A Study of Connectivity on Dynamic Graphs: Computing Persistent Connected Components

Mathilde Vernet, Yoann Pigne, Eric Sanlaville

To cite this version:

Mathilde Vernet, Yoann Pigne, Eric Sanlaville. A Study of Connectivity on Dynamic Graphs: Computing Persistent Connected Components. 4OR: A Quarterly Journal of Operations Research, 2022, 10.1007/s10288-022-00507-3 . hal-02473325v2

HAL Id: hal-02473325
https://hal.science/hal-02473325v2

Submitted on 12 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Study of Connectivity on Dynamic Graphs: Computing Persistent Connected Components

Mathilde Vernet Yoann Pigné Eric Sanlaville

Abstract

This work focuses on connectivity in a dynamic graph. An undirected graph is defined on a finite and discrete time interval. Edges can appear and disappear over time. The first objective of this work is to extend the notion of connected component to dynamic graphs in a new way. Persistent connected components are defined by their size, corresponding to the number of vertices, and their length, corresponding to the number of consecutive time steps they are present on. The second objective of this work is to develop an algorithm computing the largest, in terms of size and length, persistent connected components in a dynamic graph. PICCNIC algorithm (PersIstent Connected CompoNent InCremental Algorithm) is a polynomial time algorithm of minimal complexity. Another advantage of this algorithm is that it works online: knowing the evolution of the dynamic graph is not necessary to execute it. PICCNIC algorithm is implemented using the GraphStream library and experimented in order to carefully study the outcome of the algorithm according to different input graph types, as well as real data networks, to verify the theoretical complexity, and to confirm its feasibility for graphs of large size.

1 Introduction

In static graphs, connectivity is measured thanks to the computation of connected components. The problem of graph connectivity is relevant to many applications and contexts, such as communication networks, logistic networks or social networks. Furthermore, when a graph can be decomposed into several connected components, many problems can be decomposed too and solved separately on the different components. For instance, coloring problems, matching problems or vehicle routing problems can be decomposed.

In some cases, connectivity is also a necessary condition that needs to be checked before solving a problem. Flows, for example, cannot be computed if the source and sink do not belong to the same connected component.

Furthermore, time is an important issue that needs to be taken into account in many fields. Indeed, the interactions between entities are not necessarily static, and their nature might not be constant over time either. Static graphs do not allow the modeling of interactions which are evolving over time. The logical extension of graphs allowing this is then dynamic graphs. In a dynamic graph, vertices and edges can be present or absent depending on time. Every piece of information carried by vertices or edges can also be time-dependent.

The issue addressed in this paper is connectivity in a dynamic graph. Several questions are answered. First, what does connected component mean in a dynamic graph? And second, how can connectivity be measured in a dynamic context? We propose an extension of connected components in dynamic graphs, called Persistent Connected Components (PCC). This new definition takes into account the temporal dimension of the graph, space and time being considered simultaneously. PCCs are defined by their number of vertices, similarly to connected components in static graphs, but also by the number of consecutive time steps they are present on. We propose a polynomial time algorithm computing non-dominated PCCs in a dynamic graph and the associated Pareto front. This algorithm is studied together with experiments that show tractability even for large graphs on a long time horizon. The experiments were carried on with different graph types in order to study the impact of the graph structure on the results. There were also experiments made on large size real instances in order to verify the applicability of our algorithm on real data.

Section 2 presents the main concepts of dynamic graphs necessary for this work, and related works. Section 3 presents the persistent connected components (PCC). Section 4 introduces the algorithm designed to detect PCCs in a dynamic graph. The experimental study is presented in Section 5 and concluding remarks are given in Section 6.

2 Main concepts and state of the art

Dynamic graphs, also known in the literature as *dynamic networks*, *time varying graphs* (Casteigts et al., 2012), *evolving graphs* (Xuan et al., 2003), *temporal graphs* (Michail, 2016) or *temporal networks* (Holme, 2015), have been studied mostly in the past 20 years. Holme (2015) made an extended survey.

When we consider a graph and its evolution over time, we work on a dynamic graph. The dynamicity can be on vertices, edges or both. The presence of vertices or edges can be modified during the interval on which the graph is studied. All the information carried by vertices or edges can also be time-dependent (costs, capacities, storage, etc.).

Definition 1. *A study interval $\mathcal{T} = \{1 \dots T\}$ is a discrete set of T time steps. The end of this interval, noted T, is called the time horizon.*

Definition 2. *A t-graph, noted G_i, $i \in \mathcal{T}$, is a static graph corresponding to the dynamic graph G at a given time step i.*

Definition 3. *A dynamic graph G is simply noted $G = (G_i)_{i \in \mathcal{T}}$ and is defined on a study interval $\mathcal{T} = \{1 \dots T\}$. It is a succession of t-graphs $G_i = (V, E_i)$, $i \in \mathcal{T}$, such that all t-graphs are defined over the same vertex set.*

Note that these definitions, close to the literature, allow to isolate a vertex by removing its adjacent edges. In terms of connectivity, this is equivalent to removing this vertex.

A compact representation of a dynamic graph can be given. See for example Figure 1a where the labels on edges represent their times of presence. Figures 1b to 1e show the succession of static graphs.

Xuan et al. (2003) extend the definition of paths to dynamic graphs. The equivalent, in a dynamic graph, of a path in a static graph is a journey. A journey from a vertex u to a vertex v in a dynamic graph starts from u at time step i_{start} and ends on v at time step i_{end}. It is a succession of paths P_i in static graphs G_i. $P_{i_{start}}$ starts on vertex u in $G_{i_{start}}$. $P_{i_{end}}$ ends on vertex v in $G_{i_{end}}$. Path P_i in G_i, $i_{start} \leq i \leq i_{end}$, ending on a vertex w enforces path P_{i+1} to start on the same vertex w in G_{i+1}. A similar definition is used in the work of Kempe et al. (2002) in which the edges of the graph appear exactly once. In the following of the section, a focus is made on connectivity issues for dynamic graphs.

In static graphs, a connected component is a maximal set of vertices that are connected through edges in the graph. In other words, for two vertices u and v in the component, there exists a path between u and v in the graph. In directed graphs, the definition can be extended in two different ways, strongly and weakly connected components whether there exists a directed path from u to v and one from v to u or only one of those paths.

In dynamic undirected graphs, the existence of a journey from a vertex u to another vertex v does not imply the existence of a journey from v to u. Because of the edges time of presence, journeys are directed in dynamic graphs. In Figure 1, there is a journey from vertex 2 to vertex 3 going through edge $(2, 4)$ at time step 2 and edge $(4, 3)$ at time step 3. There is no journey from vertex 3 to vertex 2.

Based on the definition of journeys from Xuan et al. (2003), Bhadra and Ferreira (2003) give a definition of strongly connected components in a dynamic directed graph. Their definition can also be applied to undirected graphs. Such a component is a maximal set of vertices such that for all vertices u and v in the component, there exists a journey from u to v and a journey from v to u in the

graph. A distinction is made between closed strongly connected components and open strongly connected components. In the former, the journeys must cross vertices inside the component only whereas in the latter, journeys can cross vertices outside the component. In Figure 1, $\{1, 2, 4\}$ is an open strongly connected component. There exists a journey, both ways, between each pair of vertices. The journey from vertex 4 to vertex 1 goes through vertex 3 which is not in the component, because there exists no journey from vertex 3 to vertex 2. Bhadra and Ferreira also prove that the problem of finding a connected component (open or closed) of size k for a given value k is NP-Complete.

This definition implicates an interesting feature. Unlike static graphs, in dynamic graphs, connected components do not partition the vertices. Strongly connected components in dynamic graphs can overlap, as a vertex can be a part of two distinct components. In Figure 2, $\{1, 2, 3, 4\}$ is a closed connected component. There is a journey both ways between each pair of vertices of the component going only through vertices of the component. For the same reasons, $\{4, 5, 6, 7\}$ is also a closed strongly connected component. Vertex 4 is part of both components.

Jarry and Lotker (2004) use Bhadra and Ferreira's definition and show that asking whether a graph is connected or not is NP-hard even for two-layer grids but is polynomial in the case of trees. They propose an algorithm for this particular case.

Nicosia et al. (2012) work on connectivity on dynamic graphs using a definition corresponding to the open strongly connected components of Bhadra and Ferreira (2003). They propose a way to solve the problem of finding such components in a graph using a clique search in a static undirected graph, which is not polynomial.

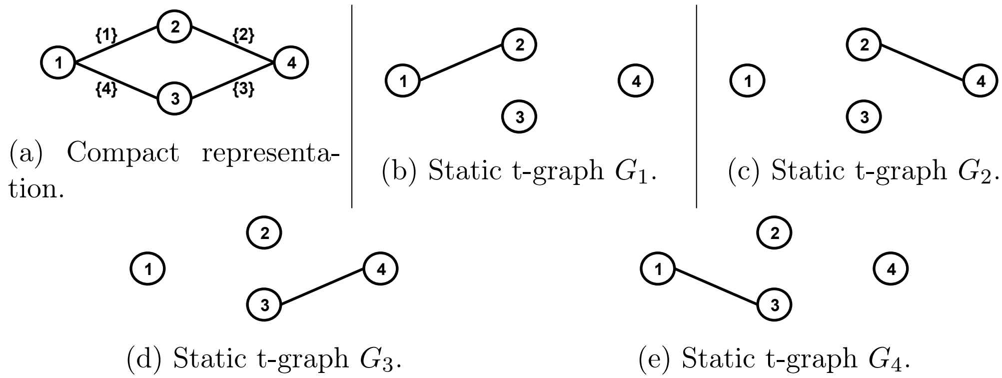
Gómez-Calzado et al. (2015) extend Bhadra and Ferreira's definition with Δ-component where the journeys connecting vertices in the component must be at most Δ time steps long.

Huyghues-Despointes et al. (2016) also propose an extension to Bhadra and Ferreira's definition. They define a δ-component (which they call Δ-component) to be a set of vertices which are open strongly connected on any time window of size δ of the dynamic graph. In their definition, the journeys connecting the vertices of the component can only cross one edge per time step.

All those definitions are based on journeys in the dynamic graph. Some vertices can be in the same component and never be connected at any time step of the graph. Vertices 1 and 4 from example in Figure 1 are never directly connected by an edge or a path in any t-graph.

The main usage of such definitions is message transmission.

