whitbeck, J., de Amorim, M. D., Conan, V., and Guillaume, J.-L. (2012) Temporal reachability graphs. *Mobicom '12: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Istanbul, Turkey, 22-26 August, pp. 377––388. Association for Computing Machinery, New York, NY, USA.

[5] Wu, G., Ding, Y., Li, Y., Bao, J., Zheng, Y., and Luo, J. (2017) Mining spatio-temporal reachable regions over massive trajectory data. *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, San Diego, CA, USA, 19-22 April, pp. 1283–1294. IEEE, New York, NY, USA.

[6] George, B., Kim, S., and Shekhar, S. (2007) Spatio-temporal network databases and routing algorithms: A summary of results. *Advances in Spatial and Temporal Databases*, Boston, MA, USA, 16-18 July, pp. 460–477. Springer Berlin, Heidelberg.

[7] Zeng, W., Fu, C.-W., Arisona, S. M., Erath, A., and Qu, H. (2014) Visualizing mobility of public transportation system. *IEEE Transactions on Visualization and Computer Graphics*, **20**, 1833–1842.

[8] Moffitt, V. Z. and Stoyanovich, J. (2016). Querying evolving graphs with portal. arXiv preprint. http://arxiv.org/abs/1602.00773.

[9] Latapy, M., Viard, T., and Magnien, C. (2018) Stream graphs and link streams for the modeling of interactions

over time. *Social Network Analysis and Mining*, **8**, 1–29.

[10] Barjon, M., Casteigts, A., Chaumette, S., Johnen, C., and Neggaz, Y. M. (2014) Testing temporal connectivity in sparse dynamic graphs. *2nd AETOS Int. conference on Research challenges for future RPAS/UAV systems*, Bordeaux, France, 9-10 September. http://arxiv.org/abs/1404.7634.

[11] Brito, L. F. A., Albertini, M. K., Casteigts, A., and Travençolo, B. A. N. (2022) A dynamic data structure for temporal reachability with unsorted contact insertions. *Social Network Analysis and Mining*, **12**, 22.

[12] Wu, H., Huang, Y., Cheng, J., Li, J., and Ke, Y. (2016) Reachability and time-based path queries in temporal graphs. *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, Helsinki, Finland, 16-20 May, pp. 145–156. IEEE, New York, NY, USA.

[13] Ponciano, J. R., Vezono, G. P., and Linhares, C. D. G. (2021) Simulating and visualizing infection spread dynamics with temporal networks. *Proceedings of the 36th Brazilian Symposium on Databases*, Rio de Janeiro, RJ, Brazil, 04-08 October. Sociedade Brasileira de Computação, Porto Alegre, RS, Brazil.

[14] Xiao, H., Rozenshtein, P., Tatti, N., and Gionis, A. (2018) Reconstructing a cascade from temporal observations. *Proceedings of the 2018 SIAM International Conference on Data Mining (SDM)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[15] Enright, J., Meeks, K., Mertzios, G. B., and Zamaraev, V. (2021) Deleting edges to restrict the size of an epidemic in temporal networks. *Journal of Computer and System Sciences*, **119**, 60–77.

[16] Rozenshtein, P., Gionis, A., Prakash, B. A., and Vreeken, J. (2016) Reconstructing an epidemic over time. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, CA, USA, 13-17 August, pp. 1835—1844. Association for Computing Machinery, New York, NY, USA.

[17] Prezza, N. (2017) A Framework of Dynamic Data Structures for String Processing. *16th International Symposium on Experimental Algorithms (SEA 2017)*, London, UK, pp. 11:1–11:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[18] Abel, D. J. (1984) A b+-tree structure for large quadtrees. *Computer Vision, Graphics, and Image Processing*, **27**, 19–31.

[19] Casteigts, A., Flocchini, P., Quattrociocchi, W., and Santoro, N. (2011) Time-varying graphs and dynamic networks. *AD-HOC, Mobile, and Wireless Networks*, Paderborn, Germany, 18-20 July, pp. 346–359. Springer Berlin, Heidelberg.

[20] Navarro, G. (2016) *Compact Data Structures: A Practical Approach*, 1st edition. Cambridge University Press, USA.

[21] Caro, D., Rodriguez, M. A., Brisaboa, N. R., and Farina, A. (2016) Compressed k^d-tree for temporal graphs. *Knowledge and Information Systems*, **49**, 553–595.

[22] Elias, P. (1975) Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, **21**, 194–203.

[23] Lemire, D. and Boytsov, L. (2015) Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, **45**, 1—29.

[24] Blelloch, G. E., Ferizovic, D., and Sun, Y. (2016) Just join for parallel ordered sets. *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, Pacific Grove, CA, USA, 11-13 July, pp. 253—264. Association for Computing Machinery, New York, NY, USA.