Fast streaming small graph canonization

Pedro Paredes, Pedro Ribeiro

CRACS & INESC-TEC
DCC-FCUP, Universidade do Porto, Portugal
pparedes@dcc.fc.up.pt, pribeiro@dcc.fc.up.pt

Abstract. In this paper we introduce the streaming graph canonization problem. Its goal is finding a canonical representation of a sequence of graphs in a stream. Our model of a stream fixes the graph's vertices and allows for fully dynamic edge changes, meaning it permits both addition and removal of edges. Our focus is on small graphs, since small graph isomorphism is an important primitive of many subgraph based metrics, like motif analysis or frequent subgraph mining. We present an efficient data-structure to approach this problem, namely a graph isomorphism discrete finite automaton and showcase its efficiency when compared to a non-streaming-aware method that simply recomputes the isomorphism information from scratch in each iteration.

1 Introduction

The *Graph Isomorphism* problem (GI) consists in finding a bijection between the vertex sets of two graphs that preserves the vertex adjacency or state that one does not exist. It is a widely studied problem in several domains. Its theoretical interest arises from the fact that GI is trivially in NP but is still unknown whether it is NP-COMPLETE or in P, even though it is considered unlikely that GI is NP-COMPLETE [5]. Recently, the upper bound on the complexity was improved to quasipolynomial time [2].

From a practical point of view, it is used as a primitive for several methods that tackle different problems, like frequent subgraph discovery [11], network motif analysis [16] and graph matching [6]. As such, efficient practical methods that compute isomorphism information were developed [9, 14] based on several heuristics. One of the most well-known algorithms is called `nauty`, an exponential algorithm that performs exceptionally well in most inputs.

The *Graph Canonization* problem (GC) is a variant of GI that consists of finding a canonical labelling (also called a *canon*) for a graph such that all isomorphic graphs have the same canon and that if two graphs are not isomorphic they have different canons. Solving GC implies solving GI, since after knowing the canonical labels of two graphs determining if they are isomorphic is simply checking if the two labels are equal. However, in general, GI is not known to be equivalent to GC [1]. The most common practical approach to GI is by solving GC [14], since it is better suited for most applications where a set of graphs needs to be partitioned in isomorphic classes.

The previous discussion focus on algorithms and problems which are static, meaning the input graph or structure is fixed. However, there is an interest in studying graph problems on a dynamic or streaming environment, that is, where the input graph is changing. There are multiple models of streaming graphs [13], that allow for either

edge addition, deletion or both. Particularly in the graph mining realm, there has been an increasing interest in studying dynamic graphs problems, namely, by introducing or altering known metrics to suit temporal graphs (graphs where edges have timestamps that represent intervals of time where they are active) [7, 10, 15].

In this paper we present a new problem that approaches the graph isomorphism problem in a dynamic environment. This formulation considers a streamed graph as a set of operations that add or remove edges in each iteration and it is required to calculate a canonical representation for each intermediate graph. Additionally, we focus on solving this problem for small graphs, that is, undirected graphs that have around 10 vertices or directed graphs that have around 6 vertices. Even though this apparently reduces the applicability of the introduced problem, it is important to note that many graph mining techniques focus on small graphs, like network motif analysis [16] or frequent subgraph mining [11]. In Section 4.3 we present a small case study that shows the applicability of the problem and apply it to practical complex networks.

Our main contribution is an algorithm that solves this problem in an efficient way, when compared to a simpler non-streaming-aware approach that fully recomputes isomorphism in each iteration. This algorithm is based on a data structure that resembles a discrete finite automaton that represents the full isomorphism class space. The method is agnostic in terms of the type of graph, meaning it generic to work with multiple graph types (undirected, directed, vertex and edge coloured, multigraphs, and more), however, in this paper, we only focus on simple undirected and directed graphs.

2 Preliminaries

A *network* or *graph* G is a pair $(V(G), E(G))$, where $V(G)$ is a set of *vertices* and $E(G)$ a set of *edges*, represented by pairs (a, b) where $a, b \in V(G)$. We assume that the graph is simple (no multiple edges or self-loops) an *labelled* so that every vertex of a graph G is assigned a distinct integer from 1 to $|V(G)|$. We denote the label of a vertex v by $L(v)$. For a given graph G we write $V(G) = \{v_1, v_2, \ldots, v_{|V(G)|}\}$ to denote a vertex set where $L(v_i) = i$. *Graph equality* between two graphs G and H is observed if, assuming both $L(g_i) = i, g_i \in V(G)$ and $L(h_i) = i, h_i \in V(H)$, we have $(g_i, g_j) \in E(G) \Leftrightarrow (h_i, h_j) \in E(H)$.