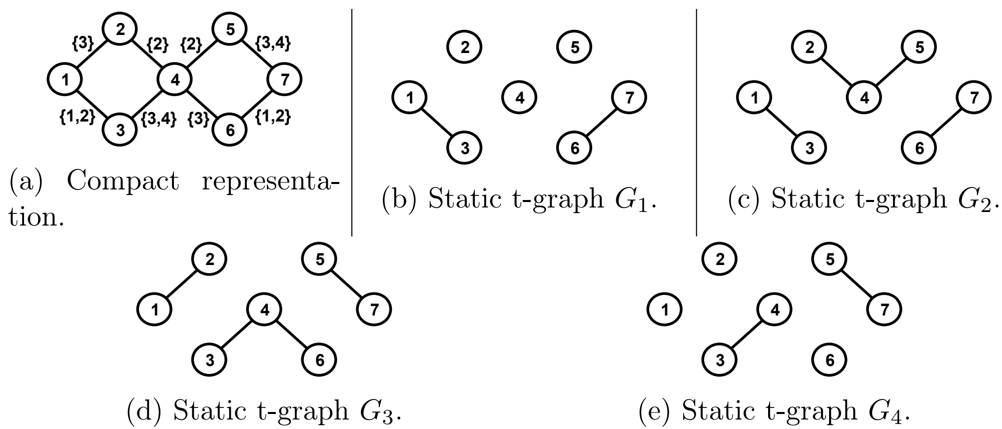
Casteigts et al. (2015) work on connectivity in dynamic graphs and define the τ-interval connectivity (which they call T-interval connectivity). A dynamic graph is τ-interval connected when the intersection of $G_i \ldots G_{i+\tau-1}$ for all $i \in [1, T-\tau+1]$,



(a) Compact representation. (b) Static t-graph G_1. (c) Static t-graph G_2.

(d) Static t-graph G_3. (e) Static t-graph G_4.

Figure 1: Dynamic graph on 4 time steps. Labels on edges of the compact representation are the time steps the edges are present on. Each t-graph is represented.



(a) Compact representation. (b) Static t-graph G_1. (c) Static t-graph G_2.

(d) Static t-graph G_3. (e) Static t-graph G_4.

Figure 2: Dynamic graph on 4 time steps. There are two closed connected components: $\{1, 2, 3, 4\}$ and $\{4, 5, 6, 7\}$.

where T is the time horizon, is a connected graph in the static sense. They propose algorithms needing $O(T)$ operations (binary intersection and connectivity test) to solve this problem. They do not propose a definition of connected component based on their definition of τ-interval connectivity.

Akrida and Spirakis (2019) present a continuous time model. They define interval temporal networks as graphs for which a set of intervals of availabilities is defined on each edge. An edge is present during the defined intervals. They propose a polynomial time algorithm able to give the longest time interval starting at a given time x and ending before a given time y on which the graph remains connected. Unlike the work of Casteigts et al. (2015), the connection is not necessarily made using the same edges. They present a second algorithm computing the sets of vertices of cardinality larger than a given bound that remain connected for the longest period of time starting at a given time x. This gives connected components that do not overlap, unlike other definitions found in the literature. For both algorithms presented, the choice of parameter x determines the outcome of the algorithm. If the graph starts to remain connected at a later time than x or if the graph has large components that start being connected later than x then the algorithms do not detect it.

3 Persistent connected components

This section defines the persistent connected components. We propose a point of view of connectivity in dynamic graphs, which is not based on journeys, unlike most of the definitions found in the literature. An example is given and applications are discussed.

3.1 Definitions and notations

Definition 4. *A persistent connected component (PCC) p of G is a set of k vertices in V that are connected in the graph (either directly or through other vertices of the graph) for l consecutive time steps. Vertices $u_1 \dots u_k$ form a persistent connected component of size k and length l if and only if there exist $G_i \dots G_{i+l-1}$ such that $u_1 \dots u_k$ are in the same connected component in the static sense in each G_j $(i \leq j < i + l)$.*
$p = (K, k, l, f)$ where K is the set of its vertices, k is the size of this set, l is the length of the component, in other words, the number of consecutive time steps the component is present on, and f is the last one of those time steps.

PCCs are characterized by their size (number of vertices) and length (number of consecutive time steps). Analogously to the static definition that implies maxi-

mality regarding to inclusion, we define maximality regarding to inclusion of both vertices and time steps.

Definition 5. *A maximal persistent connected component is a PCC which is maximal regarding its size and length. For a given maximal PCC $p = (K, k, l, f)$ with $K = \{u_1, \ldots, u_k\}$:*

$$\nexists\ p' = (K \cup \{u_{k+1}\}, k+1, l, f) \qquad \textit{and} \tag{1}$$
$$\nexists\ p'' = (K, k, l+1, f) \qquad \textit{and} \tag{2}$$
$$\nexists\ p''' = (K, k, l+1, f+1) \tag{3}$$

A maximal PCC is a PCC such that its vertex set is not included in a bigger vertex set connected on the same time steps (condition 1), and the same vertex set is not connected on the previous time step (condition 2) nor on the next time step (condition 3). In the following, we will only consider maximal PCCs, therefore by slight abuse of notation, the term PCC will be used to refer to a maximal PCC.

It should be noted that both directed and undirected graphs can be considered. In the case of undirected graphs, we consider, for PCCs, a set of k vertices simply connected. And in the case of directed graphs, we consider a set of k vertices strongly connected. All the definitions hold.

In static graphs, we can look for the largest connected component. Similarly, in dynamic graphs, we aim at finding the biggest persistent connected components in terms of size and length. Hence there are two criteria to optimize, that is the reason to look for a Pareto front formed by all non dominated PCCs.

Definition 6. *$p = (K, k, l, f)$ is a non-dominated PCC if and only if there exists no PCC $p' = (K', k', l', f') \neq p$,*

$$k' > k\ ;\ l' \geq l \qquad \textit{or} \tag{4}$$
$$l' > l\ ;\ k' \geq k \qquad \textit{or} \tag{5}$$
$$k' = k\ ;\ l' = l\ ;\ f' < f \qquad \textit{or} \tag{6}$$
$$k' = k\ ;\ l' = l\ ;\ f' = f\ ;\ K' \prec K \tag{7}$$

Condition 6 from definition 6 implies that in the case of two components of same size and same length, the earliest one is considered dominant. If furthermore two components have the same size and length and finish at the same time step, then an arbitrary total order on the vertex subsets shall be used (for instance, the lexicographic order), this is insured by condition 7.

One may question the extension of the chosen dominance, particularly considering conditions 6 and 7. The first motivation is the wish to compute some "large" components, that have a meaning considering the applications (e.g., a subset of

nodes considered as "safe" when connectivity is considered). Computing all such large components may not be useful, it may not be practical either. The second motivation is to provide the Pareto curve itself, considering size and length. Indeed, this curve provides a characterization of the network, as our experiments will show. The third motivation is practicality. Thanks to our definition, a set of at most $\min(n, T)$ PCCs is obtained. If conditions 6 and 7 are relaxed, the size of the obtained set might grow exponentially with n. A dynamic graph with this characteristic is easy to build, considering for instance that each G_t connects exactly one different vertex subset of size $\frac{n}{2}$. This entails some complexity issues, both in time and in space, to compute and to keep the set.

One may object that to get such a large number of PCCs, a number of time steps exponential in n is needed, which is true. Still, increasing the time horizon will mechanically increase the set size, which is not true with our definition. Furthermore, another dynamic graph can be built, for which the number of PCCs is $O(n^2)$ when $T = O(n)$. Suppose $n = 2 \cdot k$, and at each time step $t \leq k$, the graph G_t is composed of k disjoint edges. These edges are different at each time step. To achieve this, it suffices to consider as edges the node pairs (i, j), $i \in \{1 \dots k\}$, and j taken by circular permutation from the set of integers $\{k+i, k+i+1 \dots, k+i+k-1\}$. Hence, after k time steps, exactly $k \times k$ PCCs are obtained, all with size 2 and length 1. In this example, a number of PCCs quadratic in n is obtained in a number of time steps linear in n. Hence even for small values of T, the number of PCCs might increase rapidly.

These two examples show that our dominance definition allows to avoid an explosion on the number of solutions. The experiments presented later in the paper show that computing these solutions, with the appropriate algorithm, is possible for large values of n and T.

A PCC (as per Definitions 4 and 5) is a maximal set of k vertices that stay connected for at most l consecutive time steps. Therefore, in all of those time steps, for any two vertices in the PCC, there exists a path between them. A PCC is then always part of an open strongly connected component (as defined in Section 2). Our definition of connectivity, like those of Casteigts et al. (2015) and Akrida and Spirakis (2019), is not based on journeys. Nevertheless, it differs from their definitions. If a graph is τ-interval connected according to the definition in (Casteigts et al., 2015), then this graph is necessarily connected over $\mathcal{T}$ and it has a persistent connected component composed of the n vertices of the graph and lasting for the whole study interval $\mathcal{T}$. The reciprocal implication does not hold, because even if a graph remains connected over $\mathcal{T}$, as the connection might not be achieved with the same edges, then it will not necessarily be τ-interval connected for $\tau > 1$. In (Akrida and Spirakis, 2019), as they work with a continuous time model, the dynamic graph cannot be described as a succession of static graphs.

Unlike most problems described in Section 2, the problem of finding undominated PCCs can be solved polynomially (as proved in section 4.4).

3.2 Applications

Connectivity in a dynamic graph finds applications in many fields. Remember first that travel times associated to edges are not considered in this model. This is appropriate when the network's dynamics is slow compared to the time necessary to cross an edge.

In communication networks such as ad hoc networks or sensor networks, the transmission is almost instantaneous. In such networks, a PCC is a subnetwork that remains connected, which is essential when communications are considered, see for instance (Koster and Muñoz, 2009).

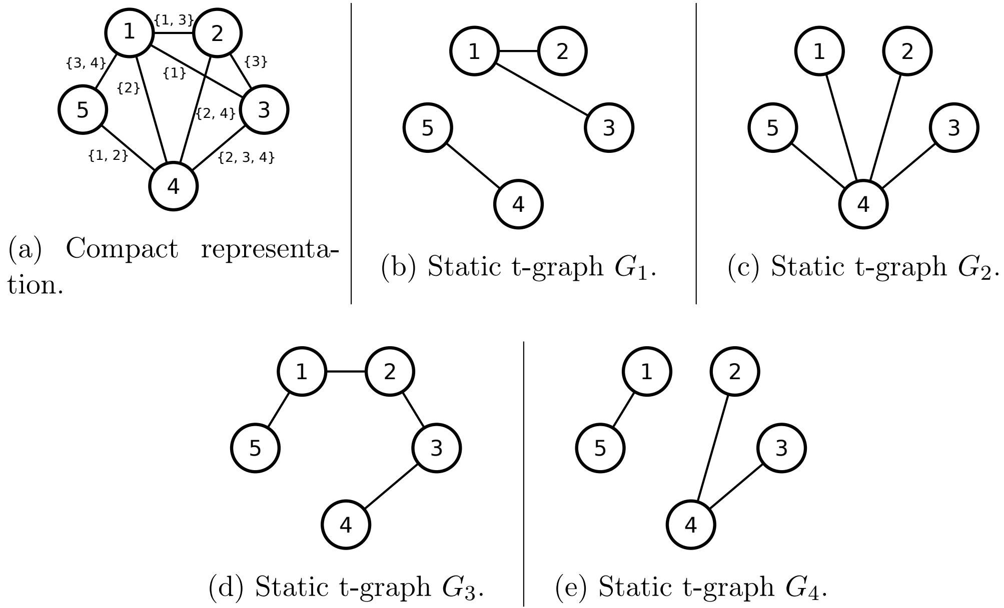
In transportation networks, roads availability can be time-dependent. The unavailability of a road can be temporary, new roads can be built and existing roads can be closed. Travel time on edges are often negligible compared to the networks dynamics. A PCC would measure in this case the reachability of different locations. Démare et al. (2017), for instance, use dynamic graphs to model the transportation network on the Seine valley.

