Proceedings of the 2014 Federated Conference on Computer Science and Information Systems pp. 447–456
DOI: 10.15439/2014F352
ACSIS, Vol. 2

Exploratory Equivalence in Graphs: Definition and Algorithms

Jurij Mihelič, Luka Fürst, and Uroš Čibej
University of Ljubljana, Faculty of Computer and Information Science
Tržaška cesta 25, SI-1000 Ljubljana, Slovenia
Email: {jurij.mihelic,luka.fuerst,uros.cibej}@fri.uni-lj.si

***Abstract*—Motivated by improving the efficiency of pattern matching on graphs, we define a new kind of equivalence on graph vertices. Since it can be used in various graph algorithms that explore graphs, we call it exploratory equivalence. The equivalence is based on graph automorphisms. Because many similar equivalences exist (some also based on automorphisms), we argue that this one is novel. For each graph, there are many possible exploratory equivalences, but for improving the efficiency of the exploration, some are better than others. To this end, we define a goal function that models the reduction of the search space in such algorithms. We describe two greedy algorithms for the underlying optimization problem. One is based directly on the definition using a straightforward greedy criterion, whereas the second one uses several practical speedups and a different greedy criterion. Finally, we demonstrate the huge impact of exploratory equivalence on a real application, i.e., graph grammar parsing.**

I. Introduction

GRAPHS are an ubiquitous format for structural-data representation and are gaining popularity in various scientific disciplines. They are used to represent diverse types of entities and relations between them in various areas, ranging from chemistry [1], [2], economy [3], politics [4], to popular culture [5]. Such representation enables a more general and global view on the data. Additionally, researchers may benefit from powerful theoretical tools developed in graph theory to extract new insights.

One of the most general problems on various graphs is search for patterns, i.e., finding occurrences of small graphs in larger graphs. In theory, this is known as the subgraph isomorphism problem and has been thoroughly studied, as this is one of the fundamental problems in theoretical computer science. The decision version of this problem is *NP*-complete, and the counting version of the problem is $\#P$-complete. Furthermore, no exponential-time algorithm with a lower bound better than the naive enumeration of pattern is known [6]. This makes the problem intrinsically hard. Despite these pessimistic results, various algorithms exist for finding patterns, a vast majority of them based on the branch-and-bound method (e.g., [7], [8]). In many practical instances, however, these algorithms perform much better than the expected worst-case scenario and are able to solve relatively large instances (e.g., patterns of 1000 vertices in graphs of 10,000 vertices, and even larger).

Despite the practical usability of the current algorithms, there is a large set of problem instances that are often very hard for all the search algorithms. These are graphs with a lot of symmetries, i.e., graphs with many automorphisms. Detecting these symmetries before the start of the search can speed up the algorithm by very large constants, since the search does not have to be repeated for the symmetrical vertices. The goal of this paper is to formally define an equivalence on graph vertices, called *exploratory equivalence*, that captures such symmetries in graphs and can be easily utilized in algorithms for finding patterns (e.g., subgraph isomorphism) in graphs. Since there can be many exploratory equivalences in a graph (and some capture more symmetries than others), we also define the corresponding optimization problem. Our work is based on the ideas already developed by Fürst *et al.* [9] for the purpose of improving the Rekers-Schürr parser [10] for context-sensitive graph grammars. However, while Fürst *et al.* recognized the concept of exploratory equivalence (under the name 'interchangeability'), they did not treat it in a general graph-theoretic and group-theoretic manner. Besides that, they did not consider the possibility of having multiple exploratory equivalences for a single graph, nor did they define the notion of optimal exploratory equivalence. In this paper, we address all of these issues.

Informally, if a group of k vertices in an unlabeled graph belong to the same exploratory equivalence class, then they are interchangeable in the following sense: if each of them were labeled with a unique label, their labels could be arbitrarily interchanged with each other without affecting the graph. The graph would remain isomorphic after any of the $k!$ possible interchanges. It is important to note that a single graph may have multiple exploratory equivalences, i.e., multiple ways of partitioning the graph vertex set into a set of exploratory equivalent classes. Among all possible exploratory equivalences for a given graph, the algorithms proposed in this paper seek the one that captures the largest number of symmetries. As we show later, this is the equivalence with the largest product of the factorials of the cardinalities of its equivalence classes.

Graph grammars [11] are production-based graph rewrite systems and are regarded as a generalization of well-known string-based formal grammars. The Rekers-Schürr parser is an algorithm that, for a given graph and a context-sensitive graph grammar, determines whether the graph belongs to the language generated by the grammar and returns a derivation of the graph in the grammar if this is the case. However, the algorithm may exhibit a heavily exponential behavior when presented with a grammar containing many symmetries. In particular, given a simple grammar for chemical formulas of linear alkanes, the algorithm failed to parse the structural formula of propane within several hours. By exploiting the symmetries in the grammar, the parser's performance is brought down to polynomial for several meaningful classes

978-83-60810-58-3/$25.00 © 2014, IEEE

of grammars [9]. For instance, the parsing of propane now takes less than a second. In general, however, the worst-case performance remains exponential, since the graph grammar parsing problem is *NP*-hard even for highly restricted graph grammar formalisms [12].

