Noname manuscript No.
(will be inserted by the editor)

Computing Small Temporal Modules in Time Logarithmic in History Length

Binh-Minh Bui-Xuan ·
Hugo Hourcade ·
Cédric Miachon

Received: date / Accepted: date

Abstract A temporal graph $\mathcal{G}$ is a sequence of static graphs indexed by a set of integers representing time instants. Given Δ an integer, a Δ-module is a set of vertices A having the same neighbourhood outside of A for Δ consecutive instants. We address specific cases of Δ-module enumeration, when $|A| = 2$ or when $\Delta = \infty$. Our main parameter for time complexity analysis is the history length $\tau = \max\{t : G_t \in \mathcal{G} \text{ not empty }\} - \min\{t : G_t \in \mathcal{G} \text{ not empty }\}$. Using red-black tree data structure, we give solutions to above enumeration problems in time logarithmic in τ. For the general Δ-module enumeration problem, we give a pre-processing using overlapping properties of minimal Δ-modules. Numerical analysis of our implementation on graphs collected from real world data scales up to a history length of 10^8 time instants[1].

Keywords graph theory · historical data · modular decomposition · temporal graph

Supported by Courtanet – Sorbonne Université convention C19.0665 and ANRT grant 2019.0485.

Binh-Minh Bui-Xuan, LIP6 (CNRS – Sorbonne Université), Paris, France
E-mail: buixuan@lip6.com

Hugo Hourcade, Courtanet, LIP6 (CNRS – Sorbonne Université), Paris, France
E-mail: hugo.hourcade@lip6.fr; hugo.hourcade@lesfurets.com

Cédric Miachon, Courtanet, Paris, France
E-mail: cedric.miachon@lesfurets.com

[1] This paper encompasses the results on temporal twins presented in [1], and extends them to temporal modules, by a careful analysis of overlapping minimal temporal modules. Our source code is open at `https://github.com/DaemonFire/deltaModules`.

1 Introduction

When retrieving information from a historical dataset, the time instant when a piece of information is recorded in the database can be as important as the recorded data itself. This can globally be modelled in the formalism of a link stream, a time varying graph, a temporal graph or an evolving graph [2,3,6,13]. These notions occur in as various use cases as transportation timetables [4,7,11], navigation programs [14], email exchanges [12], proximity interactions [16], and many other types of datasets [17]. Let us consider a temporal graph to be a sequence of classical graphs $\mathcal{G} = (G_t)_{t \in T}$. The main focus of our paper can be understood as the retrieval of a specific behaviour represented by (part of) the temporally evolving neighbourhood of a given vertex v. For convenience, we call this vertex the *deviant vertex* and the temporal neighbourhood under question the *deviant behaviour*.

In the most simple form of deviant behaviour detection, we would like to enumerate all vertices which record the same neighbourhood as v at any time instant in the dataset. Formally, we define a pair of *eternal twins* $\{u, w\} \in \binom{V(\mathcal{G})}{2}$, where $V(\mathcal{G}) = \cup_{t \in T} V(G_t)$ and $V(G_t)$ the vertex set of G_t, as a pair of vertices for which the neighbourhoods $N_t(u)$ and $N_t(w)$ are strictly equal in $V(\mathcal{G}) \setminus \{u, w\}$ for every instant $t \in T$. Here, $N_t(u)$ is the set of all vertices adjacent to u in graph G_t. Note that twins in a static graph is the base case for modular decomposition, which, among other things, helps in reducing both space and time complexity of graph problems, see *e.g* [5,8,15] for a broad survey. The enumeration problem associated to our temporal case is defined as follows.

EternalTwinsOfAGivenVertex
Input : A temporal graph $\mathcal{G}$; a deviant vertex $v \in V(\mathcal{G})$.
Output : A list of all vertices forming a pair of eternal twins with v in $\mathcal{G}$.

Vertices which form eternal twins with the deviant vertex are those having exactly the same behaviour at any time instant as the deviant vertex. In reality, this is very strict. Eternal twins are more likely data duplicates than genuine peers of the deviant vertex. An example of such twins is given in Figure 3 where 1 and 2 have the exact same neighbourhood for the entire history of the temporal graph, that we denote by $\tau = \max\{t : G_t \in \mathcal{G} \text{ not empty }\} - \min\{t : G_t \in \mathcal{G} \text{ not empty }\}$.

It is in this sense that we consider in the present paper the following notions of Δ-modules and eternal modules. Both on the other hand represent a partial likeliness between vertices which are not duplicate per se. For an integer Δ, a *Δ-module* is a set $A \subseteq V(\mathcal{G})$ where there exists a time t_0 when, for all $u, w \in A$, $N_t(u)$ and $N_t(w)$ are strictly equal in $V(\mathcal{G}) \setminus A$ for Δ consecutive instants $t_0 \leq t < t_0 + \Delta$ with $[\![t_0, t_0 + \Delta[\![\subseteq T$. When $|A| = 2$, we call the Δ-module a *pair of Δ-twins*. When $\Delta = \tau$, we call the τ-module an *eternal module*. Naturally, outside the corresponding Δ time window, the behaviour of u and w can be different. Moreover, even for eternal modules, the behaviour of u and w can be very different inside the module A, especially when $|A|$

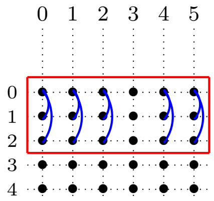

Fig. 1 In this link stream, vertices 0,1 and 2 (represented by rows with the corresponding identifiers) have exactly the same links to other vertices at every instants from instant 0 to instant $5 = \tau - 1$. They form a eternal-module.

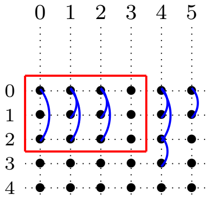

Fig. 2 In this link stream, vertices 0,1 and 2 (represented by rows with the corresponding identifiers) have exactly the same links to other vertices for instants 0, 1, 2, 3. They form a Δ-module for any $\Delta \leq 4$.

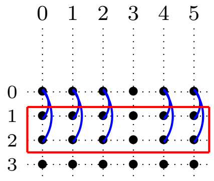

Fig. 3 In this link stream, vertices 1 and 2 (represented by rows with the corresponding identifiers) have exactly the same links to other vertices at every instants from instant 0 to instant $5 = \tau - 1$. They form a pair of eternal-twins.

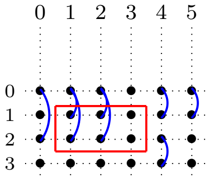

Fig. 4 In this link stream, vertices 1 and 2 (represented by rows with the corresponding identifiers) have exactly the same links to other vertices for instants 1, 2 and 3. They form a pair of Δ-twins for any $\Delta \leq 3$.

is large. In Figure 1, the set $\{0, 1, 2\}$ exemplifies an eternal module, whereas $\{0, 1, 2\}$ is a 4-module in Figure 2 and $\{1, 2\}$ is a 3-module in Figure 4. The associated enumeration problems are defined as follows.

Δ-ModulesContainingAVertex
Input : A temporal graph $\mathcal{G}$; a deviant vertex $v \in V(\mathcal{G})$.
Output : A list of all Δ-modules containing v in $\mathcal{G}$.

EternalModulesContainingAVertex
Input : A temporal graph $\mathcal{G}$; a deviant vertex $v \in V(\mathcal{G})$.
Output : A list of all eternal modules containing v in $\mathcal{G}$.

In a temporal graph representing the activity of users on a e-trade website, pairs of Δ-twins can for example characterize users displaying a similar purchase behaviour on a given period, underlining a pattern in purchase cycle, thus allowing purchase recommendations based on this local sameness. In a similar way, eternal modules could represent a likeliness of vertices who are not duplicate per se as their behaviour inside the module can differ widely. Those modules can therefore represent a population who always display the same behaviour toward the exterior while internal disparities may exist. Such sets can for example regroup people in a community having the same interactions with people outside of their group, not considering interactions internal to the group. In a social media context, eternal modules can then characterize sub-cultures or opinion groups like for instance extremists clusters. It is to be noted that in a bipartite graph represented by two colored sets of vertices, single colored eternal modules would represent a set of data-duplicates as no interaction would happen inside the module. Δ-modules, finally, combines both notions previously mentioned, characterizing sets of vertices who display the same behaviour toward the exterior for a period of time while differing in both their internal interactions and behaviour outside the considered Δ time window. The wide range of divergence allowed by this latest notion allows us to characterize populations who temporary are of a same group while not being polluted by their internal specifics. For instance, in a graph representing mail exchanges, if we consider the time instant of reception of a broadcast phishing mail, Δ-modules will characterize an identical response, regrouping fooled users on one side and users diligently reporting the scam to their mail providers on the other, regardless of differences that may intervene at other instants. Enumerating Δ-modules and eternal modules could therefore help in detecting behavioural patterns on a specific time period which would characterize populations of a same group.