A *permutation* π is an element of the symmetric group S_n, with its usual composition operation $\circ$. We denote the *image* of an integer x under the permutation π by π^x. For a permutation π, we denote by $\overline{\pi}$ the *inverse* of π, such that $\pi \circ \overline{\pi} = \mathbf{1}$, where $\mathbf{1}$ is the identity permutation. A *transposition* is a permutation that only swaps two elements and fixes all others. Given a graph G with vertex set $V(G) = \{v_1, v_2, \ldots\}$ and a permutation π, we denote by G^π the graph with vertex set $V(G^\pi) = \{v_{\pi^1}, v_{\pi^2}, \ldots\}$, meaning we permute the labels. To simplify notation, for a vertex v of a graph G with label i and a permutation π, we write π^v to denote the vertex in G^π with label π^i.

Two graphs G_1 and G_2 are said *isomorphic* if there is a permutation π such that $G_1^\pi = G_2$, we denote this by $G_1 \cong G_2$. The *isomorphism graph class* of a graph G is the equivalence class of G in the relation of isomorphism of graphs. An *automorphism* of a graph G is a permutation π such that $G^\pi = G$. We define $\mathrm{Aut}(G)$ as the set of automorphisms of G. The *orbits* of a graph G are the equivalence classes of vertices of G under the action of automorphisms, this means two vertices u, v have the same orbit

if there is $\pi \in \mathrm{Aut}(G)$ such that $\pi^u = v$ or $\pi^v = u$. A *canonical function* is a function C that, given a graph G, $C(G) \cong G$ and for any $\pi \in \mathrm{Aut}(G)$ we have $C(G^\pi) = C(G)$.

A *graph changing operation* of cardinality n is a pair (x_1, x_2), where x_1 and x_2 are integers between 1 and n. The application of a graph changing operation $\Delta = (x_1, x_2)$ of cardinality n over a graph G with $|V(G)| = n$ is the graph $G' = G\Delta$ with the same vertex set of G, where, if $v, u \in V(G)$ are such that $L(v) = x_1$ and $L(u) = x_2$: if $(v, u) \notin E(G)$ then $E(G') = E(G) \cup \{(v, u)\}$; if $(v, u) \in E(G)$ then $E(G') = E(G) \setminus \{(v, u)\}$. Thus, the application of a graph changing operation (x_1, x_2) is equivalent to toggling on or off the edge between the two vertices with labels x_1 and x_2. A *graph stream* S of cardinality n is a sequence of graph changing operations with the same cardinality. We call the *size* of a stream $|S|$ to the number of elements in S. The application of a graph stream $S = [\Delta_1, \Delta_2, \ldots]$ with cardinality n over a graph G with $|V(G)| = n$ is a sequence of graphs $[G, G\Delta_1, G\Delta_1\Delta_2, \ldots]$, denoted by $S(G)$. For a given stream S over a graph G, if we are only interested in every other k graph, meaning $S(G)_1$, $S(G)_{1+k}$, $S(G)_{1+2k}$, ..., we say the stream S has step k.

2.1 Problem definition

Now that we are armed with the appropriate set of tools, we can define the problem we aim to solve in this paper. We first define the static version of our problem in Definition 1. This problem is essentially providing a graph canonization function C.

Definition 1. *In the static canonization problem we are given a graph G and are asked to provide a canonical representation of G, such that for any $\pi \in \mathrm{Aut}(G)$, G^π has the same representation.*

This problem is a known problem and will be used as a primitive in this paper. We use `nauty` [14] in our method as the solver of this problem. However, note that any method that returns the canon of a graph could be used instead.

We now give the dynamic version of the above problem, which is the focus of this paper, and is included in Definition 2.

Definition 2. *In the dynamic canonization problem we are given a graph G with n vertices and a graph stream S of cardinality n, and we are asked to provide a canonical representation for each graph in $S(G)$.*

Note that, with this formulation, we fix the number of vertices and only vary the edge set. It is also important to note that we focus on small graphs, as stated in the introduction.