Symmetry reduction techniques are not unique to graph-related decision and optimization problems. Liberti [13], for instance, proposed a novel approach to symmetry reduction in branch-and-bound-based MIP (mixed integer programming) solvers. His approach was applied to the discretizable molecular distance problem in the field of organic chemistry [14].

The paper is structured as follows. In the next section, we briefly present definitions and notions used in the rest of the paper. The third section includes the definition of exploratory equivalence, the optimization problem of finding the best exploratory equivalence in a given graph, and an example demonstrating the introduced concepts. We also present the argument that exploratory equivalence does not belong to the class of well-known regular equivalences. The fourth section presents two heuristic algorithms for solving the optimization problem. In Section V, we briefly describe the relevant portion of the Rekers-Schürr parser, its improvement with regard to exploratory equivalence, and some experimental results. Finally, Section VI concludes the paper and gives some ideas for the future work.

II. Preliminaries

Given a (finite) set S, a family $\{P_1, P_2, \ldots, P_s\}$ of nonempty subsets of S is a *partition* of S if every element in S is exactly in one of the subsets, i.e., $P_i \subseteq S$ and $P_i \neq \emptyset$, where $1 \leq i \leq s$, $\bigcup_{1 \leq i \leq s} P_i = S$, and $P_i \cap P_j = \emptyset$ for all $1 \leq i, j \leq s$ where $i \neq j$. When the partition $\{P_1, P_2, \ldots, P_s\}$ is given explicitly, we usually use $\{i \in P_1 \mid i \in P_2 \mid \ldots \mid i \in P_s\}$ as a short form, e.g., $\{\{1,2\},\{3\},\{4\}\}$ is shortened to $\{1,2 \mid 3 \mid 4\}$. In what follows, the order of the sets in a partition is often important. In such cases, we use the form $\langle i \in P_1 \mid i \in P_2 \mid \ldots \mid i \in P_s \rangle$, e.g., $\langle 1,2 \mid 3 \mid 4 \rangle$.

A *group* $\Gamma = (A, \circ)$ with the underlying set A and the binary operation $\circ$ on the elements of A is an algebraic structure satisfying the following conditions: *closure*, i.e., $x \circ y \in A$, *associativity*, i.e., $(x \circ y) \circ z = x \circ (y \circ z)$, *identity element* e, i.e., $\exists e \in A \forall x \in A : e \circ x = x \circ e = x$, and *inverse element*, i.e., $\forall x \in A \exists x^{-1} \in A : x \circ x^{-1} = x^{-1} \circ x = e$.

A *permutation* σ is a bijective function of a finite set S onto itself, i.e., $\sigma \colon S \to S$. Let $\Pi[S]$ denote the set of all permutations of the elements in the set S. Notice that the set $\Pi[S]$ together with the operation of function composition forms a group, which is called the *symmetric group*. Since all the groups discussed in this paper are subgroups of a symmetric group, we write as a group its underlying set only. Additionally, we also define $\Pi[n] = \Pi[\{1, 2, \ldots, n\}]$.

Let Γ be a subgroup of $\Pi[S]$. An element $i \in S$ is called a *fixed point* of the permutation $\sigma \in \Gamma$ if $\sigma(i) = i$. The set of all permutations for which i is a fixed point is a subgroup and is called the *stabilizer subgroup*, i.e.,

$$\mathrm{Stab}_\Gamma(i) = \{\sigma \in \Gamma \mid \sigma(i) = i\}.$$

Notice that all stabilizer subgroups include the identity permutation.

Now let us generalize the definition of a stabilizer from an element to a set. Given $P \subseteq S$, a stabilizer on P is a set of permutations which have a fixed point for all the positions in P:

$$\mathrm{Stab}_\Gamma(P) = \{\sigma \in \Gamma \mid \forall i \in P : \sigma(i) = i\}.$$

Equivalently, $\mathrm{Stab}_\Gamma(P)$ can also be defined in terms of intersections of $\mathrm{Stab}_\Gamma(i)$, where $i \in P$, i.e.,

$$\mathrm{Stab}_\Gamma(P) = \bigcap_{i \in P} \mathrm{Stab}_\Gamma(i).$$

From the latter definition it is clear that $\mathrm{Stab}_\Gamma(P)$ also satisfies all four group conditions. We thus have the following theorem.

Theorem 1: Given a set S, a set $P \subseteq S$, and a subgroup Γ of the group $\Pi[S]$, $\mathrm{Stab}_\Gamma(P)$ is a subgroup of Γ.

We also write $\mathrm{Stab}_\Gamma(P)$ as $\mathrm{Stab}(\Gamma, P)$.

