Neighborhood-based Pseudo-canonical Representation of Graphs

Fürst, Luka

Abstract: ***A canonical representation of a graph is a string that is invariant to the numbering of vertices. Therefore, two graphs have the same canonical representation if and only if they are isomorphic. However, constructing a canonical representation is as hard as the graph isomorphism problem itself, so it is worthy to seek approximate solutions. In this paper, we propose a polynomial-time algorithm for computing a pseudo-canonical representation of a graph, a string with the property that the equality of strings implies isomorphism, whereas the opposite implication is highly desirable but not required. The algorithm renumbers the vertices based on their direct and indirect neighborhoods. In particular, it initially considers only the immediate neighborhoods but then iteratively spreads the neighborhood data over the entire graph, producing increasingly robust vertex numberings and, in turn, graph representations. We applied the algorithm to artificial and real-world graphs. For a vast majority (99.95 %) of undirected simple graphs with up to 9 vertices, the string produced by our algorithm can be regarded as canonical, since it is the same for all possible initial vertex numberings.***

Index Terms: ***graph, isomorphism, symmetry, canonical representation, pseudo-canonical representation, vertex numbering, graph invariant, adjacency matrix***

1. Introduction

The importance of networks in today's world can hardly be overstated. Social networks such as Facebook or Twitter have had a profound impact on our personal, professional, and political lives, and they continue to grow in size and prominence. Other types of networks (computer networks, transportation networks, electronic circuits, chemical structures, etc.) might be less conspicuous and possibly not even immediately recognized as networks, but that does not make them any less vital for the modern human existence. The ever-increasing size of networks compels us to design efficient data structures and algorithms for their representation, analysis, and manipulation. Networks can be straightforwardly represented by graphs: graph vertices represent individual network entities, and graph edges represent connection between the entities. Therefore, to make it possible to process large networks, we have to focus on developing efficient graph algorithms [5].

One of the most fundamental problems in graph theory is the *graph isomorphism problem*, the goal of which is to determine whether two graphs are *isomorphic*, i.e., have the same structure. This problem has numerous applications: for example, in the domain of network analysis, it is completely natural to ask whether a (sub)network is structurally equal to another (sub)network; in chemistry, questions of this type might be even more frequent. Interestingly, the graph isomorphism problem is one of the very few members of the class GI, which is considered to lie between the complexity classes P and NP [8, 16]. To date, no algorithm has been discovered that solves *every* instance of the problem in time proportional to some polynomial over the graph size, although Babai [1] proposed an algorithm that runs in quasi-polynomial time.

The graph isomorphism problem would be easy to solve if graphs could be efficiently represented by strings in such a way that a pair of graphs would be isomorphic if and only if their representations were equal. With the abil-

Manuscript received May 2019.

The author is affiliated with the Faculty of Computer and Information Science, University of Ljubljana, Slovenia (e-mail: luka.fuerst@fri.uni-lj.si).

ity to build such a representation, the problem of graph isomorphism could be solved simply by transforming the given two graphs to their respective strings and comparing the strings for equality. As it has turned out, such a representation does exist: it is called a *canonical* representation [2, 6, 9, 14]. A canonical representation must be *numbering-invariant*, depending only on the structure of the graph, not on the numbering of vertices or edges. For a given graph, an algorithm for producing a canonical representation has to output the same string regardless of how its vertices are numbered.

Canonical representations are especially beneficial when multiple graphs have to be compared with a single graph. An example of such an application is the problem of enumerating all non-isomorphic subgraphs of a given host graph [4, 10]. This problem can be solved by enumerating all subgraphs and retaining only those that are not isomorphic to any of the subgraphs discovered up to that point. Here, every existing graph has to be compared with every newly discovered one, so it certainly pays to convert graphs to strings and compare strings rather than graphs [12].

Canonical representations are perhaps most widespread in chemistry. The SMILES notation is a string that uniquely determines a given molecule [17]. Two molecules are isomorphic if and only if they have equal SMILES strings.

Unfortunately, converting a graph to a canonical representation is at least as hard as finding the graph isomorphism itself. Indeed: if a canonical representation could be computed in polynomial time, the graph isomorphism problem could be solved in polynomial time as well, since a pair of strings can be compared in linear time. In many applications, however, a canonical representation need not be 'truly' canonical. For instance, we may stipulate that the equality of strings imply the isomorphism of graphs but not necessarily the other way around. If a representation satisfies this requirement for every pair of graphs, it will be called *pseudo-canonical*. However, while it is not required that a pair of isomorphic graphs convert to the same string, this property is still desirable; the more often a pseudo-canonical string uniquely represents a graph, the better the representation. In any case, it should be possible to compute a pseudo-canonical representation in polynomial time; otherwise, there would be no reason not to compute a true canonical representation instead.