3 Proposed method

Our method explores the dimension of the total number of graphs of a certain size to build a data structure that compresses the relationship between their topologies. This data structure is analogous to a deterministic finite automaton (a finite-state machine),

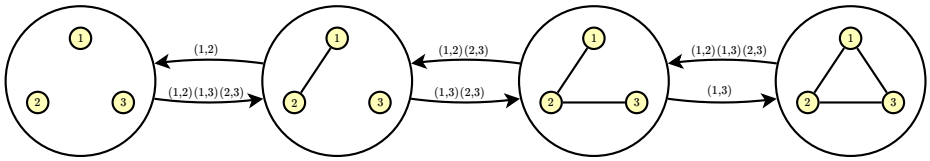

Fig. 1: An automaton representing undirected size 3 graphs

where each node represents a different graph and transitions represent additions or deletions of edges. The result is an algorithm which solves the dynamic canonization problem in an online fashion. We will first describe how the automaton works and how to use it, then we follow up with how to build the automaton efficiently. To avoid ambiguities, we use "node" and "transition" to refer to properties of the automaton and "vertex" and "edge" to refer to properties of the graphs represented by the automaton.

3.1 The automaton

As mentioned above, we use a data structure that is analogous to an automaton to support our algorithm. This will be used as we iterate through each graph in $S(G)$ to follow the isomorphism graph class.

The node set of the automaton represents the different isomorphism graph classes of a fixed number of vertices n. For each different class, we fix one label function and associate to it a single node of the automaton. This equates to fixing a permutation per isomorphism class and using it as a canonical labelling. For each node, there is one transition coming out of it per possible pair of two vertices of the underlying graph. Each one of this transitions represents an edge toggle, meaning an addition or removal of an edge to the represented graph, which depend on whether the two vertices of this transition are connected or not on the represented graph. Thus, the destination of each transition is the node whose isomorphism graph class is the one of the altered graph. We portray a pictorial representation of this object in Figure 1.

Since every change between two consequent graphs in $S(G)$ is described by a single pair of vertices it is natural to use the described automaton to follow the isomorphism graph class of each graph by walking through the automaton. On each step, we use the transition which is associated with the pair of vertices on the current graph changing operation. Initially, the automaton starts on the node that represents the empty graph with n vertices. To find the node that represents G, we build G by following all transitions that represent the pairs of vertices on each edge of the graph, in any order. Subsequently, each graph changing operation results in following one transition.

However, this is not enough to actually apply the automaton, since the order of vertices that was fixed on a certain node may not be the same as the one the current graph we are considering from $S(G)$. Thus, we keep a permutation π_p that tells us how to change the order of vertices of the current graph in order to have the same graph as the one the current node represents. If we think about labels, let G_c be the current graph and G_n be the graph represented by the current node (by definition we have $G_c \cong G_n$), π_p has the following property: $L(G_c) \circ \pi_p = L(G_n)$, since the label function works like a permutation from vertices to indices and taking $\circ$ as regular permutation composition.

To accommodate this change, we also need to update how the transitions work, since after following a transition the relation between the current graph and the graph represented by the current node may change. Thus, we associate a permutation with each transition that informs on how to update π_p. If the permutation for a certain transition is P, then the new π'_p is obtained by $\pi'_p = \pi_p \circ P$. Initially, π_p is set to the identity permutation, since the initial node represents the empty graph (where every permutation is valid). Note that in Figure 1 the permutations were omitted for brevity.

The resulting automaton represents all different graphs of size n and can be used to keep track of the canonical representation of a graph after a vertex pair change by following a transition and composing a permutation. If we are applying a change of vertex pair (a, b), we follow the transition related to (π^a_p, π^b_p), since we always work on top of the representation the automaton gives.

3.2 Building the automaton

Now that we have described how the automaton works and how to use it, we will specify how to build it. There are two important aspects here that heavily influence the complexity of the building process, but also the complexity of using the automaton. The first is how to fix the graph each node represents. The second is when to build the automaton, since we can pre-build it or build it as we process the graph stream. We will answer the first question through the following explanation and also point out why it is a relevant question. Regarding the second, we will first describe an on-the-fly method and then a method that pre-builds the automaton but leads to a more efficient automaton.

On-the-fly method In order to fix a canonical order for each node, we use the representation `nauty` provides. Our method to dynamically build the automaton is based on following the supposed transitions as the stream is processed. Whenever we find ourselves on a non existing node, we run `nauty` to know where we should be and either create a new node or point the transition to the correct destination. Additionally, we fill the transition permutations accordingly.