The set of all images of $i \in S$ under permutations of the group Γ is called the *group orbit* of i, i.e.,

$$\mathrm{Orbit}_\Gamma(i) = \{\sigma(i) \mid \sigma \in \Gamma\}.$$

Let $G = (V, E)$ denote a simple undirected graph, where $V = \{1, 2, \ldots, n\}$ is a set of vertices and $E \subseteq V \times V$ is a set of edges. When two graphs are considered, the second is usually denoted with $H = (U, F)$. To denote an edge $(i, j) \in E$, we usually use a shorter version $ij \in E$. A *neighborhood* of a vertex $i \in V$, i.e., a set of vertices adjacent to i, is denoted with $\mathcal{N}(i)$. More formally,

$$\mathcal{N}(i) = \{j \in V \mid ij \in E\}.$$

A *coloration* C of a graph G is an assignment of colors to the vertices V of G, i.e., a surjective function C from V onto $\{1, 2, \ldots, c\}$ for some c, where colors are denoted with integers from 1 to c. Any coloration defines a partition of the vertices V, and vice versa. If $S \subseteq V$, then the *spectrum* of S, denoted $C(S)$, is a set of all colors assigned to the vertices of S. If $S = \{i\}$ is a singleton, then $C(i) = C(S)$ denotes the color assigned to the vertex $i \in V$. A coloration C induces a graph partition $\{C^{-1}(1), C^{-1}(2), \ldots, C^{-1}(c)\}$, and vice versa. A coloration C_1 is *finer or equal* than a coloration C_2 (denoted $C_1 \preceq C_2$) if

$$\forall i, j \in V \colon C_2(i) < C_2(j) \implies C_1(i) < C_1(j).$$

This implies that each set of the C_1-induced partition is a subset of (or equal to) some set of the C_2-induced partition.

A graph *homomorphism* from a graph $G = (V, E)$ to a graph $H = (U, F)$ is a mapping $f \colon V \to U$ such that for each $ij \in E$ it also holds that $f(i)f(j) \in F$. Homomorphism $f \colon V \to U$ is usually denoted with $f \colon G \to H$. We also write $G \to H$ if there exists a homomorphism from G to H. A graph *isomorphism* is a bijective homomorphism, i.e., a mapping $f \colon G \to H$ such that $ij \in E$ if and only if $f(i)f(j) \in F$. We write $G \simeq H$ if there exists an isomorphism from G to H; such graphs G and H are called isomorphic. Since isomorphisms are bijective, every isomorphism also has an inverse. A graph *endomorphism* is a homomorphism whose domain is equal to its codomain, i.e., $f \colon G \to G$.

A graph *automorphism* is both an endomorphism and an isomorphism, i.e., a mapping $f \colon G \to G$ such that $ij \in E$ if

and only if $f(i)f(j) \in E$. Notice that every automorphism is a permutation. If identity is the only automorphism of a graph, we say that the graph is *rigid*. The set of all automorphisms of a graph G is denoted with

$$\mathrm{Aut}(G) = \{a \in \Pi[n] \mid G \simeq a(G)\}$$

and is called the *automorphism group* of a graph G. Constructing $\mathrm{Aut}(G)$ is at least as difficult as solving the graph isomorphism problem, since graphs G and H are isomorphic if and only if the disconnected graph formed by the disjoint union of G and H has an automorphism that swaps the two components. Several practical algorithms are known for finding $\mathrm{Aut}(G)$; the most well-known is probably NAUTY [15].

III. PROBLEM DESCRIPTION

As already mentioned in the introduction, our goal is to find equivalent (also called indistinguishable) vertices of a graph. There are many types of equivalences already discussed in the literature. We give several examples later in this section. Our definition of equivalence is associated with the algorithmic exploration of a graph; for example, when the task is to find a pattern graph that is a subgraph in another target graph. In particular, branch-and-bound search algorithms could exploit such equivalences by reducing the number of (partial) matches established between a set of equivalent vertices in the pattern graph and a corresponding set of vertices in the target graph. In the remainder of this section, we formally describe our type of equivalence and the problem of finding the corresponding equivalence classes. Additionally, we also discuss several other similar equivalences and argue that our type is novel.

First, let us define a few additional notions. Let S be a set, and let $P \subseteq S$ be a set of positions. We say that a permutation $\sigma_1 \in \Pi[P]$ is *covered* by a permutation $\sigma_2 \in \Pi[S]$ if the two permutations have the same image on the positions P, i.e.,

$$\sigma_1 \preceq \sigma_2 \equiv \forall i \in P\colon \sigma_1(i) = \sigma_2(i).$$

Observe that P is equal to the domain of σ_1.