For a practical use, the best performance of a streaming algorithm on a temporal graph would be a worst case time in $O(m + n)$ or in $O(\tau + n)$, where $n = |V(\mathcal{G})|$ and $m = \sum_{t \in T} m_t$ with m_t the number of edges in G_t. This is because n is usually small compared to m or to τ. The comparison between m and τ is more tricky. On the one hand, we can assume $m > \tau$ after removal of every time instant t where G_t is empty. However, when looking for Δ-modules where Δ does represent a real time window, say 51 seconds, this kind of preprocessing enforces us to do extra computations to preserve the correspondence between consecutive time instants in $\mathcal{G}$ and the actual real time lapse it represents in real life.

Using bucket sort, it is possible to (partly) solve EternalTwinsOfAGivenVertex with a streaming algorithm in worst case $O(\tau + n)$ time. More specifically, we can enumerate all the so-called eternal true-twins of a given vertex in that time, see the introductory part of [1]. Using a triangular exchange property of splitters (roughly, splitters are exterior vertices which give evidence for a pair not to be twins), we show in Property 1 in Section 4 a fast streaming process solving EternalTwinsOfAGivenVertex in worst case time $m + O(\delta_v \times n)$, see also [1, Property 1]. The downside of this process is in the space complexity, when a table of size $n^2 \times \tau$ must be used. Skipping the use of this table, the runtime of the streaming process becomes in $m + O(\delta_v \times n \times m)$ worst case time. However, we note that[2] it can be practically

[2] In our implementation, the inner search consists in determining whether an edge does not exist between some u and some w in graph G_t, given a time instant t. It is implemented in a hashmap called `mapEdgeByInstant` in function `computeEternalTwinsByEdgesIterationWithoutMatrices` defined in `src/TwinsAlgorithms.java`, *cf.* `https://github.com/DaemonFire/deltatwinsMEI/`. The lookup time is then practically constant instead of $O(m)$. Our full numerical analysis is

in $m + O(\delta_v \times n)$, when using a hasmap in order to speed up some inner loop of our search.

Our paper addresses the following question: *Would there be instantaneous response to* Δ-ModulesContainingAVertex *and* EternalModulesContainingAVertex *on historical graph data collected from human activities?* A subsequent concern would be to numerically confirm the question with the implementation of those algorithms. In reality we can only answer completely to the latter problem, and the special case of the former problem when Δ-modules are of size 2, that are, pairs of Δ-twins. We then use the results computed by those algorithms in order to give a preprocessing for Δ-Modules. Finally, we also try to determine whether the runtimes of our implementations remain reasonable when history length of the temporal graph is big. In particular, the foci in this paper are: long history (big τ), few vertices (small n) and good number of recorded edges (medium m). We revisit red-black tree data structure and devise a computation with runtime logarithmic in the history length of the input for Δ-Twins, independent of it for EternalModules, and linear in the history length for a pre-process which could lead to a solution to Δ-Modules. We then confront all implementations to one generated dataset and three datasets collected from real world data.

All algorithms presented in the work have been implemented in Java. The source code is available at `https://github.com/DaemonFire/deltaModules` along with the datasets used for the experiments and a file compiling all numerical results of those experiments, in order for all readers to be able to use this code as they see fit.

Contribution. For convenience, we address three more general problems, called EternalTwins, Δ-Modules and EternalModules, which are defined in the upcoming Section 2. Their solutions imply solutions for EternalTwinsOfAGivenVertex, Δ-ModulesContainingAVertex and EternalModulesContainingAVertex as a byproduct. We have examined partition refinement techniques and concluded that the list-based implementations such as in [9] or in [8, Lemma 10] would depend a lot on history length, adding to the overall computation a multiplicative factor in τ. We revisit matrix-based implementation of partition refinement and use red-black tree data structure in order to devise two variants of an algorithm for Δ-Twins. The two variants differ in the use of a large matrix in memory, the matrix-less version being the key to avoid *out of RAM* problems. Furthermore, the use of red-black trees allows our algorithm to compute even in the case where input graphs G_t for $t \in T$ are given unordered, mixing parts of one graph to another. This feature is fault tolerant for batched data which come asynchronously. All in all, the computation time is $O(m \times n \log \tau + N)$ with a $O(n^2 \times \tau)$ size adjacency matrix in memory and $O(m^2 \times n \log \tau + N)$ without it, where N represents the size of the output. Using the same core algorithm, we devise a solution

available at `https://github.com/DaemonFire/deltatwinsMEI/blob/master/results.csv` and plotted in [1] along with an extensive discussion.

to EternalModules with a computation time independent from τ. Using some properties on modules, we devise a pre-process for Δ-Modules which could help devise an algorithm solving this problem. This pre-process has a computation time in $O(n \times m^2 \times \log \tau + n^3 \times \tau)$.

Numerical experiments. We confront our implementations to generated data in order to confirm that

- The implementation for Δ-Twins is sound and its runtime is logarithmic in history length τ.
- The implementation for EternalModules has a runtime independent in history length τ.
- The implementation of the pre-process for Δ-Modules has a runtime in $O(n \times m^2 \times \log \tau + n^3 \times \tau)$.

We then confront it to real world datasets, with two collections from previous experiments [12,16] and a new one called `LesFurets`. The runtime of our algorithm for Δ-Twins averages at 12 seconds for all but one dataset, where it averages at 70 seconds. Our algorithm for EternalModules has a runtime of the same order but spacial complexity prove to be too big and *Out of RAM* issues happen on datasets on which we were able to compute Δ-Twins.

The paper is organised as follows. The formal framework and definitions involved in EternalTwins, Δ-Modules and EternalModules problems are defined in next Section 2. In Section 3, we define the red-black tree data structure used in our algorithms to achieve logarithmic runtime in the history length of the input data for Δ-Twins and a complexity in $O(n \times m^2 \times log(\tau) + n^3 \times \tau)$ for the pre-processing we designed for Δ-Modules.. In Section 4, we present our algorithms solving EternalTwins, Δ-Twins, EternalModules problems and the pre-processing we designed for Δ-Modules. All numerical analysis of our implementations are presented in Section 5, before we close the paper with concluding remarks and perspectives for further research.

2 Modules in a historical recording of graphs

Graphs in this paper are simple, undirected, and unweighted. A *temporal graph* is a sequence of graphs indexed by integers representing time instants. For practical use, it can also be formalized as a *link stream* $L = (T, V, E)$ such that $T \subseteq \mathbb{N}$ is an interval, V is a finite set, and $E \subseteq T \times \binom{V}{2}$. Usually, E is supposed to be a lexicographically ordered set, but in this paper, it is not necessarily the case. The elements of V are called *vertices* and the elements of E are called *(recorded) edges.* For $t \in T$, the subgraph G_t of L induced by t is a graph over the same vertex set V, with edge set $E_t = \{\{u, v\} : (t, \{u, v\}) \in E\}$. It is polynomial to transform link stream L into its sequence of subgraphs $(G_t)_{t \in T}$, and vice versa. In this paper, we indifferently refer to temporal graphs as link streams. The *adjacency matrix sequence* $(M_t)_{t \in T}$ is the sequence of adjacency matrices of graphs $(G_t)_{t \in T}$.

Temporal twin vertices have two variants. A pair of *eternal twins* $\{u, v\} \in \binom{V}{2}$ is a pair of vertices for which the neighbourhoods $N_t(u)$ and $N_t(v)$ are strictly equal in $V \setminus \{u, v\}$ for every instant $t \in T$. For an integer Δ, a pair of Δ-*twins* $\{u, v\} \in \binom{V}{2}$ is a pair of vertices for which the neighbourhoods $N_t(u)$ and $N_t(v)$ are strictly equal in $V \setminus \{u, v\}$ for Δ consecutive instants $t_0 \leq t \leq t_0 + \Delta$ with $[\![t_0, t_0 + \Delta[\![\subseteq T$.

Temporal modules also have two variants. An *eternal module* $A \subseteq V$ is a subset of vertices which have strictly equal neighbourhoods in $V \setminus A$ for every instant $t \in T$. For an integer Δ, a Δ-*module* $A \subseteq V$ is a subset of vertices which have strictly equal neighbourhoods in $V \setminus A$ for Δ consecutive instants $t_0 \leq t \leq t_0 + \Delta$ with $[\![t_0, t_0 + \Delta[\![\subseteq T$.

A vertex $u \in V$ is called a splitter of the pair of vertices $\{v, w\} \in \binom{V}{2}$ at time instant $t \in T$ if either

- $(u, v, t) \in E$ and $(u, w, t) \notin E$

or

- $(u, w, t) \in E$ and $(u, v, t) \notin E$.