In this paper, we present a polynomial-time algorithm that computes a pseudo-canonical string for a given graph. The algorithm iteratively builds increasingly robust representations; the longer it runs, the more likely it is that the string constructed for the input graph is truly canonical, i.e., completely insensitive to the initial numbering of graph vertices. However, as we will see, there are graphs for which the algorithm never produces a numbering-invariant string, regardless of how long it runs. In fact, such graphs exist for any polynomial algorithm; if they did not, that algorithm could be used to compute graph isomorphism in polynomial time.

We applied the proposed algorithm to a complete set of connected undirected simple graphs with up to 9 vertices. Out of 274 292 such graphs, only 133, or 0.05%, have the property that different initial numberings might lead to different pseudo-canonical strings; for all the rest, the pseudo-canonical strings produced by the algorithm are invariant to the initial vertex numbering. To compare the performance of our algorithm with that of the canonical representation algorithm built into the Sage system,[1] we applied both algorithms to a few real-world graphs.

To the best of our knowledge, the algorithm itself, as well as its analysis and applications, may be regarded as an original contribution to science. In the research work leading to this paper, we combined several idea generation approaches [3]. The most important was probably *Mendeleyevization*: for many problems that are unlikely to be solvable in polynomial time, researchers have invented at least one polynomial-time approximation method, but there appears to be none in the case of computing canonical representations, at least for general graphs.

Our algorithm for computing a pseudo-canonical representation is based on linearizing the adjacency matrix of the graph. As we

[1] `http://www.sagemath.org/`

will see, while such a representation is indeed pseudo-canonical (equal strings imply isomorphic graphs), it is not particularly useful. It is overly sensitive to the initial numbering of vertices, so it is too often the case that isomorphic graphs are converted to different pseudo-canonical strings. We therefore first renumber the vertices in a robust way, so that the likelihood of different initial numberings leading to different representations becomes significantly lower. Our renumbering algorithm is based on vertex neighborhoods. Initially, the vertices are ordered and numbered by their degrees. In the next step, the vertices having an equal degree are ordered by the ordered sequences of their neighbors' numbers, etc. Sometimes, ties still have to be broken artificially, although even in this case the obtained representations are likely to be robust to the initial numbering and can therefore be used as a reliable indicator of isomorphisms.

The rest of this paper is structured as follows. In the section PRELIMINARIES, we define the terms that will be used throughout the paper. In the section CONSTRUCTING A PSEUDO-CANONICAL REPRESENTATION, we show how to construct a pseudo-canonical representation based on a given vertex numbering. In the section ALGORITHM, we present an algorithm for renumbering the graph vertices so that the resulting pseudo-canonical representation will be robust. In the section COMPUTATIONAL COMPLEXITY, we determine the algorithm's computational complexity. The section EXPERIMENTAL RESULTS presents some experimental results, and the section CONCLUSION brings our paper to a conclusion.

2. PRELIMINARIES

A graph G is a pair (V_G, E_G), where V_G is the set of *vertices* and $E_G \subset V_G \times V_G$ is the set of *edges*. We will write V and E instead of V_G and E_G when the graph G is irrelevant or known from context. For the sake of simplicity, we will restrict our discussion to simple undirected graphs, i.e., $(v,v) \notin E$, $(u,v) \in E \iff (v,u) \in E$ for all u, $v \in V$. However, the proposed algorithm for computing a pseudo-canonical representation could be easily adapted to cover directed graphs and graphs with loops.

Without loss of generality, we will assume that $|V| = n$ and $V = \{1, 2, \ldots, n\}$. A vertex u will be called to be *smaller* (*greater*) than a vertex v if $u < v$ ($u > v$). The set of all neighbors of a vertex u will be labeled $\mathcal{N}(u)$, i.e., $\mathcal{N}(u) = \{v \in V \mid (u,v) \in E\}$.

Graphs G and H are *isomorphic* ($G \simeq H$) if there exists a bijective function $h\colon G \to H$ that preserves both adjacencies and non-adjacencies, i.e., for each pair of vertices u, $v \in V_G$ we have $(u,v) \in E_G \iff (h(u), h(v)) \in E_H$. Such a function h is called an *isomorphism*.