Now let $A \subseteq \Pi[S]$. We say that a set A of permutations *covers* a set P of positions if every permutation of P is covered by a permutation in A. More formally,

$$\mathrm{cover}(A, P) \equiv \forall \sigma \in \Pi[P]\ \exists a \in A : \sigma \preceq a.$$

Given a graph $G = (V, E)$, we say that a partition $\{P_1, P_2, \ldots, P_s\}$ of V is *exploratory equivalent* if for all $1 \leq i \leq s$ the following two conditions hold:

$$\mathrm{cover}(A_{i-1}, P_i) \text{ and } A_i = \mathrm{Stab}(A_{i-1}, P_i), \tag{1}$$

where $A_0 = \mathrm{Aut}(G)$. The sets $P_1, P_2, \ldots, P_s$ are the equivalence classes. Notice that the order of classes regarding the partition $\{P_1, P_2, \ldots, P_s\}$ is irrelevant, but it is important when checking the conditions (1), since not all orders of $P_1, P_2, \ldots, P_s$ satisfy them. In this sense the exploratory equivalence is an algorithmic concept. In particular, an algorithm processing a vertex $u \in P_i$ may ignore all other vertices in P_i, since the automorphisms A_{i-1} cover all permutations of P_i. However, it is important to observe that equivalence classes are not independent. For example, when a vertex $u \in P_i$ is processed, this may influence the rest of the algorithm. Therefore, when determining the next class P_{i+1}, one must exclude the automorphisms corresponding to the already processed classes $P_1, P_2, \ldots, P_i$, which is the same as restricting to the automorphisms where the positions $P_1 \cup P_2 \cup \cdots \cup P_i$ are fixed points. That is the reason why in each step the automorphism group is restricted from A_{i-1} to $A_i = \mathrm{Stab}(A_{i-1}, P_i)$.

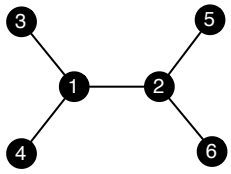

Fig. 1. An example graph with several exploratory equivalences.

Let us demonstrate the introduced concepts with an example. Consider the 6-vertex graph of Fig. 1. Its automorphism group consists of the following eight permutations (written in the one-line notation):

$$123456, 123465, 124356, 124365, \\ 215634, 215643, 216534, 216543. \tag{2}$$

There are twelve exploratory equivalent partitions of the graph. They are given in the form of a Hasse diagram (using the refinement relation $\preceq$ between two partitions) in Fig. 2. The

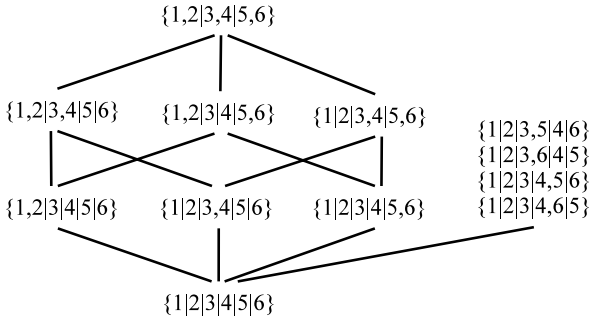

Fig. 2. Hasse diagram of all the exploratory equivalent partitions of the graph from Fig. 1. (The four partitions on the right-hand side are actually four separate vertices in the diagram.)

trivial partition ($\{1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6\}$ in the case of the graph of Fig. 1) is always exploratory equivalent. For the trivial partition, any ordering of its constituent sets satisfies the conditions (1). By contrast, for the exploratory equivalent partition $\{1, 2 \mid 3, 4 \mid 5, 6\}$, only the orderings $\langle 3, 4 \mid 5, 6 \mid 1, 2\rangle$ and $\langle 5, 6 \mid 3, 4 \mid 1, 2\rangle$ satisfy those conditions.

Corollary 1: Given a graph and its partition $\{P_1, P_2, \ldots, P_s\}$, let $A_0 = \mathrm{Aut}(G)$ and $A_i = \mathrm{Stab}(A_{i-1}, P_i)$ for $1 \leq i \leq s$. Then each A_i, where $1 \leq i \leq s$, is a subgroup.

Proof: $\mathrm{Aut}(G)$ is a group. By repeatedly applying Theorem 1, we know that A_i is a subgroup of A_{i-1}, for all $1 \leq i \leq s$. ∎

Now we are ready to define the problem. The input of the problem is a graph $G = (V, E)$ and its automorphism group $\mathrm{Aut}(G)$, and the goal of the problem is to find an exploratory equivalent partition $\{P_1, P_2, \ldots, P_s\}$ of V that maximizes the

product

$$\prod_{i=1}^{s} |P_i|!.$$

The reason for using the product of factorials in the objective function is that each class P_i covers $|P_i|!$ automorphic graphs, and the total number of automorphic graphs covered is thus the product above. In the following sections, we denote the problem with MAXEXPLOREQ.