The only node we pre-build is the node that represents the empty graph. Afterwards, we will process each new vertex pair (a, b). Let $a_p = \pi^a_p$ and $b_p = \pi^b_p$. On processing a new pair, we first check if the transition of (a_p, b_p) was already created. If not, we first run `nauty` on the transformed graph, that is, if G is the current graph after adding or removing the edge induced by (a, b), we do so on G' where $L(G') = L(G) \circ \pi_p$, meaning the graph from the current node altered by the pair (a_p, b_p). We do so because `nauty` not only returns the canonical adjacency matrix that we will use to represent the automaton node, but also a permutation P that transforms the graph represented by the canonical adjacency matrix into G'. We can then create a new transition by (a_p, b_p) from the current node $\boldsymbol{C}$ to the new node $\boldsymbol{N}$ (found with `nauty`) with permutation P, since this permutation transforms the graph on $\boldsymbol{C}$ with added vertex pair (a_p, b_p) into the graph on $\boldsymbol{N}$, which is the same that `nauty` returns.

This was implicit in the previous paragraph, but we also need a bookkeeping mechanism to store the node representations, so as to avert having a duplicated node representing the same graph class. This can be done using a dictionary data structure that maps

canonical representations, as obtained through `nauty`, to automaton nodes (if they exist). Since the graph representation is fixed by the `nauty` canonical representation, the method described in the previous paragraph is exactly the same whether the destination node ($\boldsymbol{N}$, in the previous paragraph's notation) has to be created or not. If the node is missing, we simply create a new node and feed it to the bookkeeping dictionary.

When processing a change (a, b), let P be the permutation `nauty` returns, $\boldsymbol{C}$ be the initial automaton node and $\boldsymbol{N}$ the destination node. Since $\overline{P}$ transforms graphs in the $\boldsymbol{C}$ representation to the $\boldsymbol{N}$ representation, the converse is also true, that is, P transforms graphs in the $\boldsymbol{N}$ representation to the $\boldsymbol{C}$. Thus, we can use this information to right away fill another transition, P, from $\boldsymbol{N}$ to $\boldsymbol{C}$. However, since the representation changed, the vertex edge associated with this transition is not (a_p, b_p) but rather $(\overline{P}^{a_p}, \overline{P}^{b_p})$, since this is the corresponding edge pair in $\boldsymbol{N}$.

It is important to note that the real temporal bottleneck of using this automaton lies on the application step rather than the building step, as we will observe in Section 4. This means that the advantage of using a dynamic building method is only observable if the full automaton is impossible to be generated. For example, if we are applying the method in an instance graph with a high number of vertices, but where we know the total number of different graph types in the stream is low, using the dynamic building method we only build a partial automaton, according to the input.

Consequently, it is useful to optimise the automaton underlying representation and methodology if this improves the runtime of applying, even if it worsens the building procedure. With this in mind, we can compress the permutations associated with each transition in order to avoid iterating n integers. By observing the different canonical representations given by `nauty`, one can observe that they are fairly regular, meaning that often if two graphs differ by a single edge, their adjacency matrix only differ in one (or two, in the undirected case) entries. This implies that the permutation associated with the transition between the two is simple, often either the identity or a single transposition. Thus, we can compress these cases to a special representation that instead of composing a permutation with π_p, either does nothing or simply swaps two entries of π_p. We will see a detailed analysis of the effect of this in Section 4, but theoretically this would lower the complexity of following a transition.

Pre-building method There are not many points to improve related to the on-the-fly building process, since this method does the bare minimum to know where each transition leads to. Consequently, our pre-building method works very similarly, but it does a depth-first search on the automaton in the beginning, generating all possible nodes and transitions. However, the advantage of doing a method that pre-computes the automaton is that it is easier to fix a different representation of graphs per node, since there is no need to follow the canonical representation given by `nauty` (or to have one that works regardless of the order with which we build the automaton). This is important since changing the underlying representation changes the permutations associated with each edge and this has a direct effect on their compressibility and thusly on the runtime.

It easy to prove that composing a permutation to the graph of each node does not change the correctness of the algorithm, as long as we update the transitions accordingly, since we are simply projecting the automaton to a different space. Hence, it is easy to change the underlying representation of each node by composing a permutation

to the permutation `nauty` returns during the create new transition procedure, as long as we compose the same permutation to each transition coming into the same node. In practice, we are simply changing the representation to one that better suits our goals.