A string $z^*(G)$ is a *canonical representation* of a graph G if $z^*(G) = z^*(H) \iff G \simeq H$ for each graph H. A string $z(G)$ will be called a *pseudo-canonical representation* of a graph G if $z(G) = z(H) \implies G \simeq H$ for each graph H. In other words, graphs with the same string $z(.)$ have to be isomorphic, but the opposite is not required, albeit, as we mentioned in introduction, highly desirable.

A sequence (vector) $A = (a_1, \ldots, a_s)$ is *lexicographically smaller* than a sequence (vector) $B = (b_1, \ldots, b_t)$ (denoted $A \prec B$) if either of the following is true:

- $s < t$ and $a_i = b_i$ for all $i \in \{1, \ldots, s\}$;
- there exists an index $i \in \{1, \ldots, \min\{s, t\}\}$ such that $a_i < b_i$ and $a_j = b_j$ for all $j < i$.

3. CONSTRUCTING A PSEUDO-CANONICAL REPRESENTATION

In this section, we show a general method for constructing a pseudo-canonical representation for a graph $G = (V, E)$ with $V = \{1, \ldots, n\}$. First, for a pair of vertices u, $v \in V$ such that $u < v$, let

$$s(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E, \\ 0 & \text{if } (u,v) \notin E. \end{cases} \qquad (1)$$

Now, for $u \in \{1, \ldots, n-1\}$, let

$$S_u = s_{u,u+1} s_{u,u+2} \ldots s_{u,n}, \qquad (2)$$

and let

$$z(G) = S_1/S_2/\ldots/S_{n-1}. \qquad (3)$$

For example, for graphs G_1 and G_2 in Figure 1, we have

$$z(G_1) = 1100/010/10/1,$$
$$z(G_2) = 0001/110/01/1.$$

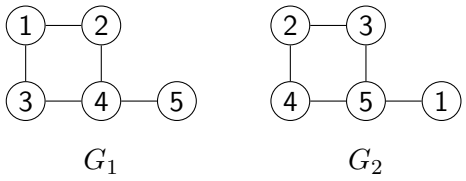

Figure 1: *A pair of sample graphs.*

Theorem 1. Given a graph G, the string $z(G)$ is a pseudo-canonical representation of the graph G.

Proof. Let $z(G) = a_1 a_2 \ldots a_k$ and $z(H) = b_1 b_2 \ldots b_l$. We have to show that $z(G) = z(H)$ implies $G \simeq H$. Clearly, if $z(G) = z(H)$, then G and H have the same number of vertices; otherwise, it would not have been possible for both strings to have the same length. Consequently, the separators / are located at the same positions in both strings, and if $a_i \in \{0, 1\}$, then a_i and b_i correspond to the same pair (u, v) (with $u < v$) in both G and H. From this it immediately follows that $(u, v) \in E_G \iff (u, v) \in H_G$ provided that $u < v$. However, since the graphs are undirected, this property holds for all pairs (u, v). Therefore, the graphs G and H are isomorphic. □

Meanwhile, for directed graphs with possible loops, the substrings S_u would have to be defined as $S_u = s_{u,1} s_{u,2} \ldots s_{u,n}$. It could be easily shown that a string $z(.)$ constructed from such substrings S_u is a pseudo-canonical representation. In either case, the pseudo-canonical representation of a graph can be obtained by linearizing its adjacency matrix.

4. Algorithm

Unfortunately, the pseudo-canonical representation defined by (3) is, by itself, far from being insensitive to different numberings of the vertices. For example, the graphs G_1 and G_2 in Figure 1 are isomorphic, but we have seen that they have different pseudo-canonical strings. The presented representation thus fails to meet our expectations; informally, a good representation should be numbering-invariant in a vast majority of cases.

Our idea is as follows: before constructing the pseudo-canonical string using the method described in the section Constructing a Pseudo-Canonical Representation, we apply a robust *renumbering* algorithm. Ideally, the algorithm should renumber the vertices of a given graph in such a way that the representation defined by (3) remains the same regardless of the initial numbering. Therefore, if a pair of graphs differ only in the numbers of their vertices (in other words, if they are isomorphic), the renumbering algorithm will ensure that the pseudo-canonical strings defined by (3) will be equal. The problem of finding a good pseudo-canonical representation has thus been reduced to the problem of a robust renumbering of the vertices.