In social networks, there is no travel time on edges because they represent a relationship, see the seminal work of Newman (2003). Most works on community detections use local edge density, some of them also consider time dimension (Nguyen et al., 2011). If we consider that a community must verify the connectivity condition between its members during some time interval, then detecting PCCs, according to our model, will help identify these communities.

3.3 Example

Figure 3 presents a dynamic graph on 4 time steps and 5 vertices. This graph is not connected through the whole study interval. The t-graphs G_1 and G_4 are disconnected and both have two connected components ($\{1, 2, 3\}$ and $\{4, 5\}$ in G_1 and $\{1, 5\}$ and $\{2, 3, 4\}$ in G_4). The t-graphs G_2 and G_3 only have one connected component containing all vertices. There are 6 maximal persistent connected components in this graph.

Vertices 1 ad 5 are connected from time step 2 to time step 4, even though they are directly connected with an edge only on time steps 3 and 4. They form a persistent connected component $p_1 = (\{1, 5\}, 2, 3, 4)$. Similarly, vertices 4 and 5 are connected from time step 1 to time step 3 and therefore they form a persistent connected component $p_2 = (\{4, 5\}, 2, 3, 3)$. In this graph, vertices 1, 2 and 3 are connected from time step 1 to time step 3. It can be noticed that in G_2, even though they are connected, they form an independent set. Those vertices form



(a) Compact representation.

(b) Static t-graph G_1.

(c) Static t-graph G_2.

(d) Static t-graph G_3.

(e) Static t-graph G_4.

Figure 3: Dynamic graph on 4 time steps.

a persistent connected component $p_3 = (\{1, 2, 3\}, 3, 3, 3)$. Vertices 2, 3 and 4 are connected from time step 2 to time step 4 and form a persistent connected component $p_4 = (\{2, 3, 4\}, 3, 3, 4)$. As the graph is connected from time step 2 to time step 3, vertices 1, 2, 3, 4 and 5 form a persistent connected component $p_5 = (\{1, 2, 3, 4, 5\}, 5, 2, 3)$. It can be noticed that vertices 2 and 3 stay connected over the whole study interval, even though they are directly connected only in G_3. Therefore vertices 2 and 3 form a persistent connected component $p_6 = (\{2, 3\}, 2, 4, 4)$. In the sense of Definition 4, $p_1, \ldots, p_6$ are all persistent connected components, and they are all maximal in the sense of Definition 5.

In the sense of Definition 6, components p_1 and p_2 are dominated by both p_3 and p_4 because of condition (4). Component p_4 is dominated by p_3 under condition (6). Components p_3, p_5 and p_6 are non-dominated. Those last components are the ones that we want to find.

Figure 4 shows a representation of the PCCs from the graph of Figure 3. Persistent connected component p_3 is represented in orange, p_5 is in blue and p_6 is in green. It is clear that PCCs are not disjoint because a vertex can belong to several PCCs.

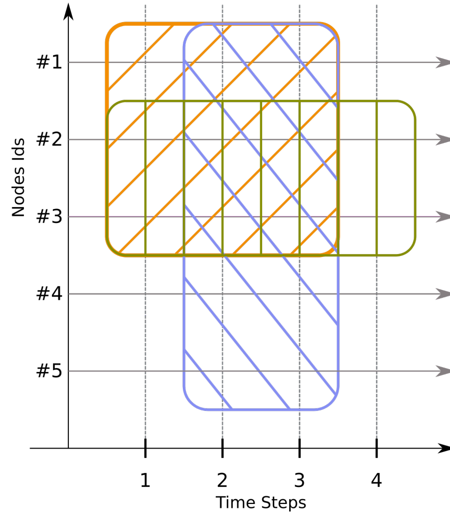


Figure 4: Visual representation of the persistent connected components of the graph from Figure 3. Each colored square represent a PCC: it covers the vertices composing it and all the time steps it is present on.

4 PICCNIC Algorithm

This section presents the PICCNIC Algorithm (PersIstent Connected CompoNent InCremental Algorithm) whose goal is to find the Pareto front containing all non dominated solutions, that is, every dominant PCC.

The algorithm will be presented together with as an execution example. Correctness and complexity of this algorithm will be proved.

4.1 Presentation

PICCNIC is presented in Algorithm 1. Its objective is, for a given dynamic graph, to find all dominant persistent connected components in the sense of Definition 6 that have size bigger than k_{min} and length bigger than l_{min}. The default value of k_{min} is 2 because a component of size 1 is not relevant as any vertex is connected to itself for the whole study interval of the graph. And the default value of l_{min} is 1 because we consider any set of connected vertices to be relevant. This algorithm works incrementally on the time steps and can therefore be used online.

We can access the vertex set of a PCC p with $K(p)$. The same way, we can access the size of a PCC p with $k(p)$, we can access it length with $l(p)$ and its finish date with $f(p)$.

Several sets of components are used to compute the persistent connected components. PCC_n contains, at each time step i, the components alive at i. To be built, it needs PCC_t which is a temporary set containing PCCS alive at time step

i. PCC_c keeps the components alive at the previous time step. PCC_o contains, at each time step, the components that just finished. PCC_f contains the non dominated persistent connected components.

Each iteration of the algorithm starts by retrieving all connected components of t-graph G_i (line 3). Those components are strongly connected components in the case of directed graphs, and simple connected components in the case of undirected graphs. We discard components of size lower than k_{min}. Default value is 2 because a vertex is necessarily connected to itself for the whole study interval, so each dynamic graph has n PCCs of size 1 and length T.

The first step of the algorithm (given in Algorithm 2) aims at finding the new persistent connected components beginning at i and keeping the components that are still going on at i. It uses the function *AddPCC* to add a PCC to a PCC set by checking that the PCC set does not contain another PCC with the same vertex. If it does, the PCC with the highest length is kept in the PCC set.

The second step of the algorithm (given in Algorithm 3) works on the persistent components that are over at the current time step (meaning that their finish date date is the previous time step) and keeps the dominant components. It uses Algorithms 4 and 5 to check if one component p_1 dominates another component p_2 depending on which one finishes first. Those domination algorithms check the conditions given in definition 6. The order on the vertex subsets of 2^V can be for example a lexicographical order.

4.2 Example

Let us look at the execution of PICCNIC Algorithm on example from Figure 3. We consider persistent connected component of size at least 2 and length at least 1 ($k_{min} = 2$, $l_{min} = 1$).

In the first iteration, we focus on G_1 (see Figure 3b). It has two connected components: $\{1, 2, 3\}$ and $\{4, 5\}$. We initialize the set of current PCCs $PCC_c = \{(\{1, 2, 3\}, 3, 1, 1), (\{4, 5\}, 2, 1, 1)\}$ and the set of final PCCs $PCC_f = \emptyset$.

In the second iteration, the current t-graph is G_2 (see Figure 3c). It has only one connected component $\{1, 2, 3, 4, 5\}$. At the end of the iteration, the set of current PCCs $PCC_c = \{(\{1, 2, 3\}, 3, 2, 2), (\{4, 5\}, 2, 2, 2), (\{1, 2, 3, 4, 5\}, 5, 1, 2)\}$. The first and second components were already present at the previous time step, therefore their length is now 2 and the last component appeared at this time step. The set $PCC_f = \emptyset$, because no component is finished yet.

In the third iteration, the current t-graph is G_3 (see Figure 3d). Just like the previous time step, it also has only one component. All components present before grow older. At the end of the iteration, $PCC_c = \{(\{1, 2, 3\}, 3, 3, 3), (\{4, 5\}, 2, 3, 3), (\{1, 2, 3, 4, 5\}, 5, 2,$ As all the component present before are still there, no component has finished and PCC_f is still empty.

Algorithm 1 *$PICCNIC$* Algorithm

Input: Dynamic Graph G, study interval $\mathcal{T} = \{1, \ldots, T\}$, lower bound on the size of PCCs k_{min}, lower bound on the length of PCCs l_{min}
Output: Dominant persistent connected components
// Loop on the number of instants
1: **for all** $i \in \{1, \ldots, T+1\}$ **do**
2: $G' = G_i$ *// Get the graph for the ith time step, void if $i = T+1$*
3: CC = set of connected components of G' of size $\geq k_{min}$
4: **if** i=1 **then**
5: $PCC_c = \{p = (K, k, l, f) | K \in CC, k = |K|, l = 1, f = 1\}$ *// for the next iteration*
6: $PCC_f = \emptyset$
// All CCs are current PCCs
7: **else**
8: $PCC_n = PICCNIC_{Step1}(CC, PCC_c, k_{min})$
9: $PCC_f = PICCNIC_{Step2}(PCC_n, PCC_c, PCC_f, l_{min})$
10: $PCC_c = PCC_n$ *// for the next iteration*
11: **end if**
12: **end for**
return PCC_f

Algorithm 2 $PICCNIC_{Step1}$

Input: CC, PCC_c, k_{min}
Output: PCC_n, the set of possible new PCCs

```
1: PCC_n = ∅
2: for all c ∈ CC AND |c| ≥ k_min do
3:     Pers = FALSE // Whether c is already a PCC or not
4:     PCC_t = ∅ // Contain PCCs for which the vertex set is in c
5:     for all p ∈ PCC_c do
6:         // Compare each static connected component c
7:         // with the set of vertices K(p) from each persistent connected component p
8:         if K(p) = c then
9:             Pers = TRUE
10:            K(p') = K(p)
11:            k(p') = |K(p')|
12:        else
13:            K(p') = K(p) ∩ c
14:            k(p') = |K(p')|
15:        end if
16:        if k(p') ≥ k_min then
17:            // Keep only the PCCs large enough
18:            l(p') = l(p) + 1
19:            f(p') = i
20:            AddPCC(PCC_t; p')
21:        end if
22:    end for
23:    if ¬Pers then
24:        // If the vertex set c is not the vertex set of an existing PCC
25:        p'' = {c, |c|, 1, i}
26:        AddPCC(PCC_t; p'') // Include new components that are not already in PCC_t
27:    end if
28:    PCC_n = PCC_n ∪ PCC_t
29: end for
30: return PCC_n
```