Such a notion is important in our paper as if u is a splitter of (v, w) at the time instant t, (v, w) can't be a pair of twins at this instant. Indeed, if u is a splitter of (v, w) at instant t, $N(v)$, the neighbourhood of v, and $N(w)$ the neighbourhood of w are not identical at instant t. Likewise, if $u \notin A$ is a splitter of $(v, w) \in \binom{A}{2}$ at time instant t, A is not a module at this instant. Indeed if $u \notin A$ is a splitter of $(v, w) \in \binom{A}{2}$, $N(v) \setminus A$ is not identical to $N(w) \setminus A$ at instant t and then A is not a module at t.

Our paper addresses the following problems.

EternalTwins
Input : A link stream L.
Output : A list of all pairs of eternal twins in L.

Δ-Twins
Input : A link stream L and an integer Δ.
Output : A list of all pairs of Δ-twins in L.

EternalModules
Input : A link stream L.
Output : A list of all eternal modules in L.

Δ-Modules
Input : A link stream L and an integer Δ.
Output : A list of all Δ-modules in L.

3 Time tree data structure

All problems addressed in this paper boil down to the problem of finding splitters for pairs of vertices. We need to store all splitters for all pairs of vertices and the instants associated in an efficient way in terms of data consumption. Rather than storing lists of time instants and associated splitters, which would

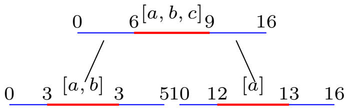

Fig. 5 Time-tree covering $[0, 16]$ where a is splitter for time instants $3, 6, 7, 8, 9, 12, 13$, b is splitter for time instants $3, 6, 7, 8, 9$ and c is splitter for time instants $6, 7, 8, 9$

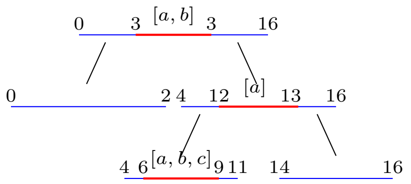

Fig. 6 This tree represents the same splitter list as the one in Fig. 5 but is one level deeper and therefore requires more operations and a greater computation time to add a new splitter.

amount to a worst case spacial complexity in $O(n \times \tau)$ where n is the number of vertices and τ the history length, for each pair of vertices (which would lead to a total worst case spacial complexity in $O(n^3 \times \tau)$), we designed a tree-based data structure inspired from red-black trees.

Each node covers a time range $P \subseteq T$ and contains a list of vertices $A \subseteq V$ and the time range $D \subseteq P$ of consecutive instants for which A is the exact list of splitters of the considered pair of vertices. The 2 sons of this node cover time ranges $Q \subseteq P$ and $R \subseteq P$ with $\forall t_1 \in Q$, $\forall t_2 \in D$, $\forall t_3 \in R$, $t_1 < t_2 < t_3$.

Each node is characterized by :

- First instant t_i of P. (For the root of Fig. 5, 0)
- Last instant t_f of P. (For the root of Fig. 5, 16)
- First instant of D. (For the root of Fig. 5, 6)
- Last instant of D. (For the root of Fig. 5, 9)
- List of vertices A. (For the root of Fig. 5, $[a, b, c]$)

for a worst case spacial complexity by node of n, where n is the number of vertices in V. Used for storing splitters of a given pair of vertices, one time-tree will have a number of node equal to the number of time ranges for which list of splitter is constant and not empty. In Fig. 6, we represented nodes that didn't contain a list of vertices but in terms of implementation, those can be left uninitialized and only initialized whenever data has to be stored in them. This means that at all time, a time-tree will have a number a of nodes such that $a \leq \tau$, where $\tau = \|T\|$ leading to a worst case spacial complexity in $O(n \times \tau)$. As we use a time-tree for each pair of vertices of the temporal graph, our total worst case space complexity will be in $O(n^3 \times \tau)$ but note that this worst case is unlikely as it would require the splitter list to change at each instant while being of a size close to the total size of the vertex set V. As we focus on sparse graphs, we are more likely to find large time span for which list of splitters of a given pair of vertices remains unchanged.

This type of tree can be balanced in order to minimize computation time of operations. But this balancing comes at a price. It requires computation of the depth of each sub-tree and to recursively balance each node.

Balancing operation of a node: if $depth(left\_son) - depth(right\_son) > 2$, then we can rotate left:
$t_f(father) \leftarrow t_i(right\_son) - 1$;
$t_i(right\_son) \leftarrow t_i(father)$;
$(right\_son(father)) \leftarrow left\_son(right\_son)$;
$left\_son(right\_son) \leftarrow father$;
$father \leftarrow right\_son$;

A rotation of a node loses no information. Indeed, both trees in Fig. 5 and Fig. 6 represent the same data, the only difference being the depth of the tree. This means that all sub-trees can be balanced, ensuring a depth in $log(p)$ where p is the number of time ranges for which the list of splitters is constant and not empty. That means that the depth of those trees would be inferior at all time to $log(\tau)$ where τ is the history length of the input link stream.

One rotation of a node is done in constant time, provided that the depth of the sub-tree rooted on each node is stored and updated accordingly.

Adding a splitter $u \in V$ for time instant $t \in T$ (see pseudo code in Alg. 1) :

- ***Node_initialization*** If this node contains an empty list of vertices A, $A \leftarrow [u]$, $D \leftarrow [t, t]$ and we create both sons of this node, which cover $\{a \in T, a < t\}$ and $\{a \in T, a > t\}$.
- ***Splitter_insertion*** If this node contains a list of vertices A and $D = [t, t]$, we add u to A.
- ***Node_split*** If this node contains a list of vertices A, $u \notin A$ and $t \in D$, we create a node N_1 and a node N_2 covering $T_1 \subset P$ (resp. $T_2 \subset P$), such that $\forall t_1 \in T_1, \forall t_2 \in D$, $t_1 < t_2$ (resp. $\forall t_1 \in T_2, \forall t_2 \in D, t_1 > t_2$). N_1 contains a list of vertices equal to A and a time range D_1 such that $\forall t_1 \in D_1, t_1 \in D \;\;\&\;\; t_1 < t$ (resp. $\forall t_1 \in D_2, t_1 \in D \;\;\&\;\; t_1 > t$). N_1's left son is the left son of the original node (resp. N_2's right son is the right son of the original node) and N_1 becomes the left son of the original node (resp. N_2 becomes the right son of the original node). Then $A \leftarrow [u]$ and $D \leftarrow [t, t]$.
- ***Node_expansion*** If this node contains a list of vertices A with $\|A\| = 1 \;\&\; u \in A$, and if $\exists t_0 \in D$, such that $t_0 - 1 = t$ (resp. $t_0 + 1 = t$), we add t to D.
- ***Recursion*** If $t \in Q$, we add splitter u at instant t in the left son of the node. Respectively, if $t \in R$, we add splitter u at instant t in the right son of the node.

After an addition of splitter in a node N, we use a consolidation operation that allows nodes to fuse together if their lists of splitter are the same and the times ranges concerned by those lists are adjacent *cf.* Algorithm 2. We dive to the rightest leaf of the sub-tree rooted on its left son and to the leftest leaf of the sub-tree rooted on its right son, which are the two nodes whose time ranges D are closest. If one of those nodes have the same list of vertices as N and its time range $D(leaf)$ is adjacent to $D(N)$, which means if $D(leaf) \cup D(N)$ is continuous, then $D(N) \leftarrow D(leaf) \cup D(N)$ and we destroy the leaf.

```
Data: A time tree $T$, and a splitter $u$ for the time instant $t$ to be listed in $T$
if $t \in T.dataRangeDeleted$ then
    if $u \notin T.splitters$ then
        Create node $newT$;
        Add all splitters from $T.splitters$ to $newT.splitters$;
        Add $u$ to $newT.splitters$;
        $T.dataRangeDeleted \leftarrow [t, t]$;
        Create node $newLeftSon$;
        Create node $newRightSon$;
        Add all splitters from $T.splitters$ to $newLeftSon.splitters$ and
          $newRightSon.splitters$;
        $newLeftSton.dataRangeDeleted \leftarrow [T.startInstant, t - 1]$;
        $newRightSon.dataRangeDeleted \leftarrow [t + 1, T.endInstant]$;
        $newLeftSon.leftSon \leftarrow T.leftSon$;
        $newRightSon.rightSon \leftarrow T.rightSon$;
        $newT.leftSon \leftarrow newLeftSon$;
        $newT.rightson \leftarrow newRightSon$;
    end
end
if $t \leq T.startInstant$ then
    if $T.leftSon$ exists then
        $T.leftSon.addSplitter(u, t)$;
    end
    $T.leftSon \leftarrow newLeftSon$;
    $newLeftSon.dataRangeDeleted \leftarrow [t, t]$;
    $newLeftSon.splitters.add(u)$;
end
if $t \geq T.endInstant$ then
    if $T.rightSon$ exists then
        $T.rightSon.addSplitter(u, t)$;
    end
    $T.rightSon \leftarrow newRightSon$;
    $newRightSon.dataRangeDeleted \leftarrow [t, t]$;
    $newRightSon.splitters.add(u)$;
end
```

Algorithm 1: Adding a splitter operation

```
Data: A time-tree $T$
Dive to the rightest son $D$ of $T.leftson$;
if $D.endInstant + 1 = T.startInstant$ $D.splitters = T.splitters$ then
    $T.startInstant \leftarrow D.startInstant$;
    Destroy $D$;
end
Dive to the leftest son $D$ of $T.rightson$;
if $D.startInstant - 1 = T.endInstant$ $D.splitters = T.splitters$ then
    $T.endInstant \leftarrow D.endInstant$;
    Destroy $D$;
end
```

Algorithm 2: Consolidation operation

Once the correct node is found, consolidation operation is done in constant time. The complexity comes from the dive which will at all time be inferior to $O(log(\tau))$.