In the paper [16], a large class of the so-called regular equivalences (called colorations therein) is surveyed. A coloration of a graph is *regular* when the equality of the spectra of two vertices implies the equality of the spectra of the corresponding neighborhoods. More formally, a coloration C of graph G is *regular* if and only if for all $i, j \in V$

$$C(i) = C(j) \implies C(\mathcal{N}(i)) = C(\mathcal{N}(j)).$$

Many different types of colorations are regular, e.g., strong and weak structural coloration, orbit coloration, perfect coloration, and exact coloration. See [16] for details. For example, coloring each orbit of $\mathrm{Aut}(G)$ gives orbit coloration. However, as it turns out, exploratory equivalence is not regular. To demonstrate this, consider again the graph from Fig. 1 and its exploratory equivalent partition $\{1, 2 \mid 3, 4 \mid 5, 6\}$, where the color of each class is different. It is easy to see that it is not regular, since $C(1) = C(2)$ but $C(\{3, 4\})$ is not equal to $C(\{6, 7\})$.

IV. ALGORITHM DESCRIPTION

In this section, we will describe two greedy algorithms for the MAXEXPLOREQ problem. The first algorithm is based on restricting the set of automorphisms to the stabilizer of the equivalent vertices found in one iteration. The second algorithm is more time-efficient owing to a faster detection of equivalent sets.

A. Greedy algorithm based on stabilizer restrictions

The first algorithm for the optimization problem MAXEXPLOREQ is based on the definition and will represent a reference algorithm that can be further improved. The idea of the algorithm is to start with the initial automorphism group, find one equivalence class of the partition, reduce the set of automorphisms only to the stabilizer of A, and recursively find new equivalence classes until the entire set of vertices is contained in the equivalence.

The input to this problem is the set of automorphisms (permutations) A and a set $V' \subseteq V$ of vertices not yet included in any equivalence class; initially V' is the entire set V.

If the set of automorphisms contains only the identity, then each vertex in V' represents a different equivalence class (i.e., no new indistinguishable vertices exist in the graph). If there is more than one automorphism in A, then at least two vertices are indistinguishable. At this point, the goal of the algorithm is to find a subset $S \subseteq V'$ that is covered by A. Usually, however, there are many possibilities for S, and different choices can lead to very different final solutions. The greedy criterion for this choice is the size of S, i.e., among many possibilities, the largest set S is chosen. When there are more sets with the same size, the algorithm chooses the one that has the largest stabilizer in A. The described algorithm is shown in more detail as Algorithm 1.

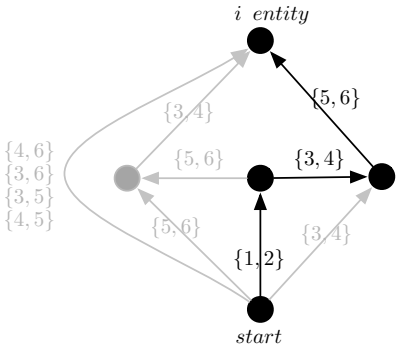

Fig. 3. The search space of Algorithm 1 for the graph in Fig. 1

Algorithm 1 Greedy algorithm for MAXEXPLOREQ based on stabilizer restrictions.

```
1:  function GREEDY1(A, V')
2:      if |A| = 1 then return singletons(V')
3:      bestP = ∅
4:      bestA = ∅
5:      for all P: P ⊆ V' ∧ cover(A, P) do
6:          A' ← Stab(A, P)
7:          if |P| > |bestP| ∨
8:             |P| = |bestP| ∧ |A'| > |bestA| then
9:              bestP ← P
10:             bestA ← A'
11:     return {bestP} ∪ GREEDY1(bestA, V' \ bestP)
```

To make this algorithm a little more clear, we will show its trace on the simple example graph of Fig. 1. The initial set of all automorphisms A is already shown in equation (2). From this set, the algorithm finds the equivalence class $\{1, 2\}$ and reduces A to the set $\mathrm{Stab}(A, \{1, 2\})$, which is:

$$A' = \{123456, 123465, 124356, 124365\}$$

In this automorphism group, it finds the equivalence class $\{3, 4\}$ and reduces the automorphisms to the stabilizer:

$$A' = \{123456, 123465\}$$

The final equivalence class from this group is $\{5, 6\}$, and the corresponding stabilizer contains only the identity. This yields the final result, namely the partition $\{1, 2 \mid 3, 4 \mid 5, 6\}$. If, at the moment when A' contained only the identity, the current partition did not include all six vertices of the graph, each of the missing vertices would be added as a singleton set to the equivalence. The entire search space for this example is shown in Fig. 3. Each vertex in this graph represents an automorphism group. The bottom vertex is the set of all automorphisms, and the top vertex is the set containing only the identity. Each edge represents a stabilization with the set that is written as the label of the edge. The bold vertices and edges are the ones that our algorithm follows.

Now we will discuss the correctness of the described algorithm.

Fig. 4. Two graphs on which Algorithm 1 returns a suboptimal solution. The left graph is the smallest counterexample in terms of the number of vertices, and the right one is the smallest counterexample in terms of the number of edges.