Algorithm 3 $PICCNIC_{Step2}$

Input: PCC_n, PCC_c, PCC_f, l_{min}
Output: Updated PCC_f

1: $PCC_o = PCC_c \setminus PCC_n$ *//Set of now finished PCCs*
2: **for all** $p \in PCC_o$ **do**
3:     **if** $l(p) < l_{min}$ **then**
4:         *// Keep only the PCCs lasting enough time*
5:         $PCC_o = PCC_o - \{p\}$
6:         STOPLOOP
7:     **end if**
8:     **for all** $p' \in PCC_o$ AND $p \neq p'$ **do**
9:         *// Compare PCCs that just ended to keep only the dominating ones*
10:         **if** $DomLaterEqual(p', p)$ **then**
11:             $PCC_o = PCC_o - \{p\}$
12:             STOPLOOP
13:         **else**
14:             **if** $DomLaterEqual(p, p')$ **then**
15:                 $PCC_o = PCC_o - \{p'\}$
16:             **end if**
17:         **end if**
18:     **end for**
19: **end for**
20: **for all** $p \in PCC_o$ **do**
21:     **for all** $p' \in PCC_f$ **do**
22:         *// Compare PCCs that just ended with non-dominated PCCs in* PCC_f
23:         *// to keep only the dominating ones*
24:         **if** $DomEarlier(p', p)$ **then**
25:             $PCC_o = PCC_o - \{p\}$
26:             STOPLOOP *// p dominated by some finished PCC*
27:         **else**
28:             **if** $DomLaterEqual(p, p')$ **then**
29:                 $PCC_f = PCC_f - \{p'\}$
30:             **end if**
31:         **end if**
32:     **end for**
33: **end for**
34: $PCC_f = PCC_f \cup PCC_o$
35: **return** PCC_f

Algorithm 4 *DomEarlier*

Input: Two PCCs p_1 and p_2 such that $f(p_1) < f(p_2)$
Output: TRUE if p_1 dominates p_2

1: **if** $(k(p_1) \geq k(p_2))$ AND $(l(p_1) \geq l(p_2))$ **then**
2: **return** TRUE
3: **end if**
4: **return** FALSE

Algorithm 5 *DomLaterEqual*

Input: Two PCCs p_1 and p_2 such that $f(p_1) \geq f(p_2)$
Output: TRUE if p_1 dominates p_2

1: **if** $(k(p_1) > k(p_2))$ AND $(l(p_1) \geq l(p_2))$ **then**
2: **return** TRUE
3: **end if**
4: **if** $(k(p_1) \geq k(p_2))$ AND $(l(p_1) > l(p_2))$ **then**
5: **return** TRUE
6: **end if**
7: **if** $(k(p_1) = k(p_2))$ AND $(l(p_1) = l(p_2))$ AND $(f(p_1) = f(p_2))$ AND $(K(p_1) \prec K(p_2))$ **then**
8: **return** TRUE
9: **end if**
10: **return** FALSE

In the fourth iteration, the current t-graph is G_4 (see Figure 3e). It has two connected components: $\{1,5\}$ and $\{2,3,4\}$. In the first step, we consider first the static component $c = \{1,5\}$. We add component $(\{1,5\},2,3,4)$ to PCC_t by intersecting $\{1,5\}$ and the PCC $(\{1,2,3,4,5\},5,2,3)\}$. Component c with length 1 and finish date 4 is not added to PCC_t with function *AddPCC* because PCC_t already contains a PCC with the same vertex set and higher length. All PCCs from PCC_t are added to PCC_n. Then we consider the second static component $c = \{2,3,4\}$. We add component $(\{2,3\},2,4,4)$ to PCC_t by intersecting c and $(\{1,2,3\},3,3,3)$. We add component $(\{2,3,4\},3,3,4)$ to PCC_t by intersecting c and $(\{1,2,3,4,5\},5,2,3)$. Th component c with length 1 and finish date 4 is not added to PCC_t with function *AddPCC* because PCC_t already contains a PCC with the same vertex set and higher length. All PCCs from PCC_t are added to PCC_n. At the end of step 1, $PCC_n = \{(\{1,5\},2,3,4),(\{2,3\},2,4,4),(\{2,3,4\},3,3,4)\}$. At the beginning of step 2, $PCC_o = \{(\{1,2,3\},3,3,3),\ (\{4,5\},2,3,3),\ (\{1,2,3,4,5\},5,2,3)\}$. The second PCC is dominated by the first one so it is deleted. At the end of the fourth iteration, $PCC_f = \{(\{1,2,3\},3,3,3),(\{1,2,3,4,5\},5,2,3)\}$ and $PCC_c = \{(\{1,5\},2,3,4),(\{2,3\},2,4,4),(\{2,3,4\},3,3,4)\}$.

In the fifth iteration, there is no current t-graph as $T = 4$. The second phase of Algorithm 1, detailed in Algorithm 3, is executed. At the beginning of step 2, $PCC_o = \{(\{1,5\},2,3,4),(\{2,3\},2,4,4),(\{2,3,4\},3,3,4)\}$. The first component is dominated by the second one, and the third component is dominated by the component $(\{1,2,3\},3,3,3)$ present in PCC_f, so only the second component is added to PCC_f. At the end of this iteration, $PCC_f = \{(\{1,2,3\},3,3,3),$ $(\{1,2,3,4,5\},5,2,3),\ (\{2,3\},2,4,4)\}$. The last component has length 4, indeed, vertices 2 and 3 are connected over the whole study interval.

On graph from Figure 3, there are 3 non-dominated components: one of size 3 and length 3, one of size 5 and length 2 and one of size 2 and length 4.

4.3 Correctness

Theorem 1. *PICCNIC provides the set of all dominant persistent connected components, in the sense of definition 6, of a dynamic graph G on interval $\mathcal{T}$.*

Proof. To prove that the algorithm is correct, we have to show that:

1. At any time $i \leq T$, any dominant PCC (in the sense of definition 6) finishing before or at i is present in $PCC_f \cup PCC_c$.

2. At the end of the algorithm, only dominant PCCs are present in PCC_f.

Let us prove the first part. At $i = 1$, the result is trivial, as the set PCC_c contains all connected components of G at the first instant.

Let us suppose the result is true for some $1 \leq i \leq T$, and consider iteration $i+1$. Let $p = (K, k, l, f)$ be a dominant PCC at $i+1$. Suppose first it is not dominant for any $\theta \leq i$, hence its finishing time $f = i+1$. If its length is 1, it is necessarily a connected component of G for instant number $i+1$. In the algorithm, its associated boolean $Pers$ is false and p is directly included in PCC_t, then in PCC_n, then in PCC_c. Its is the only PCC with set K therefore it is not removed by *SuppressDouble*, and it is put in PCC_c. If its length is larger than one, let us first suppose that $p \in PCC_c$ at the beginning of the iteration. p must be included into a connected component of G at $i+1$. Therefore it is put in PCC_t and function *AddPCC* makes sure that the oldest PCC is kept in PCC_t. It is then added to PCC_n and then to PCC_c. Even if p finishes at $i+1$, it is not removed because of the dominance tests as it is by hypothesis dominant. If p does not belong to PCC_c at the beginning of the iteration, it means at all previous iterations it was not kept by the algorithm (as a component is never removed from PCC_c until it is finished), although it was included into one connected component of each of the associated graphs. This implies that at all of these instants, p was strictly included (in terms of vertex sets) into another PCC, say q, present in PCC_c at least at i. If q does not exist, p would have been added to PCC_c before as it would have appeared as intersection of a member of PCC_c with a connected component. The length of q is equal to the length of p before $i+1$ (so that p is not dominated by q at $i+1$). PCC q is not included into a connected component at $i+1$ as p is then dominant, and the intersection of q and some connected component c at $i+1$ is p (a larger intersection implies a larger subset of q present at $i+1$, but p is dominant).

Therefore p is included into PCC_t, then PCC_n and into PCC_c during iteration $i+1$. If p is dominant for $\theta \leq i$, then by induction hypothesis, p is present at time i. Either it is finished at $i+1$ or earlier and p remains in PCC_f (it is not removed by the algorithm as it is dominant), or it is not and it stays into PCC_c: it is included in one connected component of G. By induction, the result is true also for $i = T$. After the final step, only dominated PCCs are removed, therefore p is inside the list provided by the algorithm.

Let us now prove the second part. Suppose there exists some PCC $p = (K, k, l, f)$ that is present before the final step and which is dominated by some PCC $p' = (K', k', l', f')$. We just proved that p' is kept by the algorithm at T. If $p \in PCC_c$, during the final step it is placed in PCC_o as set CC is void. It is then tested against all other elements of PCC_c, then against all elements of PCC_f. Therefore it must be tested against p' and removed. If $p \in PCC_f$ and $p \in PCC_c$ at T, then for the same reason p' will be tested against p, and p will be eliminated. Suppose now $p, p' \in PCC_f$. They have been tested together when the latest PCC has finished (if they finished simultaneously, they are tested together by the algorithm while in PCC_o). Therefore it is impossible that p and p' are both

present. □ □

4.4 Complexity

In this section, we prove that Algorithm 1 is polynomial.

Lemma 1. *At the end of any iteration $1 \leq i \leq T$, the cardinality of PCC_c is bounded by $n - NbCC(G_i)$, where $NbCC(G_i)$ is the number of connected components of the graph G at iteration i.*

Proof. PCC_t is built for each static connected component of G_i. It does not contain doubles, meaning that it does not contain two PCCs that have the same vertex set. the function *AddPCC* makes sure of that. All the elements from PCC_t are then added to PCC_n. The intersection between a PCC from PCC_c and a static connected component can produce the same vertex set only when we consider the same static component. When we switch the current static component c, the intersections will necessarily give different vertex set. Therefore, the new PCCs added to PCC_t and then to PCC_n do not contain the same vertices as PCCs present in PCC_n. Therefore, there are no doubles in PCC_n. s PCC_n is built from PCC_n a tthe end of each iteration, it does not contain doubles either.

Let $p = (K, k, l, i)$ and $p' = (K', k', l', i')$ be two elements of PCC_n at the end of iteration i such that $K \neq K'$. Let p start before p'. K and K' have at least cardinality k_{min}. Both are necessarily included into one connected component of G_i.