A robust renumbering algorithm should rely on vertex properties that are independent of the numbering of the vertices. Our algorithm relies on the direct and indirect neighborhoods of individual vertices. Consider the graph in Figure 2. (The labels A, B, etc. are not the actual labels of the vertices; they only make referencing easier. The graph is unlabeled.) In this graph, the vertex E has a single neighbor, the vertex D has three, and the vertices A, B, and C have two neighbors each. Initially, the algorithm orders the vertices by increasing degree and thus assigns the number 1 to the vertex E, the number 5 to the vertex D, and the number 2 to the vertices A, B, and C, since they cannot be distinguished at this point. Note that this assigment is completely independent of the initial numbering of the vertices. To break ties, the algorithm gradually propagates the adjacency information over the entire graph. The vertices B and C are both connected with a vertex that received the number 2 and a vertex that received the number 5 in the previous step, whereas the vertex A is connected with two vertices that were given the number 2. Consequently, the

vertex A retains the number 2, whereas the vertices B and C both receive the number 3. The process continues in a similar fashion. Since the vertices B and C can never be distinguished, the number of either can be changed to 4. In this case, the resulting numbering is not completely invariant to the initial numbering, but the pseudo-canonical string is the same regardless of which of B and C receives 3 and which 4. Therefore, all initial numberings give rise to the same pseudo-canonical string, and that string, in fact, meets the criteria to be called a true canonical string.

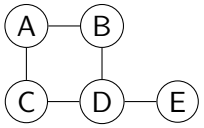

Figure 2: *A sample graph.*

Algorithm 1 computes a robust vertex numbering for a given graph G. The entry point of the algorithm is the function MAIN. At the beginning, the function assigns the number (id) 1 to all vertices. After that, the function enters the main loop, which repeatedly calls the function PROCEED and, if necessary, the function BREAKATIE until every vertex has received a unique number.

The function PROCEED operates in a loop. In the k-th iteration, it constructs a set W of all vertices that were assigned the number k. A singleton set $|W|$ means that the sole vertex in W already has a unique number. In this case, we take no further action; once the number of a vertex becomes unique, that number will not change any more. In the opposite case, the function takes two steps: (1) for each vertex w in W, it first builds a sequence T_0 composed of the numbers of the neighbors of w and then sorts the sequence T_0, obtaining a sequence $T(w)$; (2) it assigns a number to each vertex w in the set W based on the position of the sequence $T(w)$ within a lexicographically ordered sequence of the sequences $T(w_1)$, $T(w_2)$, ..., $T(w_{|W|})$. The vertex with the lexicographically smallest sequence $T(.)$ receives the smallest number, etc. The vertices having equal sequences $T(.)$ obtain the same number and will be dealt with later.

If an invocation of the function PROCEED does not change the number of any vertex, we have to modify at least one non-unique number 'artificially', or else we fall into an infinite loop. The function BREAKATIE finds the smallest number k^* such that there are at least two vertices having that number and builds a set W of vertices with the number k^*. For all vertices in W except the smallest, the function increases the number by 1 and thus breaks the tie just enough for the algorithm to continue.

Figure 3 illustrates the execution of the robust numbering algorithm using the 6-cycle graph as an example. At the beginning, all vertices receive the number 1. The function PROCEED then computes the sequences $T(.)$ and finds out that all of them are equal to $\langle 1, 1\rangle$, since each vertex has two vertices numbered 1 in its immediate neighborhood. To break the tie, the function BREAKATIE sets the number of the vertex A to 1 and the numbers of all other vertices in the tie to 2. The vertex A has just received a unique number; this number will therefore remain the same until the end. In the next call of the function PROCEED, we obtain $T(B) = T(F) = \langle 1, 2\rangle$ and $T(C) = T(D) = T(E) = \langle 2, 2\rangle$. Consequently, the vertices B and F retain the number 2, but the vertices C, D, and E receive the number 4. The process continues until every vertex has been assigned a unique number. Note how the information about vertex neighborhoods, being strictly local at the beginning, gradually spreads over the entire graph.

5. COMPUTATIONAL COMPLEXITY

The computational complexity of the robust vertex numbering algorithm can be estimated by considering the following two observations:

Lemma 2. In each iteration of the function MAIN, at least one vertex obtains its final number.