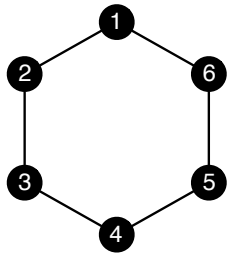

Fig. 5. A sample graph

Theorem 2: Algorithm 1 returns a partition of exploratory equivalent vertices.

Proof: Since the algorithm closely follows the definition, the proof is trivial. Each partition is covered by the automorphism group; the loop only iterates over the subsets that are covered. The second criterion from the definition is guaranteed by the recursion, since the set of automorphisms used in the recursion is only the stabilizer of the equivalence class found in the previous step. ■

Another question we need to address is the optimality of this algorithm. Unfortunately, the greedy criterion does not guarantee the optimality of the solution. We will demonstrate this by two examples shown in Fig. 4. These two examples were found by the exhaustive enumeration of all non-isomorphic connected graphs (starting with the smallest graph), and the graphs of Fig. 4 are the smallest examples where Algorithm 1 does not find an optimal solution. The optimal solution for the left graph in Fig. 4 is one with value 8 (partition $\{1,4 \mid 2,5 \mid 3,6\}$), whereas the algorithm returns a solution with value 6 (partition $\{1,3,5 \mid 2 \mid 4 \mid 6\}$). A similar situation occurs with the right graph, where the optimal solution is 8 (partition $\{1 \mid 2,3 \mid 4,5 \mid 6,7\}$), but the algorithm returns a suboptimal solution with value 6 (partition $\{2,4,6 \mid 1 \mid 3 \mid 5 \mid 7\}$).

Because of the exhaustive search over all subsets of V', the described algorithm is not very practical for larger graphs. In the next subsection, we will describe a more efficient algorithm that utilizes an incremental procedure to find the possible equivalence classes.

B. Greedy algorithm based on positional restriction of automorphisms

For a more convenient presentation of our second greedy algorithm, let us define a few auxiliary terms. The *positional restriction* of an automorphism (permutation) $a \in \Pi[S]$ to a set $R \subseteq S$ (denoted $\rho(a,R)$) is a partial function $a' \colon S \to S$ with $a'(i) = a(i)$ for all $i \in R$ and $a'(i)$ being undefined for all $i \in S \setminus R$. For example, $\rho((3,2,1,4),\{2,4\}) = (\uparrow,2,\uparrow,4)$. We use the one-line notation for representing automorphisms ($(1,2,3,4) \equiv 1234$) and the symbol $\uparrow$ for indicating the undefined values. Therefore, $a = (\uparrow,2,\uparrow,4)$ represents the fact that both $a(1)$ and $a(3)$ are undefined, whereas $a(2) = 2$ and $a(4) = 4$.

The positional restriction of a set of automorphisms $A \subseteq \Pi[S]$ to a set $R \subseteq S$ (denoted $\rho(A,R)$) is a set $\{\rho(a,R) \mid a \in A\}$. For example, $\rho(\{(1,2,3,4),(3,2,1,4)\}, \{2,4\}) = \{(\uparrow,2,\uparrow,4)\}$. As illustrated by this example, several automorphisms may collapse into one as a side-effect of a positional restriction.

For a given set S and a given set of (positionally unrestricted or restricted) set of automorphisms $A \subseteq \Pi[S]$, a *permofix* is a pair (P,F) such that the following conditions hold: (1) $P \subseteq S$, (2) $F \subseteq S$, (3) $P \cap F = \emptyset$, and (4) for each permutation $\sigma \in \Pi[P]$ there exists an automorphism $a \in A$ such that $a(i) = \sigma(i)$ for all $i \in P$ and $a(i) = i$ for all $i \in F$. In other words, a pair (P,F) is a permofix if there exists a set of automorphisms $A' \subseteq A$ that covers the set P (i.e., all permutations of P) and simultaneously fixes all elements of F. Given a permofix (P,F), the sets P and F will be called the *perm-set* and the *fix-set*, respectively. A k-permofix is a permofix (P,F) with $|P| = k$. The *potential* of a permofix (P,F) is the product $|P|!\,|F|!$. A permofix (P',F') is *contained* in a permofix (P,F) (denoted $(P',F') \sqsubseteq (P,F)$) if $P' \subset P$ (a *strict* subset) or $P' = P$ and $F' \subseteq F$.