All that is left is to choose which permutations to compose with. Instead of focusing on individual permutations, one can determine the underlying representation and choose the permutation that generates this representation. To choose a representation, we can choose the order under which we initially traverse the automaton to pre-build it and use the first graph to touch a each node as its representation. To implement this, the permutation we compose with each node is simply $\overline{P}$ (borrowing from the previous subsection's notation), where we fix the permutation P obtained on the first time we visit that node (which is when we actually create the node). This results in choosing the identity permutation as the permutation from $\boldsymbol{C}$ to $\boldsymbol{N}$ on the first visit to the node.

Different orders were tested, with the goal of increasing the percentage of transitions whose permutation was either the identity permutation or a single transposition. It would be possible to implement an optimisation algorithm here, like a local search algorithm, that would repeatedly perturb the traversal order. Although, this would be computationally heavy and would probably not yield much better results than a simply greedy approach. Consequently, we chose an altered edge lexicographical order, that is, we first follow all pairs that create edges before any pair that removes edges and we break ties choosing the lexicographical first transition vertex pairs. We tested different approaches, but this one yielded the better results.

Note that, for graphs with 4 or more vertices, it is impossible to build an automaton where each transition permutation is either the identity permutation or a single transposition. This is equivalent to saying that the graphs in two adjacent aumaton nodes differs by at most one edge. To prove the impossibility premise, we will assume that it is possible to build an automaton for 4 vertex undirected graphs. Consider Figure 2, which represents a partially constructed automaton for 4 vertex graphs. We omit the multiple transitions between nodes and simply fix each node's graph representation and show the relationships between nodes (where one or more transitions would be present). In this example, there is a mismatch between node $\boldsymbol{G}$ and $\boldsymbol{H}$, since their graph's adjacency matrices differs by more than an edge. We can show that this example is "canonical", meaning that all possible automata for 4 vertex graphs are equivalent to this example, but here we omit the formal proof because of space constraints.

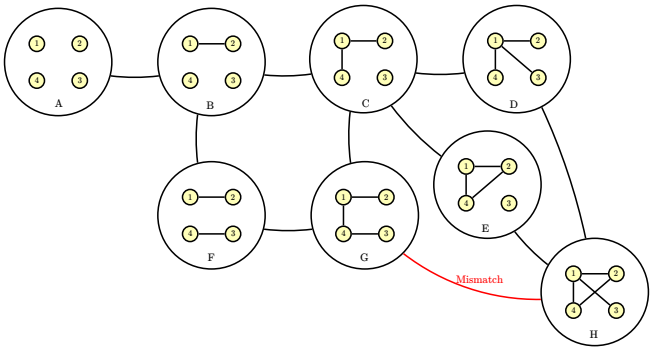

Fig. 2: A partial automaton representing undirected size 4 graphs

	On-the-fly				Pre-build			
	Undirected		Directed		Undirected		Directed	
	C_0	C_1	C_0	C_1	C_0	C_1	C_0	C_1
3	25%	75%	31%	73%	33%	78%	34%	69%
4	18%	52%	25%	62%	24%	53%	29%	56%
5	14%	39%	20%	53%	20%	44%	26%	46%
6	12%	29%	-	-	15%	30%	-	-
7	9%	21%	-	-	11%	22%	-	-
8	6%	15%	-	-	11%	19%	-	-

Table 1: Values of C_0 and C_1 for different automata and build methods

4 Analysis

4.1 Theoretical analysis

First of all, a note on the automaton's general behaviour. Let $\mathcal{G}_n$ denote the set of different graphs with n vertices (note this is an agnostic analysis, since it works for both undirected and directed graphs). Let E_n be the maximum number of edges for a graph with n vertices, that is, $E_n = n^2$ for directed graphs and $E_n = n(n+1)\frac{1}{2}$ for undirected graphs. Since the automaton has one node per different isomorphic graph and each node has a transition per possible pair of vertices, it has $|\mathcal{G}_n|$ nodes and $|\mathcal{G}_n|E_n$ transitions. These pose as the main bottleneck of the automaton method, since they are directly related with memory usage, where each node holds a canonical label and each transition a permutation and destination node. Since $\mathcal{G}_n$ grows rapidly with n, this method is only appropriate to small graphs, depending on the available memory.

For the base building on-the-fly method, we run `nauty` once per transition pair (since we build a transition and its reverse per `nauty` call), thus we call it $|\mathcal{G}_n|E_n\frac{1}{2}$ times. To follow a transition of the automaton, if it exists, it is necessary to compose a permutation, which takes at most $\mathcal{O}(n)$ time for a graph with n vertices. This is true if we have the default representation, if the permutation to compose with can be compressed, then the time needed is only $\mathcal{O}(1)$.