Proof. To begin with, note that a unique number is also final; once a vertex receives a unique number, it will not change any more, since the function PROCEED processes only non-singleton sets W. If the function PROCEED re-

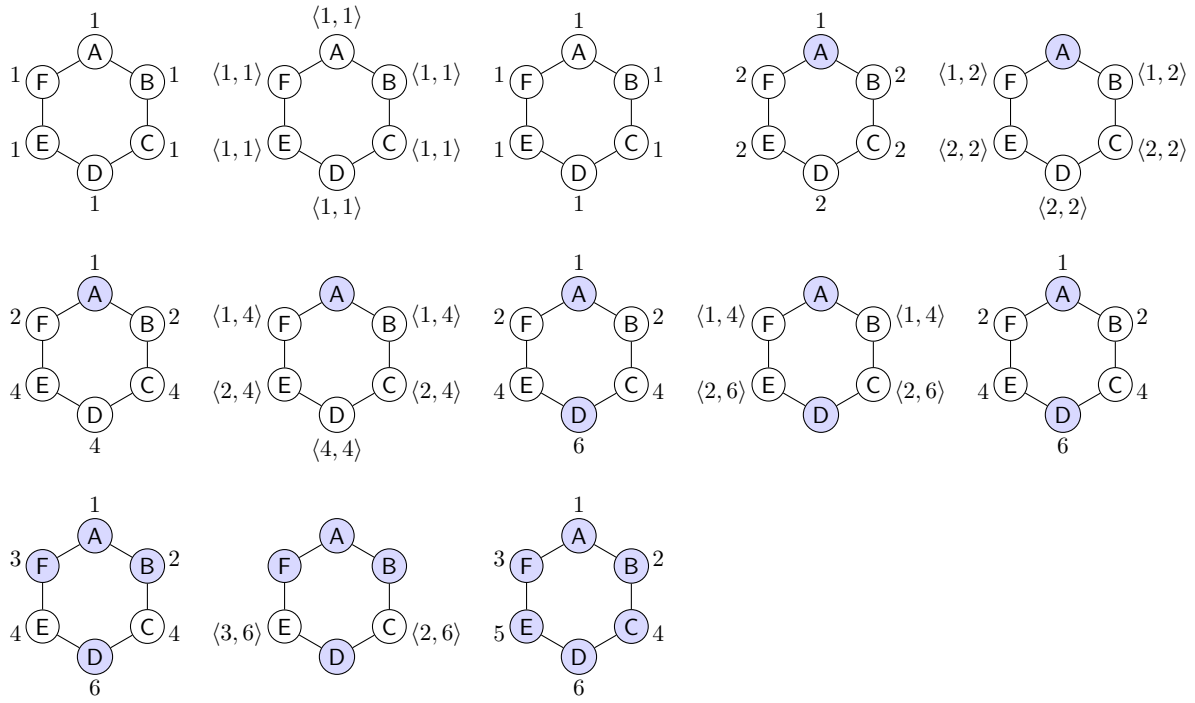

Figure 3: *The renumbering algorithm applied to the 6-cycle.*

turns *false*, the function BREAKATIE assigns a unique (and hence final) number to some vertex in a set of vertices with equal non-unique numbers. Alternatively, if PROCEED terminates with result *true*, it has either assigned a unique number to at least one vertex or increased the numbers of at least one non-singleton set of vertices. Although the numbers in this set are not unique, at least one vertex from the set will retain its number until the end. □

Lemma 3. Each iteration of the function MAIN runs in time $O(n^2 \log n)$.

Proof. For each vertex w with a non-unique number, the function PROCEED lists its neighbors and sorts their numbers to obtain $T(w)$. Since there are at most n neighbors, this can be achieved in time $O(n \log n)$ per vertex and $O(n^2 \log n)$ for the entire graph. The next step, i.e., finding the position of the sequence $T(w)$ within the lexicographically ordered sequence of sequences $T(1)$, $T(2)$, ..., $T(n)$, can be performed by sorting. Since each of the sequences $T(1)$, ..., $T(n)$ has $O(n)$ elements, sorting the sequence of these sequence takes $O(n^2 \log n^2) = O(n^2 \log n)$ time. The function PROCEED thus performs in time $O(n^2 \log n)$. □

By combining these observations, we readily arrive at the following result:

Theorem 4. The time complexity of Algorithm 1 is $O(n^3 \log n)$.