When the correct node is found for addition of the splitter, the operation is done in constant time. If we balance our nodes at each addition of splitter, we only have to recursively balance nodes that have been modified by the operation, which means the node on which we finally added the splitter and his fathers. Balancing the tree after a splitter addition and consolidation of the tree then requires $O(log(d))$ operations, which means that adding a splitter in a balanced tree and re-balancing it afterwards comes with a total complexity in $O(log(\tau))$.

4 Matrix and matrix-less algorithms for enumerating temporal twins and modules

In many cases, an input link stream $L = (T, V, E)$ is not given by its adjacency matrix sequence $(M_t)_{t \in T}$, but rather by a list of its recorded edges E. The main idea of our algorithms is to iterate on this list of edges, allowing those algorithms to be used in real-time, as new edges are registered by the system. For each new edge in our link stream, we want to see if this edge introduces a new splitter for a pair of vertices.

Eternal-twins listing algorithm based on edges iteration (MEI and MLEI) Computing EternalTwins in time independent from history length τ can be done by using the triangular structure of splitters, as in Algorithm 3. If a pair $(u, v) \in V^2$ has a splitter at any given time instant t, $N_t(u) \neq N_t(v)$ with $N_t(u)$ the neighbourhood of u at instant t. Therefore, u and v are not eternal-twins. We start by initializing a matrix of size n^2 that will store a boolean (initialized at `true`) for each pair of vertices. Then, we iterate on the edge set and for each edge (u, v, t), we scan V for a w such that $(u, w, t) \notin E$ (resp. $(w, v, t) \notin E$) which would mean that w is a splitter of (u, v) at time instant t.

It is a standard exercise to prove the correctness of Algorithm 3, *e.g.* by an induction on $|E|$. The overall complexity of the algorithm is $O(m \times n + n^2)$ if $(M_t)_{t \in T}$ is also given as input, allowing us to test $(u, w, t) \notin E$ in constant time. This is the matrix version or Matrix Edge Iteration algorithm (MEI). But when $(M_t)_{t \in T}$ is part of the input, $O(n^2 \times \tau)$ space complexity at runtime is required, which usually causes *out of RAM* problems for big τ. Temporal complexity is in $O(m^2 \times n + n^2)$ otherwise, since scanning $(u, w, t) \notin E$ could take $O(m)$ especially if E is given unordered by $t \in T$. This is the matrix-less version or Matrix-less Edge Iteration algorithm (MLEI). We note that in practice the latter $O(m)$ factor is small, especially when E is chronologically ordered, by dichotomy search. Additionally, we create a HashMap storing all edges in sets indexed by their t. This way, we do not have to scan all E to find out if an edge $(t, \{u, w\}) \notin E$ but only the set of E_t, the edges at time instant t, reducing time complexity. We have proven the following property.

```
Data: Linkstream $L : (T, V, E)$
Result: List of all eternal-twins of $L$
for vertex $u$ do
    for vertex $v \neq u$ do
        Initialize $\texttt{Tw}(u, v) = \texttt{true}$;
    end
end
for recorded edge $(t, \{u, v\}) \in E$ do
    for vertex $w \in V \setminus \{u, v\}$ do
        if $(t, \{u, w\}) \notin E$ then
            $\texttt{Tw}(v, w) = \texttt{false}$;                // $u$ is a splitter of $\{v, w\}$
        end
    end
end
return every entry $u \neq v$ of table Tw where $\texttt{Tw}(u, v) = \texttt{true}$.
```

Algorithm 3: Edge Iteration algorithm for eternal-twins listing

Property 1 EternalTwins *can be solved in time independent from history length.*

Δ-twins listing algorithm based on edges iteration (MEI and MLEI). We now adapt the algorithm solving the eternal-twins problem to our Δ-twins listing problem, as in Algorithm 4. For this problem, it doesn't suffice to know that $(u, v) \in V^2$ have a splitter, we have to store in memory the time instant t at which the vertex w is a splitter. We use our time-tree data structure introduced in Section 3. But for the problem of Δ-twins, we don't care to store in our trees the vertices that are splitters of the pair but only the time instant t for which there exists at least one splitter. We can then discard the list of splitter A of each node of our trees and replace it by a boolean, `false` if the list of splitter is empty, `true` else, disminishing spacial complexity to a worst case $O(\tau)$ by tree. We will need $O(n^2)$ of them for a total worst case spacial complexity's upper bound in $O(n^2 \times \tau)$.

If $(M_t)_{t \in T}$ is given, the overall complexity of this Matrix Edge Iteration algorithm (MEI) is $O(m \times n \log \tau + N)$ with n the number of vertices, m the number of recorded edges, τ the history length and N the number of pairs of Δ-twins. If $(M_t)_{t \in T}$ is not given, complexity is $O(m^2 \times n \log \tau + N)$. This is the Matrix-less Edge Iteration algorithm (MLEI). The overall space complexity of algorithm MLEI is $O(n^2 \times \tau)$, due to the use of `Tw`. But it is really unlikely to reach this upper bound as it would require each pair of vertices to have at least one splitter each 2 time instants. Size of the data wasn't monitored during our experiments but the fact that the MLEI didn't encounter *out of RAM* issues where MEI did convinces us that at least on the focus of this paper (sparse graphs of few vertices on big time ranges), this space complexity remains reasonable. Furthermore, little optimizations can be done, like systematically turning the boolean to `true` for time ranges of size inferior to Δ between two time instants for which the boolean is `true` because the pair of vertices can't be a pair of Δ-twins on those time ranges, which, by collapsing node, will reduce the number of nodes and the size of the tree.

```
Data: A linkstream $L : (T, V, E)$ and an integer $\Delta$
Result: A list of all $\Delta$-twins of $L$
We initialize for each pair of vertices $\texttt{Tw}(u, v) = Tree(T)$;
Initialize a list $R$ of all entries in $\texttt{Tw}$;
for recorded edge $(t, \{u, v\})$ of $E$ do
    for vertex $w$ do
        if $(t, \{u, w\}) \notin E$ then
            Add the instant $t$ as an instant at which a splitter exists in $\texttt{Tw}(v, w)$;
            if an instant insertion exhausts the time instants in $\texttt{Tw}(v, w)$ then
                remove entry $(v, w)$ from $R$;
            end
        end
    end
end
for $(u, v)$ left in $R$ do
    scan all time ranges of at least $\Delta$ consecutive instants for which $A = \texttt{false}$ in
      $\texttt{Tw}(u, v)$ and add all ranges to output
end
return output
```

Algorithm 4: Edge Iteration Algorithm for Δ-twins listing

Theorem 1 *On input a link stream with n vertices, m recorded edges, τ history length, and N pairs of Δ-twins, Δ-*TWINS *can be solved in time $O(m \times n \log \tau + N)$ with $O(n^2 \times \tau)$ space complexity, or in time $O(m^2 \times n \log \tau + N)$ with a lesser space complexity.*

Eternal Modules listing using edge iteration We define the notion of minimal module containing a pair of vertices $\{u, v\}$.

Definition 1 In a static graph G, we refer to minimal module containing a pair of vertices $\{u, v\}$ of G as the module of G containing u and v of minimal size.