4.2 Empirical analysis

This analysis is based on our implementation of the described method in `C++`, which is publicly available [1]. Our C++ code is compiled with `GCC 4.8.3`, and runs on a single core of an AMD Opteron(tm) with 2.30 GHz under Fedora 20, with 4GB of RAM. Here we focus on two main themes: namely the compressibility of the transition permutations and the runtime of using the automaton versus using a simpler base approach, namely recalculating the isomorphism class for every instance using `nauty`.

We define two notions of compressibility: C_0 is the *zero compressibility* of an automaton, meaning the percentage of transition permutations that are the identity permutation; C_1 is the *one compressibility* of an automaton, meaning the percentage of transition permutations that are either a single transposition or the identity permutation. In Table 1 we show the C_0 and C_1 values for some automata of different sizes, both undirected and directed, for the two building methods. We omit the results pertaining to automata that were too memory intensive to compute (directed sizes 6, 7 and 8).

[1] https://github.com/ComplexNetworks-DCC-FCUP/streaming-small-isomorphism

Designation	Direction	$\|V(G)\|$	Origin	Step
`ER-6, ER-7, ER-8`	Undirected	6, 7, 8	ER Model	1
`PR-6, PR-7, PR-8`	Undirected	6, 7, 8	PR Model	1
`SW-5, SW-6, SW-7`	Undirected	5, 6, 7	SWAP Model	4
`dER-4, dER-5`	Directed	4, 5	D-ER Model	1
`dPR-4, dPR-5`	Directed	4, 5	D-PR Model	1

Table 2: Graphs used for the experimental analysis

It is clear that the pre-build method achieves better compressibilities, specially C_0 compressibilities, which are more critical in terms of runtime. If we discount the building time, which is slightly higher for the pre-build method (but constant), in general, this results in a speedup of up to 2 times, for most input graph streams. However, the increased building time means that for higher vertex numbers (from 8 up) the runtime advantage only becomes noticeable for larger stream sizes. This result was obtained empirically using the graph streams used in the analysis of the following paragraphs.

To compare the temporal behaviour of our method with the base `nauty` recomputation method we generated several synthetic networks, with different goals and variants. Here we use the version using the on-the-fly building method. We selected 13 graph streams descriptions with different properties and for each one studied the runtime of our method and of the base recomputation method for several stream sizes. We summarise them in Table 2, where the step k is the number of graph changing operations between each canonization request, that is, we are only interested on the canonization of every other k element of $S(G)$, as we defined previously.

The following list summarises each model used to generate graph streams:

- **ER Model**, is based on the Erdos-Rényi [4] random graph model, where each graph changing operation is chosen uniformly at random from all the possible vertex pairs. Its directed version, the **D-ER Model** is analogous.
- **PR Model**, is based on a preferential attachment rule for networks [3] where each vertex pair is chosen as a graph changing operation depending on the degree of each of its vertices. Its directed version, the **D-PR Model** is analogous.
- **SWAP Model**, simulates edge swapping operations, each 4 contiguous graph changing operation represent swapping two edges (chosen uniformly at random). It has a step of 4 because we are only interested in the graphs after each swap.

To study each one we generated multiple streams with increasing sizes, from 10^4 to 10^7 and observed the runtime of both methods. We plot the results of that analysis in Figure 3 (note that the X axis is in logarithmic scale). The top left figure pertains to the undirected models, the top right figure directed models, the bottom left figure contains all streams based on the `SWAP` model and the bottom right figure represents a growing step experiment that will be further explained bellow.

It is noticeable that our method greatly outperforms the base method on all streams. Furthermore, the asymptotic behaviour of our method suggests that for even greater stream sizes the benefit will increase. The same applies to the speedups obtained. For the unit step streams, the speedup grew approximately linearly from about 1 up to 15 times. For the `SWAP` model the speedup was more stable, varying between 2.7 and 3.1. It is also interesting to note that our method had very similar results for different stream