As an example, consider the graph of Fig. 5. The 12 automorphisms of this graph are as follows:

$$\begin{aligned} a_1 &= (1,2,3,4,5,6) \\ a_2 &= (2,3,4,5,6,1) \\ a_3 &= (3,4,5,6,1,2) \\ a_4 &= (4,5,6,1,2,3) \\ a_5 &= (5,6,1,2,3,4) \\ a_6 &= (6,1,2,3,4,5) \\ a_7 &= (1,6,5,4,3,2) \\ a_8 &= (2,1,6,5,4,3) \\ a_9 &= (3,2,1,6,5,4) \\ a_{10} &= (4,3,2,1,6,5) \\ a_{11} &= (5,4,3,2,1,6) \\ a_{12} &= (6,5,4,3,2,1) \end{aligned} \tag{3}$$

For this graph, the pair $(\{1,3\},\{2,5\})$ is a permofix, since the automorphisms a_1 and a_9 cover both permutations of the set $\{1,3\}$ while fixing the elements 2 and 5. The pair $(\{1,3,5\},\emptyset)$ is a permofix as well, since the automorphisms a_1, a_3, a_5, a_7, a_9, and a_{11} collectively cover all permutations of the set $\{1,3,5\}$. We also have $(\{1,3\},\{2,5\}) \sqsubseteq (\{1,3,5\},\emptyset)$.

Given the set of automorphisms $A \subseteq \Pi[n]$ of a n-vertex graph, the algorithm works as a greedy iterative process. In each iteration, it produces the set of all permofixes in the current set of automorphisms (in the first iteration, this is the unrestricted set A) and greedily selects a permofix with the

largest potential. After making its selection, the algorithm positionally restricts all automorphisms to the fix-set of the selected permofix. The positionally restricted set of automorphisms serves as the input to the next iteration. The process stops once all automorphisms have become completely undefined functions. The output of the algorithm is a set composed of all perm-sets of the permofixes selected in individual iterations and of the singletons containing the individual vertices that are not present in any of the selected perm-sets. Later, we shall show that the algorithm's output is an exploratory equivalent partition of the vertex set.

The rationale for selecting a permofix with the highest potential is based on the following heuristics: Recall that the algorithm's goal is to find a partition $\{P_1, \ldots, P_s\}$ of $\{1, \ldots, n\}$ with a maximum value of $|P_1|! \ldots |P_s|!$. A permofix (P, F) is guaranteed to contribute at least a factor of $|P|!$ to the target product $|P_1|! \ldots |P_s|!$ (since the perm-set of the selected permofix is part of the algorithm's output), but it can potentially contribute up to $|P|!\,|F|!$. The optimal scenario takes place when the entire fix-set F serves as a perm-set of some permofix selected later in the process. Therefore, a permofix (P, F) having the largest value of $|P|!\,|F|!$ may potentially contribute the largest factor to the target product.

The pseudocode of the greedy algorithm based on positional restrictions of the automorphism set is shown as Algorithm 2.

To show that the output produced by the algorithm conforms to our problem definition, we shall first prove the following lemma:

Lemma 1: Each element of the set returned by the procedure GREEDY2 is a perm-set of the input set A of automorphisms.

Proof: The singletons are perm-sets by definition, so let us focus on the elements of the set $\mathcal{P}$ inside the procedure GREEDY2. In each iteration, the algorithm first applies the procedure FIND2PERMOFIXES to the current set of automorphisms A. This procedure returns a set of all pairs $(\{p, q\}, \{r_1, \ldots, r_t\})$ such that there exists an automorphism a with $a(p) = q$, $a(q) = p$, and $a(r_1) = r_1$, …, $a(r_t) = r_t$. By the definition of automorphism group, the set A always contains the identity automorphism a_{id} with the property $a_{\mathrm{id}}(p) = p$, $a_{\mathrm{id}}(q) = q$, and $a_{\mathrm{id}}(r_1) = r_1$, …, $a(r_t) = r_t$. The automorphisms a and a_{id} jointly form a proof that the pair $(\{p, q\}, \{r_1, \ldots, r_t\})$ is indeed a permofix.

The procedure EXTEND iteratively produces k-permofixes based on sets of $(k-1)$-permofixes in the set of automorphisms A. For $k = 3$, the procedure creates a pair $PF = (\{p, q, r\}, F_1 \cap F_2 \cap F_3)$ from the permofixes $PF_1 = (\{p, q\}, \{r\} \cup F_1)$, $PF_2 = (\{p, r\}, \{q\} \cup F_2)$, and $PF_3 = (\{q, r\}, \{p\} \cup F_3)$. Neglecting the sets F_1, F_2, and F_3 for the time being, the permofix PF_1 represents the permutation $(p\ q)(r)$ in the cycle notation. Likewise, PF_2 and PF_3 represent the permutations $(p\,r)(q)$ and $(q\,r)(p)$, respectively. Since $(A, \circ)$ is a group, the permutation $(p\,q)(r) \circ (p\,r)(q) \circ (q\,r)(p) = (p\,q\,r)$ has to be completely present in A; in other words, A has to contain an automorphism for each of the 3! permutations of the set $\{p, q, r\}$. Therefore, $\{p, q, r\}$ is a perm-set in A. The fix-set corresponding to this perm-set is (a superset of) the intersection of the fix-sets of PF_1, PF_2, and