If they are included into two different components, K and K' are disjoint.

Suppose now they are included into the same component c of G_i and not disjoint ($K \cap K' \neq \emptyset$). If one is not included into the other, $K'' = K' \cup K$ is also included into c. Furthermore, a PCC p'' associated to K'' is present since the iteration θ where p and p' were first both present. As p starts before p', it means p'' appears simultaneously with p'. But this is impossible since K'' strictly contains K and K'. Therefore p' does not appear. Consequently $K \subset K'$ (if $K' \subset K$ p' cannot be included into PCC_n). And we can say that the vertex sets of the PCCs present in PCC_c are either strictly included or disjoint.

It is easy to verify by induction that the number of such subsets is bounded by the cardinality of the set minus 1, regardless of the way those subsets are chosen. Indeed, a subset contains at least 2 elements (because $k_{min} \geq 2$) and the subsets are either disjoint or strictly included. It is obvious that if the set has 2 elements, at most one subset is acceptable according to the previous conditions. Now consider a set Ω having ω elements. Suppose it has $\omega - 1$ subsets, such that this number is maximal (it is impossible to create another subset without violating the strictly included or disjoint condition of the subsets). When one element e is added to Ω, we can either keep the same subsets, and in this case there are still $\omega - 1$ subsets,

or we can create a new subset by making an union between $\{e\}$ and an existing subset, and in this case there are ω subsets. As two subsets are either strictly included or disjoint, we cannot create another subset, regardless of the way the subsets were initially chosen. We can conclude that a set Ω with ω elements has at most $\omega - 1$ subsets such that each subset has at least cardinality 2 and two subsets are either strictly included or disjoint.

G_i is divided into α connected components of size $\kappa_1, \ldots, \kappa_i, \ldots, \kappa_\alpha$. Applying the previous reasoning for each component gives at most $\kappa_i - 1$ subsets in each component, hence the result. □ □

Lemma 2. *At the end of any iteration $1 \leq i \leq T$, the cardinality of PCC_f is bounded by $min(n-1, i)$.*

Proof. This is trivially true for $i = 1$, the set is then empty. Now suppose it is true for some $1 \leq i \leq T$. Theorem 1 states that two components linked by a dominance relation cannot be together in PCC_f.

Suppose two PCCs p and p' of same length l are present in PCC_f at i. One of them necessarily dominates the other and is alone in PCC_f. Therefore, for a given $1 \leq l \leq T$, there is at most one PCC of length l in PCC_f, so at most i elements in PCC_f. Conversely, two elements of PCC_f must have different sizes, so they are at most $n - 1$ (remember singletons are not considered). □ □

Lemma 3. *The complexity of one iteration of Algorithm 1, including the final one, is $O(n^2)$.*

Proof. Let us first remark that all sets of PCCs considered during one iteration have at most n elements. This is immediate for PCC_c and PCC_f from the two previous lemmas. It is true for PCC_n as it is equal to PCC_c at the end of the iteration. It is also true for PCC_o as it is included into PCC_c from the previous iteration, and for CC by definition. With adequate data structures for these sets, like binary search trees, adding or removing one element from one of these sets is done in $O(\log(n))$ and operations of comparison and intersections can be done in time linear in n.

The first iteration reduces to the assignment of PCC_c which is done in time $n \cdot \log(n)$. So let us concentrate on the other iterations, the final one included. In each iteration of the algorithm, the static connected components of graph G_i are computed. This is done in time $O(m + n)$, which is bounded by $O(n^2)$.

The first step of the algorithm (given in Algorithm 2) contains a loop on the set of connected components CC ($|CC| \leqslant n$) of G_i in which there is a loop on the set PCC_c ($|PCC_c| \leqslant n$). Each element of each set has also its size bounded by n but the elements from CC form a partition of graph's vertices. So a set C_α of CC has size κ_α such that $\sum_\alpha \kappa_\alpha = n$. The comparisons (line 8 of algorithm 2) and the

intersections (line 13 of algorithm 2) between c from the set CC and p from the set PCC_c are made in time $O(\min\left\{|c|\ ;\ |p|\right\})$. For each component C_α of CC, each one of these operations costs $O(n \cdot \kappa_\alpha)$. The total costs $O(n \cdot (\kappa_1 + \cdots + \kappa_{|CC|}))$, so $O(n^2)$. With the same reasoning as for lemma 1, we obtain that the PCCs in PCC_t are either disjoint or strictly included into one another. They only contain vertices from the current component C_α of size κ_α. This set does not contain doubles as the role of function *AddPCC* is to add a PCC to PCC_t without creating doubles. Therefore, the size of PCC_t is bounded by κ_α. We stated earlier that adding an element to PCC_t could be done in time logarithmic regarding the size of the set. The function *AddPCC* compares the lengths of identical vertex sets of PCC in constant time, so adding a PCC to PCC_t with function *AddPCC* can be done in time $\log(\kappa_\alpha)$. Instruction line 20 costs $O(\log(\kappa_\alpha))$ for each p in PCC_c. So this instruction costs $O(n \cdot \log(\kappa_\alpha))$ for each C_α in CC. for the whole algorithm it costs $O(n \cdot (\log(\kappa_1) + \cdots + \log(\kappa_{|CC|})))$. As $\log(\kappa_\alpha) < \kappa_\alpha$, the total complexity of instruction line 20 is bounded by $O(n^2)$. Instruction line 26 costs $O(\log(\kappa_\alpha))$ for each C_α in CC. For the whole algorithm, it costs $O(\sum_\alpha \log(\kappa_\alpha))$, which is bounded by $O(n)$. Instruction line 28 adds all the elements from PCC_t to PCC_n. The size of PCC_n is bounded by n so adding one element is done in time $O(\log(n))$. As the set PCC_t contains at most κ_α elements, adding them all costs $O(\kappa_\alpha \cdot \log(n))$ for each component C_α in CC. For the whole algorithm, it costs $O(\sum_\alpha \kappa_\alpha \cdot \log(n))$ which is bounded by $O(n \log(n))$. In total, algorithm 2 costs $O(n^2)$.

In the second step of the algorithm (given in algorithm 3), both loops contain at most two domination tests made in an inner loop in constant time. This step of PICCNIC also deletes elements from PCC_o or PCC_f. As the size of each one of these sets is bounded by n, it is not possible to delete more than n elements. So the deletion operations costs $O(n \cdot \log(n))$ in total. the number of inner loops is at most $|PCC_o|^2$ for the first loop and $|PCC_o| \cdot |PCC_f|$ for the second one. so both loops have complexity $O(n^2)$. This step also contains operations of differences and unions of PCC sets outside any loop. These operations are done in $O(n \log(n))$. Overall, the complexity of algorithm 3 is also $O(n^2)$.

Finally, the instruction line 10 of algorithm 1 is done in constant time. Therefore, one iteration of PICCNIC costs $O(n^2)$. □ □

Theorem 2. *The complexity of PICCNIC is $O(n^2 \cdot T)$, where n is the number of vertices and T the time horizon.*

Proof. The proof is immediate from the previous lemma, as the number of iterations is $T + 1$. □ □

4.5 Finding all maximal PCCs

The PICCNIC algorithm (given in algorithm 1) identifies all non dominated persistent connected components in the sense of definition 6 in polynomial time with complexity $O(n^2 \cdot T)$. One can also use a slightly modified version of PICCNIC to retrieve every maximal persistent connected component (in the sense of definition 5) in a given dynamic graph.

If we do not compare the PCCs in the second step to delete dominated components, then $PICCNIC_{Step2}$ can be reduced to the given algorithm 6. All the proofs still hold, and this new version of PICCNIC gives all maximal persistent connected component in time $O(n^2 \cdot T)$.

Algorithm 6 $PICCNIC_{AlternativeStep2}$

Input: PCC_n, PCC_c, PCC_f, l_{min}
Output: Updated PCC_f

1: $PCC_o = PCC_c \setminus PCC_n$ *//Set of now finished PCCs*
2: **for all** $p \in PCC_o$ **do**
3:     **if** $l(p) < l_{min}$ **then**
4:         $PCC_o = PCC_o - \{p\}$
5:         STOPLOOP
6:     **end if**
7: **end for**
8: $PCC_f = PCC_f \cup PCC_o$
9: **return** PCC_f

5 Experimental Study

In this section, we propose an experimental study of our algorithm. This is done using the GraphStream Java library[1] (Dutot et al., 2007). The virtual machines used for this experiment have Intel Core 64 bits processors with 8 cores, 4 MB cache size, 2 GHz frequency and 96 GB of RAM. The OpenJDK 1.9 Runtime Environment is used.

We tested our algorithm both on randomly generated graphs and on graphs from real world instances. In the first case, we generated undirected graphs, and in the second case the graphs from real instances are directed. We present both situations next.

[1] http://graphstream-project.org

5.1 Random graphs

Let us first focus on the experiments made on random graphs. We start by presenting the experimental settings chosen for this experiment, then we present the results of PICCNIC Algorithm, and finally we present its execution times.

5.1.1 Experiment Settings

Graph generation In order to evaluate PICCNIC Algorithm, we test it on randomly generated graphs. We do not aim at focusing on too sophisticated graph generation models because our first objective in this part is to verify the tractability of the approach on different classes of dynamic graphs, without focusing on specific applications. The second objective is to provide preliminary results on the pareto curve of the obtained PCCs, and to show the impact of the input graph classes. That is the reason a handcrafted generator was proposed in order to test various dynamic graph families, even though there is a large literature on various generative models, see the seminal paper from Holme (2015) and the more recent survey (Gauvin et al., 2020) on randomized reference models. The proposed generator is a link-driven model with memory mechanism (Karsai et al., 2014) which can be seen as a simplified version of link-node memory models (see for instance (Vestergaard et al., 2014)). Thanks to our model, the edge presence rate is a feature which is maintained over the whole study interval of the dynamic graph. First we generate the structure of the graph (vertices and edges). Then we add dynamicity to the edges using a Markovian process.

First, an underlying graph (V, E) is generated. Its vertices correspond to the ones of the dynamic graph G. Its set of edges includes all sets E_i. The underlying graph is generated using generators from the GraphStream library. We test four different types of graphs that present specific features:

1. Random graphs, corresponding to the Erdős-Rényi model (Erdős and Rényi, 1960). It is the most common way of randomly generating a static graph. GraphStream Random Graph generator is used. This generator adds a vertex and randomly connects it to the other vertices of the graph. This operation is repeated for each vertex added.

2. Regular graphs, which are generated using GraphStream Grid generator. This generator generates a torus with the given number of vertices, all with the same degree.