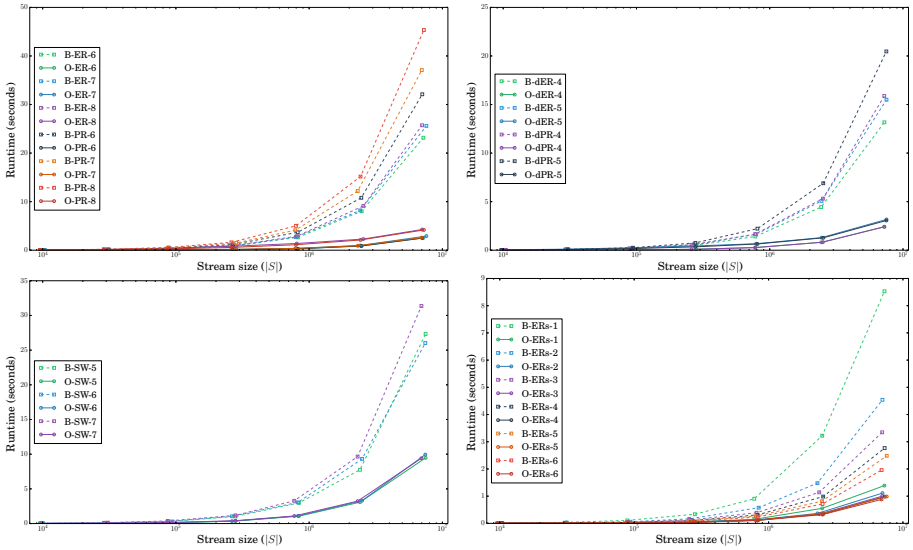

Fig. 3: Comparison of our method (solid lines and prefix `O-`) versus the base method (dashed lines and prefix `B-`) for multiple streams.

models with the same number of vertices, whereas the base method was much more input dependant, which shows that our method is agnostic to the input source.

In the bottom left figure, regarding the `SWAP` model, it is interesting to note that, even though there is a step of 4, our method still maintains a good speedup when comparing to the base method. Note that the higher the step the worse is the benefit of our method, since the base method only performs computation when it is required to return a canonical label whereas our method has to update the automaton after each change.

There is a clear tipping point observable in the data, which represents the minimum stream size for which it is more beneficial to use our method instead of the base method. For the top left figure, it appears to be around 10^5. This value is related to the automaton size and with the number of times that the method needs to run `nauty` in the building time. We can extrapolate from here and estimate for different streams sizes and different inputs (even with a number of vertices higher than memory restrictions would allow) and estimate how good our method is going to be in relation to the base method.

Building on this tipping point argument, the bottom right figure shows a growing step experiment. We used the `ER` model to generate various networks with 6 vertices and artificially vary the step from 1 to 6 (each integer in the figure legend indicates the step of that measure). It is important to point out that for all different steps, our method outperformed the base method, with decreasing speedups. Additionally, as we increase the step, the mentioned tipping point of efficiency also increases. Further similar experiments indicate that there is always a tipping point when the step is of the order of $\mathcal{O}(n)$, which means our method is useful as long as the average number of edge modifications between required canonical labels is in the order of the number of vertices.

4.3 Case study

To further show the usefulness of our method, we present a brief case study problem and present a solution based on our method. Recently, there have been many contributions

Designation	Name	Direction	$\|V(G)\|$	$\|S(G)\|$	Origin
email	email-eu-core	Directed	986	332,334	Communication [12]
college	college-msg	Directed	1,899	20,296	Communication [12]
infectious	infectious	Undirected	410	17,298	Social [8]
arxiv	arxiv-hep-th	Undirected	22,908	2,673,133	Coauthorship [12]

Table 3: Graphs used for the case study

	Using the base method				Using our method			
	email	college	infectious	arxiv	email	college	infectious	arxiv
3	10.86	8.81	8.33	32.81	6.62	5.08	3.19	31.52
4	22.55	17.12	17.81	66.72	10.73	8.62	4.98	60.48
5	34.45	30.01	34.36	113.95	16.24	13.34	8.56	98.74

Table 4: Runtimes, in seconds, for the case study analysis

to the study of network motif and subgraph counting analysis in temporal graphs, as stated in Section 1. Here we present a problem formulation that is inserted in this trend.

Let a G be a temporal graph with edges changing. We want to analyse how patterns evolve in this network and for that we will focus on how a determined induced subgraph of G in a certain timestamp evolves through time. Thus, given two graph types H_1 and H_2 (with the same number of vertices, and possibly the same), we want to know the percentage of times that a set of nodes in a certain timestamp in G is isomorphic to H_1 and in a future timestamp isomorphic to H_2. If we do this for all possible graphs H_1 and H_2 of a certain size n, then we get a Markov chain of temporal subgraph transitions that can be used as a fingerprint of the network and be further used for multiple graph mining tasks This technique is similar to what was done in [7], but here only patterns of at most 3 vertices were studied, and to what was done in [15], but here this was done in a edge oriented fashion and with a slightly different formulation.