If $\{u, v\}$ is a pair of twins, then, the minimal module containing $\{u, v\}$ is $\{u, v\}$. If they are not twins, there exists a splitter w of $\{u, v\}$ in G. If A is a module of G containing u and v, then $w \in A$, otherwise A cannot be a module. We describe then this naive algorithm for computing the minimal module containing $\{u, v\}$

One iteration of the main loop of this algorithm has a worst case complexity in $O(n^3)$ where n is the number of vertices. If this worst case is reached, it's because $A = V$ and that's the last iteration of the loop as V is a module. We can then conclude that the worst case complexity of the whole algorithm is in $O(n^3)$. The following property can be seen as part of [10, Theorem 1]. However, since [10] only deals with modules of the so-called permutation graphs, we also give a short proof for our case. Two sets A and B overlap if: $A \setminus B \neq \emptyset$, $B \setminus A \neq \emptyset$, and $A \cap B \neq \emptyset$.

Property 2 ([10]) All modules in a graph G are either singletons or unions of overlapping minimal modules.

Proof Let A be a module of size at least 2 which is not a minimal module. For every distinct vertices $u \in A$ and $v \in A$, we claim that the minimal module

```
Data: Static graph $G(V, E)$, a pair of vertices $\{u, v\} \in V$
Result: The minimal module $A$ containing $\{u, v\}$ in $G$
$A \leftarrow \{u, v\}$;
while $A$ not a module do
    for vertex $x \in A$ do
        for vertex $y \in A$ with $x \neq y$ do
            if $splitters(u, v) \setminus A$ not empty then
                $A$ is still not a module;
                for vertex $w \in splitter(u, v) \setminus A$ do
                    Add $w$ to $A$ ;
                end
            end
        end
    end
end
return $A$.
```

of $\{u, v\}$ is a subset of A. Indeed, since A is a module containing u and v, the latter cannot be a super set of A, which would contradict its minimality. Now, if it overlaps A then, by intersection closure of overlapping modules, see *e.g.* [5,8,15], its intersection with A would contradict its minimality. Hence, every minimal modules of pairs of vertices of A is a subset of A.

If every pair of distinct vertices of A is a module then A is the union of overlapping minimal modules. We suppose there are pairs of distinct vertices in A whose minimal modules have size at least 3. Among the said pairs, we chose u and v be two distinct vertices whose minimal module, when combining with other minimal modules of A in an overlapping way, yields a union M of overlapping minimal modules with $k = |M|$ maximum. Since every minimal modules of vertices in A is a subset of A, we have that $M \subseteq A$. If $M = A$, then A is the union of overlapping minimal modules. Let's consider $M \neq A$ and $k < |A|$ and find a contradiction to the maximality of k.

There exists $w \in A \setminus M$. Let N be the minimal module of $\{u, w\}$. As explained above, $N \subsetneq A$. By maximality of k, $|N| \leq k$ and therefore we cannot have $N \supsetneq M$. Since $w \notin M$, $u \in N \cap M$ and we cannot have $N \supsetneq M$, the only way left for N and M is to overlap. But then $N \cup M$ contradicts the maximality of k. □

Using the property, we can compute all modules of a graph G recursively from a list of all minimal modules containing each pair of vertices of the graph, using this algorithm.

Complexity for this algorithm is in $O(M)$ where M is the number of modules in G. Now that we have defined those notions for static graphs, we adapt them to the EternalModules problem on linkstreams.

Definition 2 A minimal eternal-module A containing $\{u, v\}$ in L is the eternal-module of L containing u and v of minimal size.

Property 3 All eternal-modules of L are either singletons or unions of overlapping minimal eternal-modules.

```
Data: Static graph $G(V, E)$, a list of all minimal modules containing all pairs of
      vertices
Result: The list $MODULES$ of all modules of $G$
$MODULES \leftarrow$ list of all minimal modules containing all pairs of vertices;
while new modules are found in this iteration do
    for module $A \in MODULES$ do
        for module $B \in MODULES$ do
            if $A \cap B$ not empty then
                if $A \cup B \notin MODULES$ then
                    Add $A \cup B$ to $MODULES$;
                end
            end
        end
    end
end
return $MODULES$.
```

Proof The proof follows the same arguments as with Property 2, since eternal modules are very similar to modules. □

Both our algorithm for computing minimal modules for a given pair of vertices and our algorithm listing all modules of G when given the list of all minimal modules can be used as such to compute eternal modules as a subset $A \subseteq V$ is an eternal module if and only if there is no splitter outside of A for each pair of vertices $u \in A$ and $v \in A$ at no instant $t \in T$. What remains to define to be able to compute all eternal modules of a linkstream L is a means to compute all splitters for all pair of vertices of V. This can be accomplished thanks to our MEI, MLEI algorithms.

Listing all splitters for all pairs of vertices is accomplished by the Edge Iteration Algorithm in $O(n \times m^2)$ where n is the number of vertices and m the number of edges. From those lists of splitters, we compute minimal modules for all pair of vertices in $O(n^3)$ as previously established for the static version of the algorithm. The last step of our algorithm, the computation of all modules from those minimal modules has a complexity in $O(M)$ where M is the number of eternal-modules of L. This adds up to a complexity in $O(n^3 + n \times m^2 + M)$.

This complexity is independent in the history length τ of L, which is our goal, as the graphs we deal with present a large τ while n and m are smaller. Algorithm 5 sums up the above mentioned steps.

Pre-processing of Δ-modules

Now that we have presented methods of computation for Δ-TWINS and ETERNALMODULES problems, we need to tailor a solution for the Δ-MODULES problem combining those approaches.

Definition 3 A minimal instantaneous modules containing a pair of vertices $\{u, v\}$ for an instant t is the module in the static graph G_t representing the linkstream L at the instant t, containing u and v, of minimal size.

We can use the Edge Iteration Algorithm to compute splitters of each pair of vertices for each instant and use those to compute minimal instantaneous

```
Data: A linkstream $L : (T, V, E)$
Result: A list of all eternal-modules of $L$
We initialize for each pair of vertices $\texttt{Tw}(u, v) = \emptyset$;
for recorded edge $(t, \{u, v\})$ of $E$ do
    for vertex $w$ do
        if $(t, \{u, w\}) \notin E$ then
            Add vertex $u$ in $\texttt{Tw}(v, w)$;
        end
    end
end
We initialize a list output of eternal-modules containing all singletons constituted
  of all vertices of $V$;
for pair of vertices $\{u, v\} \in V$ do
    $MinimalModule\{u, v\} \leftarrow \{u, v\}$;
    while there is change happening do
        for pair of vertices $\{x, y\} \in MinimalModule\{u, v\}$ do
            if $\texttt{Tw}(x, y) \setminus MinimalModule\{u, v\} \neq \emptyset$ then
                Add all vertices of $\texttt{Tw}(x, y) \setminus MinimalModule\{u, v\}$ to
                  $MinimalModule\{u, v\}$;
            end
        end
    end
    Add $MinimalModule\{u, v\}$ to output;
end
while there is change happening do
    for pair of modules $(A, B) \in output^2$ do
        if $A \cap B \neq \emptyset$ then
            if $A \cup B \notin output$ then
                Add $A \cup B$ to output;
            end
        end
    end
end
return output
```

Algorithm 5: Edge Iteration Algorithm for Eternal-modules listing

modules for each pair of vertices, using the same method as for computation of minimal modules in static graphs.

In the interest of saving computation time, we compute those minimal instantaneous modules on time ranges. After computing all splitters for each pair of vertices for each instant, for each pair of vertices (u, v), for each node N of the time-tree listing their splitters, we create the minimal module containing (u, v) on time range $N.timerange$ and initiate it by adding (u, v) to $Module.vertexset$. Then, as long as there is change happening, we look up, pair of vertices (x, y) by pair of vertices in $Module.vertexset$. We access the time-tree representing splitters of (x, y) and check if $N.timerange$ is covered by a single node N_2 of this tree. If this is the case, we add all vertices of $N_2.splitters$ to $Module.vertexset$. Else, we need to divide the $Module$ we're currently building. For each node N_2 of the time-tree representing splitters of (x, y) for which $N_2.timerange \cap N.timerange \neq \emptyset$, we create $Module_2(u, v, N_2.timerange \cap N.timerange)$ with $Module_2.vertexset =$