3. Scale-free graphs, which are used to model many social networks or web networks. Graphstream Barabasi-Albert generator is used. This generator adds a vertex to the graph and connects it to one or several vertices randomly

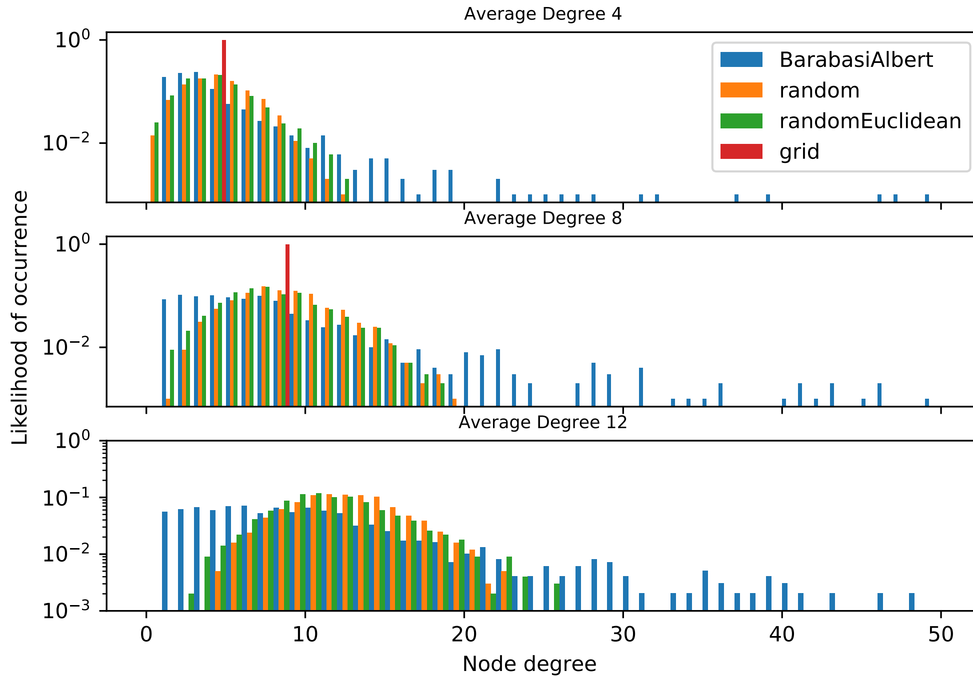


Figure 5: Degree distribution of vertices for each graph type and average degree.

chosen using Barabasi-Albert's (Albert and Barabási, 2002) preferential attachment rule. This operation is repeated for each vertex added.

4. Random geometric graphs, which are particularly well suited for applications with explicit space dimension, such as communication networks or logistic networks. GraphStream's Random Euclidean graph generator randomly places vertices on a finite space $[0, 1] \times [0, 1]$. Two vertices are connected if their euclidean distance is below a given threshold.

Figure 5 presents the degree distribution of vertices from these graphs and Table 1 presents their average clustering coefficients (see (Watts and Strogatz, 1998)). Figure 5 is cropped, indeed, Barabasi-Albert graphs have some vertices with very high degree. As grids are actually toruses, all vertices have same degree, exactly the average degree. Both Random and Random Euclidean graphs have a degree distribution centered on the average degree. In the following, we use the name of the generators.

Once the underlying graph is generated, the dynamics are obtained on edges thanks to a Markov chain. The one used on each edge is presented on Figure 6. When an edge is present at time step i, it remains present at time step $i + 1$ with probability p. When an edge is absent at time step i, it remains absent at time

Degree	Barabasi-Albert	Random	Random Euclidean	Grid
4	0.0221	0,0037	0,5267	0.0000
8	0.0428	0.0084	0.5958	0.0000
12	0.0581	0.0117	0.6040	0.0000

Table 1: Average Clustering Coefficient for each graph type and average degree.

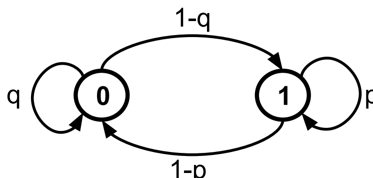


Figure 6: Markov chain representing the states of an arc and probabilities to go from one state to the other. State 0 corresponds to the edge being absent and state 1 corresponds to the edge being present.

step $i + 1$ with probability q.

We introduce a new parameter: the *presence*. It is equal to the stationary probability of edge presence in the Markov chain, often noted π_1 ($\pi_1 \in [0, 1]$). It is asymptotically equal to the rate of presence of each edge over the time study interval.

For a given presence value, there exist many values for p and q. We made experiments (which are not detailed here because it is out of the scope of this paper) and they showed that the values chosen for p and q had negligible influence on the results. Therefore we choose to fix p and q such that $p = \pi_1$ and $q = \pi_0$ ($\pi_0 = 1 - \pi_1$) in the described experiments.

Parameter Selection We plan to evaluate the results of our algorithm as well as its execution time. To this end, some parameters are fixed and the others vary for each experiment.

We considered PCCs of size larger than 2 and of length at least 1. In the algorithm, we have $k_{min} = 2$ and $l_{min} = 1$, which are the default values.

The study interval needs to be long enough so we can observe relevant results. For this reason the number of time steps is fixed to 1000.

To observe the outcome of the algorithm, the number of vertices is fixed to 1000 in order to deal with rather large graphs. In order to obtain execution times as a function of the number of vertices, n takes values in $\{100, 250, 400, 550, 700, 850, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500\}$. These values are enough to obtain good insight on the shape of the execution time curves.

Results are obtained from underlying graphs with average degrees 4, 8 and 12.

	PICCNIC Results	PICCNIC Execution Time
Graph Type	Barabasi-Albert, random, random Euclidean, grid	Barabasi-Albert, random, random Euclidean, grid
n	1000	100, 250, 400, 550, 700, 850, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500
T	1000	1000
Average Degree	4, 8, 12	4
presence	0.7, 0.9	0.9

Table 2: Parameters used for the experiments

Execution times are obtained from underlying graphs with average degree 4. It should be noted that grids are only available with degree 4 and 8.

Values 0.7 and 0.9 are used as presence parameter for the outcome of the algorithm. Only value 0.9 is taken into account to observe the execution time of the algorithm (results were found to be very similar for value 0.7).

In order to obtain statistically relevant observations, 10 instances of graphs are tested for each set of parameters. Table 2 synthesizes the parameters values chosen.

5.1.2 PICCNIC Results

Figures 7, 8 and 9 represent the average Pareto fronts obtained with PICCNIC algorithm for each average degree tested and each type of graphs. Each point represents the average size of non-dominated PCCs, for each possible length.

It can be noticed that the higher the presence, the higher the fronts. When edges have a higher presence, the graph is more connected and thus PCCs are of bigger size.

By comparing Figure 7 and Figure 9, we can notice that the fronts are higher on Figure 9. It means that when the average degree is higher, as the graph is more connected, the PCCs are bigger.

With degree 4 (Figure 7) Random Euclidean graphs do not have components with a high number of vertices. Those graphs do not have "giant" components. With degree 8 and 12, the clustering coefficient of Random Euclidean graphs presented in Table 1 and its degree distribution presented in Figure 5 show that even though Random Euclidean graphs have many clusters, those clusters are highly connected to each other, therefore components are easily kept alive from one time step to the next.

Barabasi -Albert graphs do not have "giant" components. Compared to other graph types, the size of non-dominated components drops drastically as the length of the components increases. This can be explained by the degree distribution.

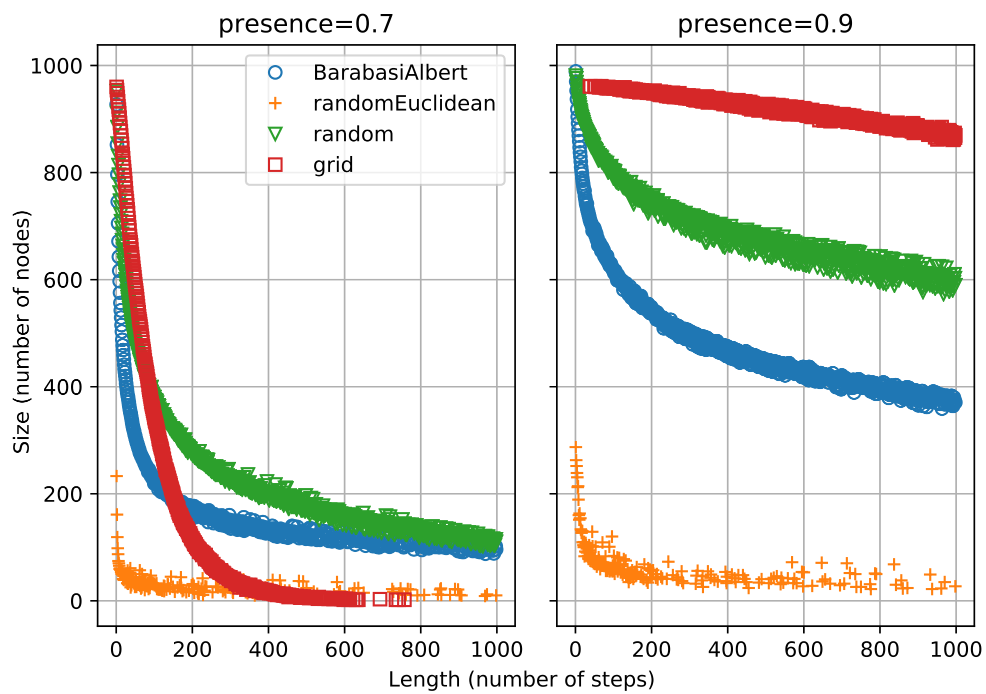


Figure 7: Average values of Pareto front results of PICCNIC algorithm on graphs with 1000 vertices, 1000 time steps, average degree 4. On the left are the results for presence 0.7 and on the right for presence 0.9.