Doing a complete search of all possible patterns and transitions is possible, but very heavy, even for a relatively small network. Because of that, we only consider connected induced subgraphs and we propose an approximated approach to this problem. We will first sample a single connected induced subgraph H from G in any timestamp. We then follow the vertex set of H through time in G. To do so, we use our automaton to first represent H and then follow the edge changes We fix a time step δ, such that whenever δ units of time have passed, we record the current isomorphism class and add a transition on the Markov chain table from the previous class to the current one. By doing so, we can follow the isomorphism information of that particular vertex set throughout the whole life time of G. If we repeat this procedure enough times, we have effectively sampled a portion of the temporal transition space.

Since our goal is not to provide a graph mining method to the stated problem but to showcase a possible usage of our automaton, here we will not discuss this method much further. We implemented a basic version of this approach and ran it using both the base method and our method as the underlying isomorphism tool. To compare their runtimes, we ran them on a small set of complex networks with 1,000,000 samples, which we list in Table 3. The runtimes obtained for multiple subgraph size n are shown in Table 4. These runtimes include the time for sampling and performing other supporting computation, which lowers the speedup in relation to what was seen in Section 4.

5 Conclusion

In this paper we introduced a new problem consisting on computing graph isomorphism on a fully dynamic streaming environment, supporting edge insertion and deletion. We presented an efficient algorithm that tackles this problem using a data structure similar to a discrete finite automaton to represent the full space of different isomorphism classes. Compared to a simple non-streaming-aware approach of recomputing the solution for each iteration of the stream, the automaton method and its variations obtained a much better performance, with speedups increasing with the stream size. We also briefly studied the applicability of our method, studying how the stream parameters (ex: the stream size and the stream step) vary while keeping the usefulness of our method in relation to the simpler approach, and he have shown a possible application.

Acknowledgments: This work is partly financed by ERDF within project "POCI-01-0145-FEDER-006961", by FCT as part of project "UID/EEA/50014/2013", and by FourEyes, a research line within "TEC4Growth/NORTE-01-0145-FEDER-000020" financed by NORTE2020 through ERDF.

References

1. Arvind, V., Das, B., Köbler, J.: The space complexity of k-tree isomorphism. In: Int. Symposium on Algorithms and Computation. pp. 822–833. Springer (2007)
2. Babai, L.: Graph isomorphism in quasipolynomial time [extended abstract]. In: 48th Annual ACM SIGACT Symposium on Theory of Computing. pp. 684–697. ACM (2016)
3. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science 286(5439), 509–512 (1999)
4. Erdos, P., Rényi, A.: On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci 5(1), 17–60 (1960)
5. Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. Journal of the ACM (JACM) 38(3), 690–728 (1991)
6. Gori, M., Maggini, M., Sarti, L.: Exact and approximate graph matching using random walks. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(7), 1100–1111 (2005)
7. Huang, H., Tang, J., Liu, L., Luo, J., Fu, X.: Triadic closure pattern analysis and prediction in social networks. IEEE Trans. on Knowledge and Data Eng. 27(12), 3374–3389 (2015)
8. Isella, L., Stehlé, J., Barrat, A., Cattuto, C., Pinton, J.F., Van den Broeck, W.: What's in a crowd? analysis of face-to-face behavioral networks. Journal of Theoretical Biology 271(1), 166–180 (2011)
9. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: 9th Workshop on Algorithm Engineering and Experiments. pp. 135–149 (2007)
10. Kovanen, L., Karsai, M., Kaski, K., Kertész, J., Saramäki, J.: Temporal motifs in time-dependent networks. J. Stat. Mechanics: Theory and Experiment 2011(11), P11005 (2011)
11. Kuramochi, M., Karypis, G.: An efficient algorithm for discovering frequent subgraphs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1038–1051 (2004)
12. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (Jun 2014)
13. McGregor, A.: Graph stream algorithms: a survey. ACM SIGMOD Rec. 43(1), 9–20 (2014)
14. McKay, B.D., Piperno, A.: Practical graph isomorphism, ii. Journal of Symbolic Computation 60, 94–112 (2014)
15. Paranjape, A., Benson, A.R., Leskovec, J.: Motifs in temporal networks. In: 10th ACM International Conference on Web Search and Data Mining. pp. 601–610. ACM (2017)
16. Wernicke, S.: Efficient detection of network motifs. IEEE/ACM Transactions on Computational Biology and Bioinformatics 3(4) (2006)