$Module.vertexset \cup N_2.splitters$. And we iterate until nothing is changing for either of those modules.

```
Data: A linkstream $L : (T, V, E)$
Result: A list of all minimal instantenous modules for each pair of vertices of $L$
We initialize for each pair of vertices $\texttt{Tw}(u, v) = Tree(T)$;
for recorded edge $(t, \{u, v\})$ of $E$ do
    for vertex $w$ do
        if $(t, \{u, w\}) \notin E$ then
            Add splitter $u$ for instant $t$ in $\texttt{Tw}(v, w)$;
        end
    end
end
We initialize a list output of minimal instantaneous modules containing all
  singletons constituted of all vertices of $V$ on a time range equal to $T$;
for pair of vertices $\{u, v\} \in V$ do
    for TimeTreeNode $N \in$ Tw$(u, v)$ do
        Add $MinimalModule\{u, v, N.timeRange\}$ to output;
        $MinimalModule\{u, v, N.timeRange\} \leftarrow \{u, v\}$;
    end
    while there is change happening do
        for pair of vertices $\{x, y\} \in MinimalModule\{u, v, N.timeRange\}$ do
            Put in a list $NL$ all $N_2 \in \texttt{Tw}(x, y)$, $N_2.timerange \cap N.timerange \neq \emptyset$;
            if $\|NL\| = 1$ then
                $MinimalModule\{u, v, N.timeRange\}.add(NL[0].splitters)$;
            end
            else
                for $N_2 \in NL$ do
                    Add $MinimalModule\{u, v, N.timeRange \cap N_2.timeRange\}$ to
                      output ;
                    $MinimalModule\{u, v, N.timeRange \cap N_2.timeRange\} \leftarrow$
                      $MinimalModule\{u, v, N.timeRange\} \cup N_2.splitters$;
                end
                Remove $MinimalModule\{u, v, N.timeRange\}$ from output;
            end
        end
    end
end
return output
```

Algorithm 6: Edge Iteration Algorithm for pre-process of Δ-modules

As we showed that our splitter addition operation was done in $O(log(\tau))$, the splitter listing operation of this algorithm is done in $O(n \times m^2 \times log(\tau))$. From those splitters, the computation of all minimal instantaneous modules is done in $O(n^3 \times \tau)$ as we iterate on pair of vertices, and for each pair, we build a module of maximal size n, and at each addition of splitters, we search a tree, for a complexity in τ, the worst case number of nodes in one tree. We then reach an overall complexity for this pre-process in $O(n \times m^2 \times log(\tau) + n^3 \times \tau)$ with a spacial complexity in $O(n^3 \times \tau)$. While being far from ideal, this pre-process could allow us to compute Δ-modules by the following property, whose proof is straightforward.

Problem	Complexity of partition refinment	Complexity of MEI	Complexity of MLEI
EternalTwins	$O(n^3 \times \tau)$	$O(m \times n + n^2)$	$O(m^2 \times n + n^2)$
$\Delta - Twins$	$O(n^3 \times \tau \times \Delta)$	$O(m \times n \log \tau + N)$	$O(m^2 \times n \log \tau + N)$
EternalModules	$O(2^n \times n \times \tau)$	$O(n^3 + m \times n + M)$	$O(n^3 + m^2 \times n + M)$
$\Delta - Modules$	$O(2^n \times n \times \tau \times \Delta)$	$O(m \times n \log \tau + n^3 \times \tau)$	$O(m^2 \times n \log \tau + n^3 \times \tau)$

Table 1 Comparative table of complexities of algorithms solving our problems

Property 4 Given a time range of at least Δ time instants, for a subset A of vertices to be a Δ-module on this time range, all minimal instantaneous modules containing each pair of vertices $\{u, v\} \in A^2$ for all time instants of the time range must be subsets of A.

Thus, this is our belief that a method to compute all Δ-modules of L could be designed from this listing of minimal instantaneous modules. But to conceive this method would require a more thorough work on the subject.

Comparison with other techniques. In order to confirm the relevance of our contribution, we compared the complexity of our algorithm to other approaches already present in the literature. But we didn't manage to find algorithms treating those precise problems. So we used approaches used of the static versions of those problems and adapted them to our temporal problems.

The matrix based technique for *partition refinement*, used to solve the EternalTwins problem, is defined as follows. Assume initially that all pair of vertices are eternal twins: $\texttt{Tw}(u, v) = \texttt{true}$ for all $u \neq v$. For every pair of vertices $u \neq v$, time instant $t \in T$, and vertex $w \in V \setminus \{u, v\}$, if $M_t(w, u) \neq M_t(w, v)$, then $\texttt{Tw}(u, v) = \texttt{false}$. At the end of the process, output every entry $u \neq v$ of table Tw where $\texttt{Tw}(u, v) = \texttt{true}$. This results in a naive $O(n^3 \times \tau)$ solution for EternalTwins.

A similar process repeated Δ times using Δ different tables Tw allows to solve Δ-Twins in time $O(n^3 \times \tau \times \Delta)$.

To use such an algorithm to solve module problems, we would need to check for all subsets $A \subseteq V$ if we can find a pair of vertices $(u, v) \in A^2$ who has a splitter outside of A. There are 2^n subsets in the graph, with $|A|$ at worst case equal to n which would result, for EternalModule in a complexity in time $O(2^n \times n \times \tau)$ and for Δ-Module in a complexity in time $O(2^n \times n \times \tau \times \Delta)$, which are both unpractical.

5 Numerical Analysis

All the algorithms listed in the previous sections have been implemented in Java[3] and run on a standard laptop clocking at 2.7 Ghz. Since the use of EternalTwins is practically somewhat limited, plus the fact the algorithm is independent from history length, we only present numerical results for Δ-Twins and EternalModules. Experiments on the pre-process for Δ-Modules faced *Out of RAM* issues as spacial complexity was too big.

[3] Source code at `https://github.com/DaemonFire/deltaModules`.

This setback points the need to find a more economic method to compute Δ-module, the solution presented in this article being theoretically interesting but proving ineffective on the datasets we are working on.

Basically, for the correctness of the various implementations of twins computation, our methodology is unit-testing. The control groups are obtained from running an implementation of the naive partition refinement algorithm in $O(n^3 \times \tau \times \Delta)$ described in Section 4. Due to the high time complexity of the naive algorithm, we only test for instances where the naive computation do not exceed 45 minutes. This covers $\approx 33\%$ instances of all our experiments on twins in both below Section 5.2 and Section 5.3. Results are positive on all these samples. For modules problem, we could not assert correctness of the implementations as naive algorithms computing modules are too complex to be used on our datasets. We asserted that our algorithm solving EternalModules problem proved to be correct by comparing its result to the results of the algorithm solving EternalTwins. `LesFurets` dataset being a bi-partite graph, eternal-modules are either singletons, the entire vertex set V or are composed of vertices that are pairwise eternal-twins, as either vertices of the module are from different halves of the graph and for parts of their neighbourhood to be equal, the neighbourhoods must be empty, making them eternal-twins, or the module is contained in one of the half of the graph and then no edge can exist between vertices of the module, while they have the same neighbourhood in the other half, which means that all pairs of vertices of the module are pairs of eternal-twins. In our experiments, no eternal-twins existed, while our algorithm solving EternalModules found exactly $n+1$ eternal-modules, where n is the number of vertices, which is what the solution should be. In what follows we will totally skip the discussion about correctness, and focus only on computation time. We first present the datasets on which we are going to experiment in Section 5.1. Then we stress-test the implementation of our Δ-twins algorithm on big values of history length with a generated dataset, in Section 5.2. We continue our study by confronting this implementation to three different datasets collected from real world data, in Section 5.3. Then we finish by following the same steps on our eternal-module algorithm in Section 5.4 Our overall experiments run for more than 8000 (eight thousand) hours CPU time.

5.1 Datasets used in the study

`Timeprogression` : Our first dataset was generated in order to monitor history length's influence on computation time of the different algorithms while maintaining constant numbers of vertices and edges. Number of vertices was set to 50 and number of edges to 10^5, ascertaining that both dimensions are small enough so that history length's influence on algorithm's computation time would not be prone to be negligible. There are 199 instances, with history length varying from 5000 to 10^6 time instants. Those graphs are not ordered by time instants, allowing us to test that our algorithms run on batched data that comes asynchronously.

`Rollernet` dataset [16] has been collected from rollerbladers touring Paris. Links will be recorded at instant t whenever two rollerbladers are close enough during a given period. This is the denser of our linkstreams, ranging on a shorter history length, allowing us to test the limits of our algorithms which are designed for sparse graphs with great history length. This geometrical dataset is more likely to present a relatively greater number of Δ-twins and Δ-modules than the two other datasets as rollerbladers cruising together for a period of time will have similar neighbourhoods, it being based on their closeness to others. We run our experiments on the following seven batches of extracts, each batch containing 100 extracts. Two first batches contain link streams induced by $n_1 = 40$, resp. $n_2 = 50$, vertices of the raw Rollernet dataset. Three batches contain link streams induced by $m_1 = 10^5$, $m_2 = 2 \cdot 10^5$, and $m_3 = 3 \cdot 10^5$, recorded edges of the raw dataset. Two last batches contain link streams induced by $\tau_1 = 5000$, resp. $\tau_2 = 8000$, successive time instants of the raw dataset.