Proof. Since each iteration of the loop in the function MAIN assigns a final number to at least one vertex, the function performs at most n iterations. Since each iteration takes $O(n^2 \log n)$ time, we can conclude that the robust vertex numbering algorithm runs in time $O(n^3 \log n)$. □

6. EXPERIMENTAL RESULTS

Algorithm 1 renumbers the vertices of a given graph G in such a way that the pseudo-canonical string $z(G)$ is likely to be the same regardless of the initial vertex numbering. Such a string $z(G)$ can thus be regarded as a canonical representation for the graph G. To determine whether Algorithm 1 gives rise to a numbering-invariant string $z(G)$ for a given graph G, we

Algorithm 1 *Robust renumbering of graph vertices.*

```
 1: function Main(G = (V, E))
 2:     for all v ∈ V do
 3:         id(v) := 1
 4:     end for
 5:     repeat
 6:         changed := Proceed(G, id)
 7:         finished := (∀u, v ∈ V : id(u) ≠ id(v))
 8:         if ¬finished ∧ ¬changed then
 9:             BreakATie(G, id)
10:         end if
11:     until finished
12:     return id
13: end function
14: function Proceed(G = (V, E), id)
15:     id' := id
16:     for all k ∈ {1, 2, ..., |V|} do
17:         W := Peers(V, id', k)
18:         if |W| > 1 then
19:             for all w ∈ W do
20:                 T_0 := ⟨id'(w') | w' ∈ N(w)⟩
21:                 T(w) := Sort(T_0)
22:             end for
23:             for all w ∈ W do
24:                 P := {w' ∈ W | T(w') ≺ T(w)}
25:                 id(w) := |P| + 1
26:             end for
27:         end if
28:     end for
29:     return (id ≠ id')
30: end function
31: function BreakATie(G = (V, E), id)
32:     k* := min{k | |Peers(V, id, k)| > 1}
33:     W := Peers(V, id, k*)
34:     for all w ∈ W \ {min(W)} do
35:         id(w) := id(w) + 1
36:     end for
37: end function
38: function Peers(V, id, k)
39:     return {v ∈ V | id(v) = k}
40: end function
```

run Algorithm 1 and build a string $z(G)$ for each of the $n!$ possible initial numberings of the graph vertices. If all resulting strings are equal, the renumbering algorithm can be said to 'work' for the graph G, and the resulting string can be considered to be a canonical representation for G.

As we already know, the function Main performs at most n iterations, but the n-th iteration never changes any number. (For example, if the graph has two vertices, the first iteration changes the number 1 to 2 for one of the vertices, whereas the second only finds out that both numbers are now unique.) If the algorithm is terminated before the last iteration that actually changes something, the numbers of the vertices will not be unique, so we will have to distinguish between vertices with equal numbers. It makes sense to update the numbers of those vertices so that their order matches the initial vertex numbering. For example, if the algorithm running on the 6-cycle (Figure 3) were stopped upon obtaining the numbering $\{A \mapsto 1,\ B \mapsto 2,\ C \mapsto 4,\ D \mapsto 4,\ E \mapsto 4,\ F \mapsto 2\}$, that numbering would be updated to $\{A \mapsto 1,\ B \mapsto 2,\ C \mapsto 4,\ D \mapsto 5,\ E \mapsto 6,\ F \mapsto 3\}$.

We will call a graph *i-canonicalizable* if for all $n!$ initial vertex numberings, all of the numberings obtained after running the function Main for i iterations give rise to the same pseudo-canonical string $z(G)$. Conversely, if there are at least two initial numberings leading to at least two distinct strings $z(G)$ after performing i iterations of the function Main, the graph will be called *non-i-canonicalizable*. A graph is *canonicalizable* if it is n-canonicalizable and *non-canonicalizable* if it is non-n-canonicalizable. The fact that the numbering algorithm is polynomial implies the existence of non-canonicalizable graphs; if such graphs did not exist, Algorithm 1 could be used to solve the graph isomorphism problem in polynomial time. However, as we shall see, such graphs are rare.