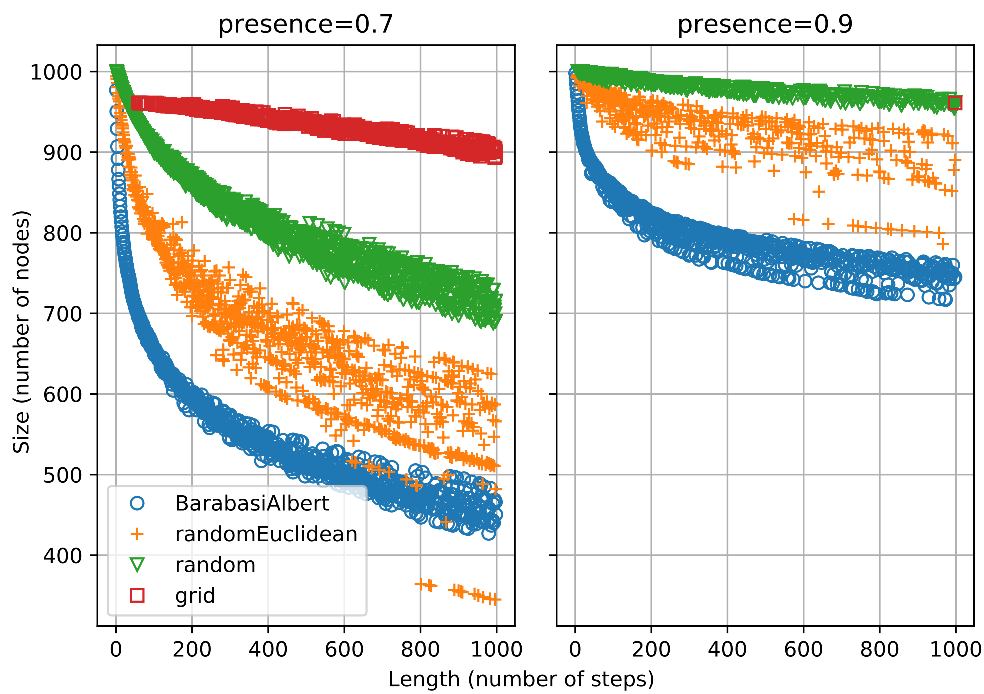


Figure 8: Average values of Pareto front results of PICCNIC algorithm on graphs with 1000 vertices, 1000 time steps, average degree 8. On the left are the results for presence 0.7 and on the right for presence 0.9.

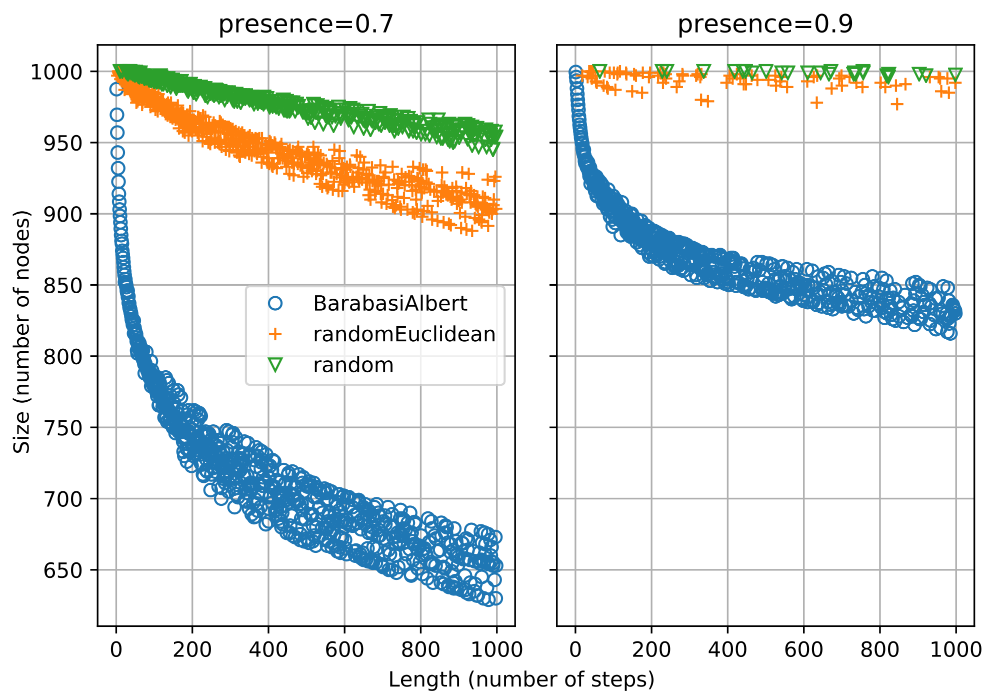


Figure 9: Average values of Pareto front results of PICCNIC algorithm on graphs with 1000 vertices, 1000 time steps, average degree 12. On the left are the results for presence 0.7 and on the right for presence 0.9.

Figure 5 shows that most vertices of those graphs have small degree. Therefore, such a graph is easily disconnected in future time steps. So keeping a large component alive for a long period of time is very unlikely.

For each presence value and average degree, the front corresponding to random graphs is above the one corresponding to Barabasi-Albert graphs. For a given length, a PCC is bigger in random graphs than in Barabasi-Albert graphs. For a low average degree, the front corresponding to Random Euclidean graphs is lower than all other types of graphs whereas with high average degree, it is higher than Barabasi-Albert graphs. PCCs in Random Euclidean graphs are way bigger with a high average degree. Barabasi-Albert graphs present smaller components than Random and Random Euclidean graphs because, as previously explained, they have a majority of vertices with a low degree, therefore there are great chances that such graph breaks into small components.

The front corresponding to grids in Figure 7 for presence 0.9 is high, meaning that PCCs have big size and stay for a long period of time steps. Grids present "giant" components. For presence 0.7, the front drops drastically. PCCs of big size are short and long PCCs are small. Grids are highly connected and robust to changes with a high presence value but they are easily disconnected when presence decreases. In Figure 8, with a presence value 0.9, the graphs are very connected, therefore grids have only one non-dominated component with almost 1000 vertices and length 1000.

5.1.3 PICCNIC Execution Time

To study computation time of the algorithm, we compute it for all four types of graphs, with 1000 time steps, underlying graph average degree 4, presence 0.9 and different number of vertices, from 100 to 4500 (see Table 2).

Figure 10 presents median values of PICCNIC algorithm execution time for each type of graph. For each, a regression function of the form n^2 is also represented. PICCNIC worst case complexity is $n^2 \cdot T$. In this specific experiment, T is fixed (to 1000), so the complexity becomes n^2, hence the regression function of the form n^2.

For Barabasi-Albert graphs, the R^2 value of the regression is 0.984. For grids, it is 0.979, for Random graphs it is 0.992 and for Random Euclidean graphs it is 0.997. Those R^2 values confirm that the regression of the form n^2 fits the experimental values of computational time. The experimental results fit the worst case complexity.

With a higher number of vertices, it is getting clearer that the execution of PICCNIC algorithm takes more time on random Euclidean graphs than on all other types of graphs and that it is faster on grids than on all other types of graphs. When compared to Figure 7, it can be noticed that the algorithm on

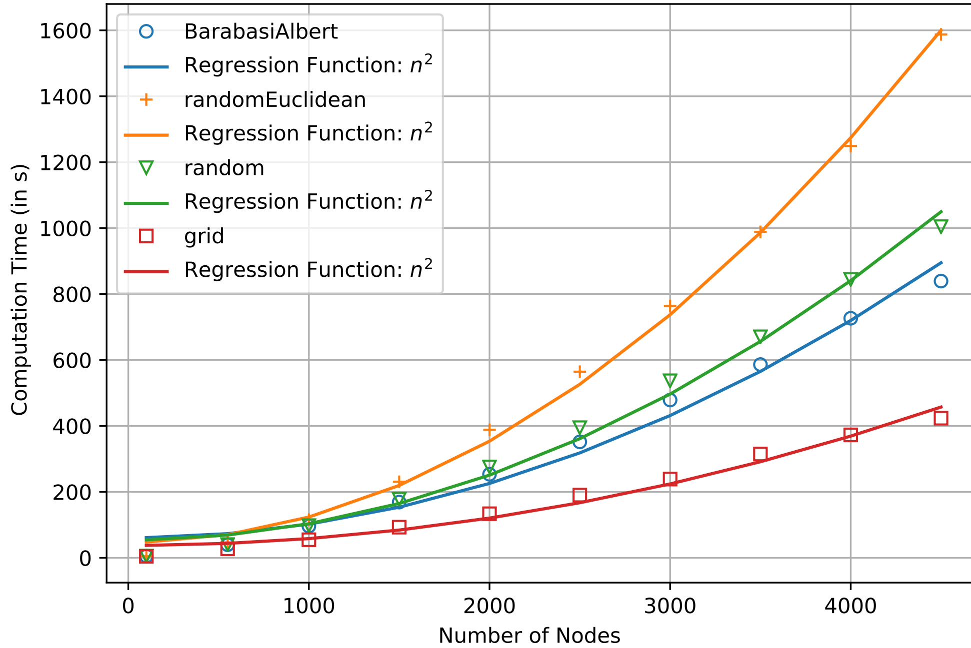


Figure 10: Median execution time of PICCNIC algorithm depending on the number of vertices of the graphs, for each type of graph.

grids, which present big components (lot of vertices for a long period of time), is executed faster whereas the execution takes more time on random Euclidean graphs which present small components. This is consistent with the theoretical observations: the complexity factor is the number of components in the different sets. Although there is a significant difference of computation times between each type of graphs, the order of magnitude remains the same.

The computation time of PICCNIC algorithm, between about 692 seconds to 3045 seconds for graphs with 4500 vertices, depending on the graph type, is quite reasonable and shows that this algorithm can be used in practice for rather large graph sizes. The next section confirms that for real very large graphs.

Computing the whole algorithm in the conditions of the experiment is done in 1000 iterations. On graphs with 1000 vertices, the average computation time of one iteration is 0.056 second for grids, 0.096 second for Barabasi-Albert graphs, 0.096 second for random graphs and 0.116 second for Random Euclidean graphs. On graphs with 4500 vertices, the average computation time of one iteration is 0.422 second for grids, 0.853 second for Barabasi-Albert graphs, 1.054 second for random graphs and 1.577 second for Random Euclidean graphs. Computing one iteration of the algorithm is possible under a very reasonable time limit, so PICCNIC algorithm can be used online while the graph changes.

5.2 Real instances

Let us now present the experiments made on real instances. We start by presenting the data we used, then we present how the dynamic graphs were built based on this data, and then we present the results of PICCNIC Algorithm.

5.2.1 Data used

We used data from the Stanford Network Analysis Projet (SNAP)(Leskovec and Krevl, 2014). This dataset collection offers a large choice of real networks, including temporal networks. We focused on two specific networks.

The first one we worked on is the Stack Overflow temporal network. In this network, each node is a user of the forum Stack Overflow. This network is directed and an arc either represents a user answering another user's question, or a user commenting another user's question or answer. All those interactions happen at a specific time over 2 774 days. There are 2 601 977 nodes and 63 497 050 temporal arcs in this network.

The second network we worked on is the Wikipedia's Talk page temporal network. A node is a user, and an arc represents a user editing another user's talk page. Those interactions happen at a specific time over 2 320 days. There are 1 140 149 nodes and 7 833 140 temporal edges in this network.