`Enron` dataset [12] is parsed from the log of e-mail exchanges between employees of a same company over a period of 3 years. Δ-modules would emerge from this link stream as people from the same service are sent the same e-mails for a certain period of time while having a uniform response to the outside (it being null or agreed upon inside the service). This link stream is very sparse and is the biggest of our study, with a huge history length and the greatest number of vertices and edges. We do the following seven batches of 100 extracts each, similarly as for `Rollernet`, with $n_1 = 50$, $n_2 = 100$, $m_1 = 5000$, $m_2 = 10000$, $m_3 = 20000$, $\tau_1 = 10^7$, and $\tau_2 = 5 \cdot 10^7$.

`LesFurets` dataset is parsed from the log of user behaviour on the lesfurets e-trade website's funnel, some vertices representing the various users and the others representing events on the funnel. This link stream is therefore a bipartite link stream. This dataset is not ordered by time instants. The latter feature also provides a way to test the robustness of our approach, in the sense of fault tolerance. In this context, Δ-modules would model groups of users interacting similarly with a portion of the funnel, having similar reaction times and realizing operations in the same order. We hope that with the best parameters, those Δ-modules would regroup peoples of similar behaviours, allowing the website to identify population that are not going to perform a purchase so that the system could take positive action to change this outcome. We do the following seven batches of 100 extracts each, similarly as the other datasets, with $n_1 = 300$, $n_2 = 600$, $m_1 = 3000$, $m_2 = 6000$, $m_3 = 8000$, $\tau_1 = 10000$, and $\tau_2 = 13000$.

Dataset	$\|\|V\|\|$	$\|\|E\|\|$	τ
Timeprogression (generated)	50	10^5	5000 to 10^6
Rollernet	40 to 62	10^5 to $4 \cdot 10^5$	5000 to 10^4
Enron	50 to 150	5000 to 25000	10^7 to 10^8
LesFurets	300 to 1000	3000 to 10000	10^5 to $1.5 \cdot 10^5$

Table 2 Datasets used in this study and their characteristics

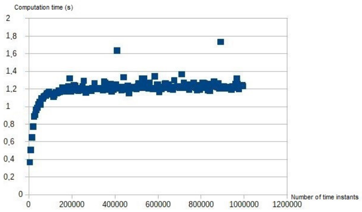

Fig. 7 Computation time of MLEI algorithm solving the Δ-Twins listing problem in function of history length on the `Timeprogression` datasets. We do not have consistent results for MEI due to *out of RAM*.

5.2 Logarithmic dependency in history length for Δ-twins's runtime

Theoretically, our algorithms compute twin vertices in time logarithmic in the history length of the input temporal graph. We would like to confirm this on runtime of their implementation.

Hypothesis 1. The computation time of MLEI algorithm solving Δ-twins problem is logarithmic in the history length of the input temporal graph.

Experimental results.
Results are presented in Figure 7.

Discussion: confirmation of Hypothesis 1. Progression of computing time is logarithmic, with few jumps (2 cases) probably due to some noisy use of the PC during computation.

5.3 Runtime on real world datasets

We confront our implementations to real world datasets, and would like to experiment both hypothesis below.

Hypothesis 2. Δ-twins can be enumerated in reasonable time.

Hypothesis 3. Algorithm MLEI is able to solve Δ-twins problem on link streams that cause exhaustion of memory for the MEI version.

Datasets and experiment result. Our methodology is to confront the implementations on three different datasets collected from real world data. We focus on Δ-Twins with $\Delta = 102$. In the sequel, we describe our sampling method over the three datasets. Then, a global view of all computation time is captured in Figure 8. We develop with detailed views on each of the three

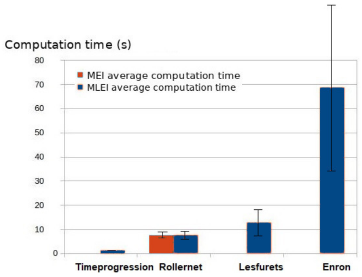

Fig. 8 Overview of computation time of all experiments.

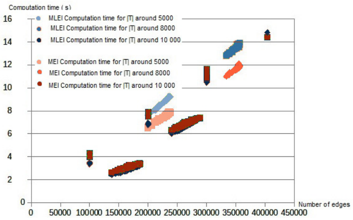

Fig. 9 Computation time of MLEI and MEI algorithms solving the Δ-twins listing problem in function of the number of edges in the link stream on the Rollernet datasets.

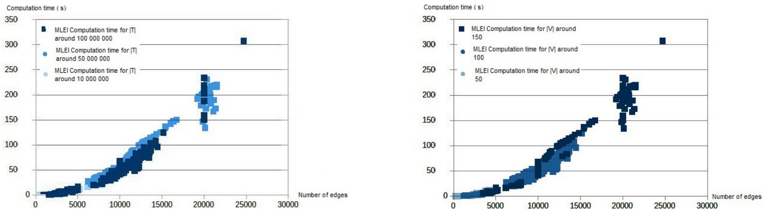

Fig. 10 Computation time of MLEI algorithm solving the Δ-twins listing problem in function of the number of edges in the link stream on the Enron datasets.

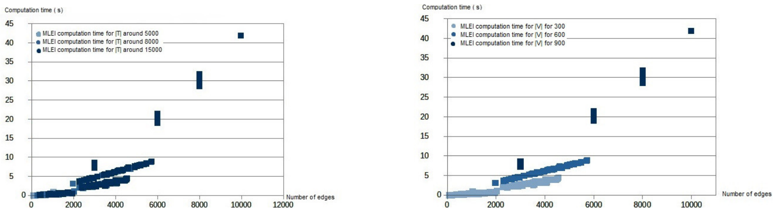

Fig. 11 Computation time of MLEI algorithm solving the Δ-twins listing problem in function of the number of edges in the link stream on the Lesfurets datasets.

datasets, in Figures 9, 10, and 11, respectively. We leave all discussions for the corresponding paragraph at the end of this section.

Discussion: slight confirmation of Hypothesis 2; confirmation of Hypothesis 3. Our experiments on the Rollernet datasets are where naive algorithm computes in reasonable time, allowing us to ascertain that MEI and MLEI algorithms compute Δ-twins correctly, *cf.* Figure 9. Naive algorithms computation time is not represented on those figures as it is far greater than computation time of both MEI and MLEI and it would have messed with the figure scale. It is also the only dataset we treated on which MEI does not encounter *out of RAM* issues, where we observe that MEI tends to be a bit more quicker to compute than MLEI. According to the heat-map, $|T|$ seems to have a minor impact on computation time as many datasets with large $|T|$ compute faster than datasets with small $|T|$. The overall tendency towards greater computation time being due to the increase of the number of vertices and of edges more than history length.

As soon as we reach Enron and LesFurets datasets, $|V|$ gets too big and naive algorithm computation time grows unreasonably. $|T|$ also grows to a large number and MEI, as expected, starts to face memory issues. Hypothesis 2 seems to be strained on Enron dataset. But we can still use those datasets to experiment on MLEI algorithm. For Enron, the left hand side of Figure 10 allows us to confirm our theoretical complexity regarding $|E|$ as our experimental curve of computation time in function of number of edges seems to describe a second degree polynomial function. The heat-map once again allows us to picture that history length's influence on complexity seems not to be so clear, indicating that the number of edges has a greater impact on computation time than history length of the link stream. On the other hand, the right hand side of Figure 10, where the heat-map correspond to number of vertices shows us what can be expected of a heat-map about an important complexity factor as darker points correspond to greater computation time than lighter ones. We proceed similarly for the results on LesFurets datasets, *cf.* Figure 11. They confirm the same tendencies as with Enron.

We conclude from our experiments that Hypothesis 2 is strained on Enron dataset, whereas Hypothesis 3 is confirmed. All in all, twin vertices can be computed within some minutes, even on Enron dataset.

5.4 Experimental study of Eternal-Modules algorithm

Theoretically, our algorithms compute eternal twins in time independent on history length of the input temporal graph. We would like to confirm this on runtime of their implementation. We use the same datasets as for Δ-twins and try to validate our hypothesis.

Hypothesis 4. The computation time of MLEI solving eternal-modules problem is independent in the history length of the input temporal graph.

For this, we use the `TimeProgression` dataset and analyse the results.