We tested our algorithm using the entire set of simple undirected graphs with up to 9 vertices. Let $\mathcal{G}_n$ denote the set of simple undirected graphs with n vertices. For each $n \in \{2, \ldots, 9\}$ and $i \in \{1, \ldots, n-1\}$, Table

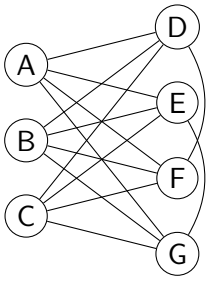

Figure 4: *The graph G^*: the smallest non-canonicalizable graph.*

1 shows the number of non-i-canonicalizable graphs within the set $\mathcal{G}_n$. The graph G^* in Figure 4 is the smallest non-canonicalizable graph and the only such member of the set $\mathcal{G}_7$. For example, the initial numbering $\{A \mapsto 1,\ B \mapsto 2,\ C \mapsto 3,\ D \mapsto 4,\ E \mapsto 5,\ F \mapsto 6,\ G \mapsto 7\}$ results in the final numbering $\{A \mapsto 1,\ B \mapsto 6,\ C \mapsto 7,\ D \mapsto 2,\ E \mapsto 4,\ F \mapsto 3,\ G \mapsto 5\}$ and, in turn, in the pseudo-canonical string $111100/10011/0011/111/11/0$, while the initial numbering $\{A \mapsto 2,\ B \mapsto 3,\ C \mapsto 4,\ D \mapsto 1,\ E \mapsto 5,\ F \mapsto 6,\ G \mapsto 7\}$ leads to the numbering $\{A \mapsto 1,\ B \mapsto 3,\ C \mapsto 4,\ D \mapsto 5,\ E \mapsto 6,\ F \mapsto 2,\ G \mapsto 7\}$ and the representation $111100/11100/0011/011/11/1$. As we can see, the graph G^* is regular; all vertices have the same degree. Unsurprisingly, regular graphs pose the greatest challenge to the renumbering algorithm.

In another set of experiments, we built pseudo-canonical strings for selected real-world graphs from the datasets KONECT [11] and SNAP [13]. In the pre-processing stage, we removed all vertex labels, changed directed edges to undirected, removed loops, and turned every occurrence of multiple edges between the same pair of vertices into a single edge. This time, we were interested in the performance of our renumbering algorithm rather than the canonicalizability of individual graphs. In particular, we compared the performance of Algorithm 1 with the algorithm for computing true canonical strings that is part of the Sage system. Algorithm 1 was implemented in the C programming language. All experiments were carried out on a computer with a 8-core Intel Core i7-3770 processor operating at a clock rate of 3,40 GHz.

The results of the experimental validation of our approach are gathered in Table 2. In addition to time consumption in milliseconds (t_{PC} for Algorithm 1 and t_{C} for the canonicalization algorithm in the Sage system), we also show the number of iterations (N) of the loop in the function MAIN. As expected, the polynomial-time renumbering algorithm was almost in all cases faster than the worst-case exponential algorithm in Sage. The only exception was the (fairly dense) graph Advogato; however, to answer the question why, in this particular case, our algorithm takes more time than the one in Sage would require a more thorough investigation. Because of a large number of vertices and enormous number of possible initial vertex numberings, it is untractable to determine whether the computed pseudo-canonical strings can be regarded as canonical representations, although, after trying out numerous random permutations, it appears likely to be so.

7. Conclusion

We introduced a method for constructing a pseudo-canonical representation of a graph. The method is based on an algorithm for renumbering the graph vertices in a manner that is robust to the initial numbering of the vertices. Initially, the vertices are all assigned number 1. Subsequently, in each iteration, the ties are broken by sorting the vertices in the order of increasingly ordered sequences of their neighbors' numbers. If no tie is broken in this way, one vertex in a group of equally numbered vertices retains its number, but all others in the same group are assigned a number higher by 1. The initially strictly local neighborhood data thus iteratively propagate through the graph. Since each iteration of the algorithm assigns a final number to at least one vertex, the algorithm runs for at most n iterations, with each iteration taking time $O(n^2 \log n)$ owing to sorting $O(n)$ sequences of size $O(n)$.

The numbering produced by the renumbering algorithm immediately gives rise to a pseudo-canonical string (if two strings are equal, the corresponding graphs are isomorphic but not necessarily the other way around). However, in many cases, as shown by the exper-

Table 1: *The number of non-i-canonicalizable simple undirected graphs with n vertices.*

		i							
n	$\lvert\mathcal{G}_n\rvert$	1	2	3	4	5	6	7	8
2	1	0							
3	2	0	0						
4	6	2	1	0					
5	21	12	7	3	0				
6	112	88	52	27	7	0			
7	853	781	427	235	89	20	1		
8	11 117	10 873	6032	2780	1164	319	52	19	
9	262 180	260 105	145 392	44 952	16 295	4691	899	201	113