5.2.2 Building dynamic graphs from real data

In order to use this data, we had to build dynamic graphs from it. We first had to determine how much real time a time step in the graph represents. We set this amount of time to one day. It means that all the interactions happening the same day appear as arcs in the same t-graph. Similarly, we had to determine how much time we consider an interaction to last. We call this parameter the event duration. We tested several values (see Section 5.2.3). As a time step is one day, we set the event duration to several days. It means that when there is an interaction between two users, we consider that those users are in contact for several days. We considered PCCs lasting at least one time step ($l_{min} = 1$ in the algorithm), and bigger than 100 ($k_{min} = 100$ in the algorithm).

One can wonder what a PCC means in this context, and more specifically, what it means for a set of vertices to be in the same PCC. Both datasets used for this experiment represent the interactions between users of a forum. If two vertices are in the same PCC, it means that the corresponding users have been interacting with each other for a certain amount of time. So a PCC represents a group of users interacting with each other during a period of time. Either on StackOverflow of Wikipedia's Talk Page, it can be inferred that those users are interested in the same subjects.

5.2.3 PICCNIC results

Figure 11 shows the Pareto fronts of non-dominated PCCs obtained from the Wikipedia's Talk Page and from the StackOverflow network. We tested different values of event duration: from 5 to 100 days for Wikipedia's Talk Page and from 5 to 30 days on StackOverflow forum.

In both cases, it can be noticed that the longer we consider an interaction to last, the bigger (in terms of number of vertices) and the longer (in terms of time steps) the PCCs. On Figure 11, the sizes of PCCs are plotted along the Y-axis on a logarithmic scale. The shapes of the Pareto fronts indicate that the biggest PCCs do not last for a long time and the smallest ones last way longer. The fronts drop fast, which is consistent with what we could notice on the previous experiment with the randomly generated graphs.

PICCNIC Algorithm's computation time does not clearly depends on the event duration parameter. The algorithm is executed on the Wikipedia network in about 5 to 7 hours. It is executed on the StackOverflow network in about 55 to 60 hours, moreover, 400 GB of RAM were necessary to run the experiment on this last dataset.

PICCNIC Algorithm successfully identifies the major groups of users interacting with each other along time. This is computed in a large but reasonable amount of time considering the time horizon.

6 Conclusion

In this paper, we proposed a new definition of connected components in a dynamic graph, namely the persistent connected component. Related problems have been addressed in the literature before but unlike most of those works, our definition is not based on journeys in a dynamic graph and does not use travel time but instantaneous connection between vertices. Our generalization is quite natural, as the vertices of a PCC associated to an interval $\mathcal{I}$ belong to the same connected component at each time step of this time interval. Like the extension of connected components found in the literature, our definition of persistent connected components does not form a partition of the graph. A notion of dominance between PCCs was also introduced.

We presented a polynomial time algorithm computing all non-dominated PCCs in a dynamic graph. PICCNIC algorithm has complexity $O(n^2 \cdot T)$, with n the number of vertices and T the time horizon. It is online as it works successively on each time step of the study interval.

The algorithm computes the length of a PCC using the number of time steps. But if we consider that a time step in the study interval corresponds to a time

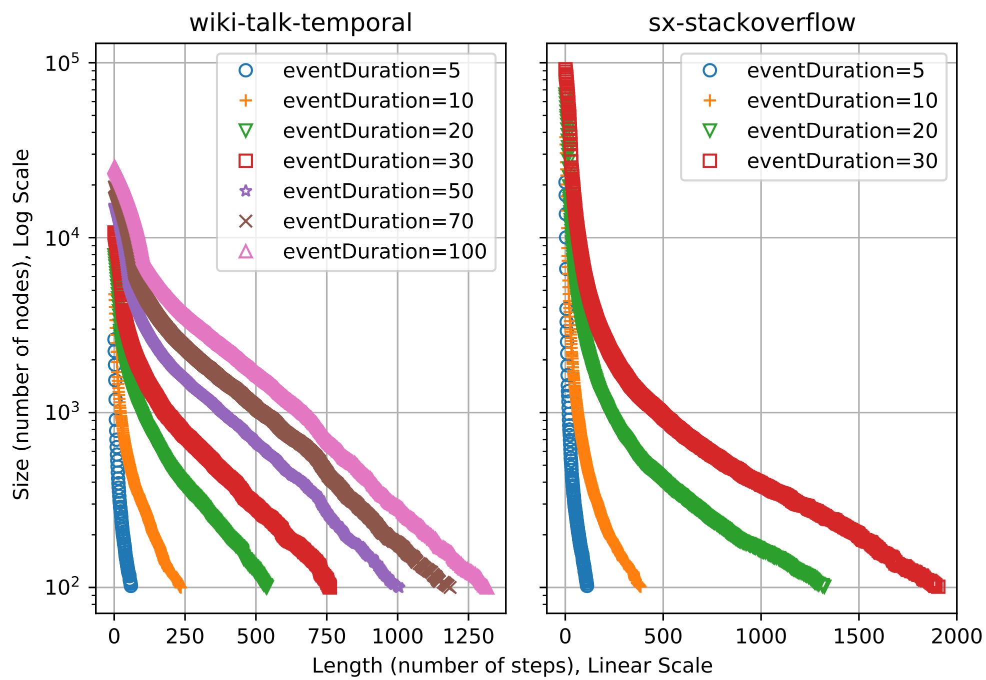


Figure 11: Results of PICCNIC Algorithm on the dynamic graph of Wikipedia's Talk Page on the left and on the dynamic graph of StackOverflow forum on the right. Each event duration tested is represented by one Pareto Front.

when the graph changes, then we can use a model where the actual amount of time elapsed between time steps i and $i+1$ and j and $j+1$ is not the same. PICCNIC algorithm can easily be modified to compute the length of a PCC with “real time” instead of number of time steps.

We presented an experimental study. In the first experiments, we executed PICCNIC on different types of graphs to study the impact of the graph’s structure on PCCs and on its execution time. Then we showed that PICCNIC’s execution time is consistent with its theoretical complexity and that its execution time makes it usable in practice on rather large graphs. In the second experiment, we ran our algorithm on instances made of real data with millions of vertices and arcs. This experiment showed that PICCNIC Algorithm can be used on such real large data. Indeed, its execution time remains quite small with regard to their time horizon.

Another natural extension of connected components seems worth to be investigated. Let us consider that connected components can be interrupted and start again later. From definition 4, it would mean that the vertices stay connected directly or indirectly for l time steps that are not necessarily consecutive. Unfortunately, it is not possible to eliminate dominated connected components without considering all the time steps. Therefore, the number of candidate components might be $\Theta(2^n)$, and finding non dominated ones is not tractable for medium values of n.

Acknowledgements

The project is co-financed by the European Union with the European regional development fund (ERDF) and by the Normandie Regional Council (CLASSE2 project). The authors would like to thank Laurent Chion who worked on a preliminary version of this work for his master thesis.

References

Akrida, E. C., Spirakis, P. G., 2019. On verifying and maintaining connectivity of interval temporal networks. Parallel Processing Letters 29 (02).

Albert, R., Barabási, A.-L., 2002. Statistical mechanics of complex networks. Reviews of modern physics 74 (1), 47–97.

Bhadra, S., Ferreira, A., 2003. Complexity of connected components in evolving graphs and the computation of multicast trees in dynamic networks. In: International Conference on Ad-Hoc Networks and Wireless. Springer, pp. 259–270.

Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N., 2012. Time-varying graphs and dynamic networks. International Journal of Parallel, Emergent and Distributed Systems 27 (5), 387–408.

Casteigts, A., Klasing, R., Neggaz, Y. M., Peters, J. G., 2015. Efficiently testing t-interval connectivity in dynamic graphs. In: International Conference on Algorithms and Complexity. Springer, pp. 89–100.

Démare, T., Bertelle, C., Dutot, A., Lévêque, L., 2017. Modeling logistic systems with an agent-based model and dynamic graphs. Journal of Transport Geography 62, 51 – 65.

Dutot, A., Guinand, F., Olivier, D., Pigné, Y., 2007. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. In: Emergent Properties in Natural and Artificial Complex Systems. Satellite Conference within the 4th European Conference on Complex Systems (ECCS'2007).

Erdős, P., Rényi, A., 1960. On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci 5 (1), 17–60.

Gauvin, L., Génois, M., Karsai, M., Kivelä, M., Takaguchi, T., Valdano, E., Vestergaard, C. L., 2020. Randomized reference models for temporal networks.

Gómez-Calzado, C., Casteigts, A., Lafuente, A., Larrea, M., 2015. A connectivity model for agreement in dynamic systems. In: European Conference on Parallel Processing. Springer, pp. 333–345.

Holme, P., 2015. Modern temporal network theory: a colloquium. The European Physical Journal B 88 (9), 234.

Huyghues-Despointes, C., Bui-Xuan, B.-M., Magnien, C., 2016. Forte Δ-connexité dans les flots de liens. In: ALGOTEL 2016-18èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications. Abstract in English: Strong Δ-connectivity in link streams.

Jarry, A., Lotker, Z., 2004. Connectivity in evolving graph with geometric properties. In: Proceedings of the 2004 joint workshop on Foundations of mobile computing. ACM, pp. 24–30.

Karsai, M., Perra, N., Vespignani, A., 2014. Time varying networks and the weakness of strong ties. Scientific reports 4 (1), 1–7.

Kempe, D., Kleinberg, J., Kumar, A., 2002. Connectivity and inference problems for temporal networks. Journal of Computer and System Sciences 64 (4), 820–842.

Koster, A., Muñoz, X., 2009. Graphs and algorithms in communication networks: studies in broadband, optical, wireless and ad hoc networks. Springer Science & Business Media.

Leskovec, J., Krevl, A., Jun. 2014. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

Michail, O., 2016. An introduction to temporal graphs: An algorithmic perspective. Internet Mathematics 12 (4), 239–280.

Newman, M. E., 2003. The structure and function of complex networks. SIAM review 45 (2), 167–256.

Nguyen, N. P., Dinh, T. N., Xuan, Y., Thai, M. T., 2011. Adaptive algorithms for detecting community structure in dynamic social networks. In: 2011 Proceedings IEEE INFOCOM. pp. 2282–2290.

Nicosia, V., Tang, J., Musolesi, M., Russo, G., Mascolo, C., Latora, V., 2012. Components in time-varying graphs. Chaos: An interdisciplinary journal of nonlinear science 22 (2).

Vestergaard, C. L., Génois, M., Barrat, A., 2014. How memory generates heterogeneous dynamics in temporal networks. Physical Review E 90 (4), 042805.

Watts, D. J., Strogatz, S. H., 1998. Collective dynamics of 'small-world'networks. Nature 393 (6684), 440–442.

Xuan, B. B., Ferreira, A., Jarry, A., 2003. Computing shortest, fastest, and foremost journeys in dynamic networks. International Journal of Foundations of Computer Science 14 (02), 267–285.