Our hypothesis seems to be a bit strained as computation time of MLEI on those datasets seems to describe an exponential curve, albeit highly cushioned for small history length. As numbers of vertices and edges are constant, the only factor in our theoretical complexity that could vary with history length

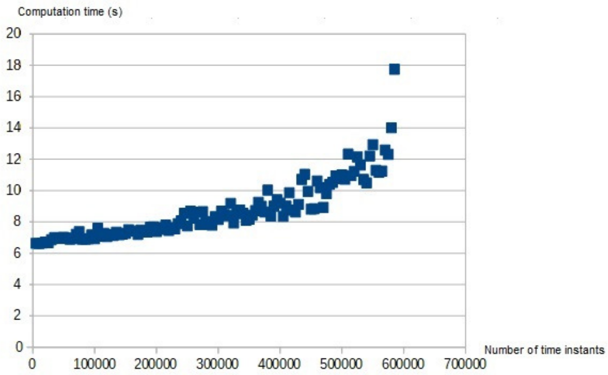

Fig. 12 Computation time of MLEI algorithm solving the EternalModules listing problem in function of history length on the `Timeprogression` datasets. We do not have consistent results for MEI due to *out of RAM*.

is the size of the solution, the number of eternal-modules in the input graph. But the number of eternal-modules is constant. It seems then, that some factor in history length has been disregarded in the theoretical analysis of the complexity of this algorithm. This curve also allows us to determine that spacial complexity seems to be an issue, with *Out of RAM* issues occurring as history length goes past 600 000 time instants. Once again, this issue of spacial complexity seems to point at a factor in history length for the spacial complexity of this algorithm.

Hypothesis 5. Eternal modules can be enumerated in reasonable time.

Hypothesis 6. Algorithm MLEI solving eternal-modules problem is able to compute link streams that cause exhaustion of memory for the MEI version.

Hypothesis 5 is again strained as our algorithm for listing eternal-modules present a complexity quadratic in the number of edges and cubic in the number of vertices. This was foreseen in our theoretical analysis but numerical experiments stress the fact that this high dependency remains a problem, even if our graphs are sparse with a small number of vertices. Furthermore, it is to be noted that as the number of vertices grows larger, number of eternal modules gets larger too, as each vertex is part of an eternal module. Therefore, as vertices increase, so does the amount of memory used to store the result, resulting in *out of RAM* issues due to the size of the solution itself. On `Enron` datasets, our algorithm runs out of memory as the number of vertices gets too big.

As for Hypothesis 6, this is partly confirmed by our experiments as MEI is able to compute only on Rollernet datasets. This is due, like for Δ-twins, to the use of adjacency matrices stored in memory. MLEI is able to compute on extracts of `LesFurets` and `TimeProgression` datasets on which MEI is unable to compute. But MLEI is way more limited on this problem than on

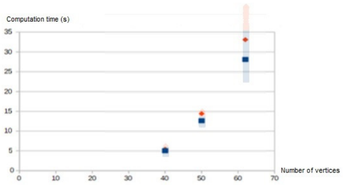

Fig. 13 Computation time of MLEI and MEI algorithms solving the Eternal-modules listing problem in function of the number of vertices in the link stream on the Rollernet datasets.

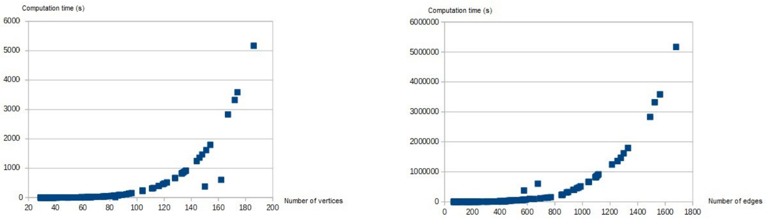

Fig. 14 Computation time of MLEI algorithm solving the Eternal-modules listing problem in function of the number of vertices on the left and of edges on the right on the Lesfurets datasets.

Δ-twins as we explained earlier that *Out of RAM* issues occured as history length or number of vertices reached a threshold.

Discussion: slight confirmation of Hypothesis 6; . Our experiments on Rollernet and LesFurets datasets allowed us to confirm that MLEI was able to compute datasets too large to be computed by MEI while MEI computed faster on the datasets it could compute on. But MLEI reaches its limits too as number of vertices and history length reach a threshold and begin experiencing *Out of RAM* issues due to its spacial complexity. Experiments run on `LesFurets` also proved the high dependency of MLEI on number of vertices and of edges, denying the Hypothesis 5 with a runtime growing too large. As for our Hypothesis 4, it was proven wrong by our experiments on `TimeProgression`, which displayed a dependency of the computation time on history length that seemed exponential, albeit highly cushioned below a threshold. Overall, while interesting theoretically and putting some grounds which could be used for further works, those algorithms proved not to be entirely convincing.

6 Conclusion and perspectives

We introduced two variants of modules in a historical collection of graphs and two specific cases when modules were twins. The corresponding algorithmic problems of enumerating all such modules are polynomial. We addressed the problem of solving them in time depending the least in the history length. Revisiting partition refinement techniques along with red-black tree data structures, we devise a logarithmic solution for Δ-twins and a solution for eternal-modules listing highly cushioned in function of history length. All algorithms are subject to two sub-variants: with or without the use of adjacency matrices in runtime memory. Confronting to datasets collected from real world data, our solutions for twins scales up to 10^8 history length while eternal-module listing scales up to 6×10^5. The pre-process we designed for Δ-modules does not scale on the datasets we used but proved theoretically interesting. While twins listing algorithms presented in this paper are functional for use on real world data, our algorithms on modules are interesting theoretically speaking but would require some tailoring to be able to compute on huge datasets. An interesting development of this work could be the replacement of all matrix data by hash-maps or sorted arrays. Then, extensive numerical analysis should be made in order to compare these three approaches (matrix, hash-maps and sorted arrays).

Acknowledgements We are grateful to Emmanuel Chailloux for helpful discussion and pointers. We are grateful to the anonymous reviewers of the conference version of this paper for their helpful comments which greatly improved the paper and to the reviewers of the present version who guided us in polishing structure and sense.

References

1. Bui-Xuan, B., Hourcade, H., Miachon, C.: Computing temporal twins in time logarithmic in history length. In: 9th International Conference on Complex Networks and their Applications, *SCI*, vol. 944, pp. 651–663 (2020)
2. Bui-Xuan, B.M., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. International Journal of Foundations of Computer Science **14**(2), 267–285 (2003)
3. Casteigts, A., Flocchini, P., Godard, E., Santoro, N., Yamashita, M.: Expressivity of time-varying graphs. In: 19th International Symposium on Fundamentals of Computation Theory, pp. 95–106 (2013)
4. Dibbelt, J., Pajor, T., Strasser, B., Wagner, D.: Connection scan algorithm. ACM Journal of Experimental Algorithmics **23** (2018)
5. Ehrenfeucht, A., Harju, T., Rozenberg, G.: The Theory of 2-Structures: A Framework for Decomposition and Transformation of Graphs. World Scientific (1999)
6. Erlebach, T., Hoffmann, M., Kammer, F.: On Temporal Graph Exploration. In: 42nd International Colloquium on Automata, Languages, and Programming, *LNCS*, vol. 9134, pp. 444–455 (2015)
7. Foschini, L., Hershberger, J., Suri, S.: On the complexity of time-dependent shortest paths. Algorithmica **68**(4), 1075–1097 (2014)
8. Habib, M., Paul, C.: A survey of the algorithmic aspects of modular decomposition. Computer Science Review **4**(1), 41–59 (2010)
9. Habib, M., Paul, C., Viennot, L.: Partition refinement techniques: an interesting algorithmic tool kit. International Journal of Foundations of Computer Science **10**(2), 147–170 (1999)
10. Heber, S., Mayr, R., Stoye, J.: Common intervals of multiple permutations. Algorithmica **60**, 175–206 (2011)
11. Kempe, D., Kleinberg, J., Kumar, A.: Connectivity and inference problems for temporal networks. Journal of Computer and System Sciences **64**(4), 820–842 (2002)
12. Klimt, B., Yang, Y.: Introducing the Enron Corpus. In: CEAS (2004)
13. Latapy, M., Viard, T., Magnien, C.: Stream graphs and link streams for the modeling of interactions over time. Social Network Analysis and Mining **8**(61) (2018)

14. Ros, F., Ruiz, P., Stojmenovic, I.: Acknowledgment-based broadcast protocol for reliable and efficient data dissemination in vehicular ad-hoc networks. IEEE Transactions on Mobile Computing **11**(1), 33–46 (2012)
15. Spinrad, J.: Efficient Graph Representations. Field Institute Monographs, vol. 19. American Mathematical Society (2003)
16. Tournoux, P.U., Leguay, J., Benbadis, F., Conan, V., De Amorim, M.D., Whitbeck, J.: The Accordion Phenomenon: Analysis, Characterization, and Impact on DTN routing. In: 28th IEEE Conference on Computer Communications (2009)
17. Tsalouchidou, I., Baeza-Yates, R., Bonchi, F., Liao, K., Sellis, T.: Temporal betweenness centrality in dynamic graphs. International Journal of Data Science and Analytics pp. 1–16 (2019)