Table 2: *Results on real-world graphs.*

Graph	$\lvert V\rvert$	$\lvert E\rvert$	t_{PC}	t_{C}	N
Les Misérables	77	254	1,8	8,8	28
David Copperfield	112	425	0,7	5,6	5
Jazz	198	2742	4,5	16,9	10
US Power Grid	4941	6594	32 700	58 700	481
Advogato	6541	39 285	230 000	124 000	2013
ca-HepTh	9877	25 973	615 000	1 450 000	2266
ca-CondMat	23 133	93 439	9 220 000	23 700 000	5999

iments, the pseudo-canonical string is invariant to the initial numbering of the vertices and can be regarded as a canonical representation (two strings are equal if and only if the corresponding graphs are isomorphic). In particular, out of 274 292 simple undirected graphs with up to 9 vertices, only for 133 it holds that there are at least two initial numberings that give rise to distinct pseudo-canonical strings.

Our approach to building pseudo-canonical strings is related to graph symmetries. For instance, in the fully symmetric graph K_6 (the complete graph on 6 vertices), it is immaterial how we number the vertices; the pseudo-canonical string will be the same for all $6!$ vertex numberings. Regarding the graph C_6 (Figure 3), there are fewer degrees of freedom; to keep the pseudo-canonical string fixed, only the numbers of the vertices A, C, and E can be freely permuted, whereas the numbering of the vertices B, D, and F has to 'follow' the numbering of the other three (or vice versa). This concept is closely related to that of *exploratory equivalence* [15, 7]. It is thus worthy to determine the exact relationship between the two concepts and to investigate if and how one could benefit another.

References

[1] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Symposium on Theory of Computing (STOC 2016), Cambridge, MA, USA*, pages 684–697, Cambridge, Massachusetts, USA, 2016.

[2] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Symposium on Theory of Computing (STOC 1983), Boston, MA, USA*, pages 171–183, 1983.

[3] V. Blagojević, D. Bojić, M. Bojović, M. Cvetanović, J. Đorđević, Đ. Đurđević, B. Furlan, S. Gajin, Z. Jovanović, D. Milićev, V. Milutinović, B. Nikolić, J. Protić, M. Punt, Z. Radivojević, Ž. Stanisavljević, S. Stojanović, I. Tartalja, M. Tomašević, and P. Vuletić. A systematic approach to generation of new ideas for PhD research in computing. In A. R. Hurson and V. Milutinović, editors, *Creativity in Computing and DataFlow SuperComputing*, volume 104 of *Advances in Computers*, chapter 1, pages 1–31. Elsevier, 2017.

[4] Diane J. Cook and Lawrence B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[5] Dragoš Cvetković and Slobodan K. Simić. Spectral graph theory in computer science. *IPSI BgD Transactions on Advanced Research*, 8(2):35–42, 2012.

[6] Michael Elberfeld and Pascal Schweitzer. Canonizing graphs of bounded tree width in logspace. *ACM Transactions on Computation Theory*, 9(3):12:1–12:29, 2017.

[7] Luka Fürst, Uroš Čibej, and Jurij Mihelič. Maximum exploratory equivalence in trees. In *Federated Conference on Computer Science and Information Systems (FedCSIS 2015), Łódź, Poland*, pages 507–518, 2015.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[9] Aimin Hou, Qingqi Zhong, Yurou Chen, and Zhifeng Hao. A practical graph isomorphism algorithm with vertex canonical labeling. *Journal of Computers*, 9(10):2467–2474, 2014.

[10] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review*, 28(1):75–105, 2013.

[11] Jérôme Kunegis. KONECT: the Koblenz network collection. In *International Web Observatory Workshop (WWW 2013), Rio de Janeiro, Brazil*, pages 1343–1350, 2013.

[12] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.

[13] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu /data`, 2014.

[14] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

[15] Jurij Mihelič, Luka Fürst, and Uroš Čibej. Exploratory equivalence in graphs: Definition and algorithms. In *Federated Conference on Computer Science and Information Systems (FedCSIS 2014), Warsaw, Poland*, pages 447–456, 2014.

[16] Jacobo Torán. On the hardness of graph isomorphism. *SIAM Journal on Computing*, 33(5):1093–1108, 2004.

[17] David Weininger. SMILES, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of Chemical Information and Computer Sciences*, 28(1):31–36, 1988.

Luka Fürst received his Ph.D. in computer science from the University of Ljubljana in 2013. He is employed as a teaching assistant at the University of Ljubljana, Faculty of Computer and Information Science. His research areas include graph grammars, general graph theory (with a particular emphasis on graph symmetries), software engineering, machine learning, and computer programming education.