Algorithm 2 Greedy algorithm based on positional restrictions

```
1:  function GREEDY2(A, V)
2:      // A: a set of automorphisms, V = {1, ..., n}
3:      P := ∅;
4:      W ::= V;
5:      while A contains at least one valid element do
6:          R := CLEANUP(FIND2PERMOFIXES(A));
7:          k := 3;
8:          repeat
9:              R' := R;
10:             R := CLEANUP(EXTEND(R, k));
11:             k := k + 1
12:         until R' = R;
13:         (P_m, F_m) := highest-potential permofix in R;
14:         P := P ∪ {P_m};
15:         W := W \ P_m;
16:         A := ρ(A, F_m)
17:     return P ∪ singletons(W)
18:
19: function FIND2PERMOFIXES(A)
20:     R := ∅;
21:     for all a ∈ A do
22:         for all (i, j): i ≠ j ∧ a(i) = j ∧ a(j) = i do
23:             P := {i, j};
24:             F := {k | a(k) = k};
25:             R := R ∪ {(P, F)}
26:     return R
27:
28: function EXTEND(R, k)
29:     for all P: P ⊆ {1, ..., n} ∧ |P| = k do
30:         F := {1, ..., n};
31:         i := 0;
32:         for all p ∈ P do
33:             if ∃F': (P \ {p}, {p} ∪ F') ∈ R then
34:                 F := F ∩ F';
35:                 i := i + 1
36:             else
37:                 break
38:         if i = k then
39:             R := R ∪ {(P, F)}
40:     return R
41:
42: function CLEANUP(R)
43:     for all (P, F) ∈ R do
44:         for all (P', F') ∈ R \ {(P, F)} do
45:             if (P', F') ⊑ (P, F) then
46:                 R := R \ {(P', F')}
47:     return R
```

PF_3. Consequently, PF is a permofix in A. This reasoning can be straightforwardly extended to the general case of $k > 3$. Therefore, every pair created by the procedure EXTENDS is a permofix in the current set of automorphisms.

The procedure CLEANUP does not produce anything new; it merely reduces the number of permofixes. For a permofix (P, F), all permofixes (P', F') with $(P', F') \sqsubseteq (P, F)$ are heuristically pronounced redundant. If $P' = P$ and $F' \subseteq F$, the permofix (P', F') is clearly superfluous. If $P' \subset P$, then the permofix (P, F) has been created from (P', F') within the EXTEND procedure.

The positional restriction can only reduce the set of permofixes. It is easy to see that if a pair (P, F) is a permofix in a positionally restricted set of automorphisms, then it is a permofix in the original set, too.

In summary, the set $\mathcal{R}$ consists of permofixes of the initial set of automorphisms A, and every element of the set returned from the procedure GREEDY2 is a perm-set of A. ■

In the following theorem, we show that the algorithm produces a solution to our problem, i.e., an exploratory equivalent partition of the vertex set.

Theorem 3: The procedure GREEDY2 returns an exploratory equivalent partition of the vertex set V.

Proof: Let $\{P_1, \ldots, P_s, \{i_1\}, \ldots, \{i_r\}\}$ be the result of the algorithm GREEDY2, where P_1, …, P_s are the perm-sets produced in individual iterations, and $\{i_1\}, \ldots, \{i_r\}$ are the singletons created from the vertices that do not belong to the set $P_1 \cup \ldots \cup P_s$. By construction, the elements of the output set are mutually disjoint sets that collectively cover the entire vertex set. The output set is thus a partition of the vertex set.

By definition, each of the produced perm-sets P_1, …, P_s is covered by the initial set of automorphisms $A_0 \equiv A$, i.e., we have cover(A_0, P_i) for all $i \in \{1, \ldots, s\}$. Let us now show that cover(Stab(A_0, P_s), P_{s-1}) also holds. The perm-set P_s has to be a subset of the fix-set F_{s-1}; otherwise, the algorithm would, at some earlier stage, have set $a_1(j) := \uparrow$, …, $a_{|A|}(j) := \uparrow$ for at least one $j \in P_s$ and hence could not produce P_s. By the definition of permofix, there exists a set of automorphisms that fixes F_{s-1} and simultaneously covers P_{s-1}. Since $P_s \subseteq F_{s-1}$, the same set of automorphisms also fixes P_s. Consequently, the set of automorphisms where P_s is fixed (i.e., Stab(A_0, P_s)) covers P_{s-1}. In the same manner, we can prove cover(Stab(Stab$(A_0, P_s), P_{s-1}), P_{s-2}$), etc. Therefore, the perm-sets P_s, P_{s-1}, P_{s-2}, …, P_1, together with the singleton sets formed by the missing elements, constitute an exploratory equivalent partition of the vertex set V. ■

In practice, the algorithm GREEDY2 is more efficient than GREEDY1. For each combination P of the current set of vertices, the first greedy algorithm checks whether P is covered by the current set of automorphisms (in other words, whether P is a perm-set in the current set of automorphisms). By contrast, the algorithm GREEDY2 generates candidate perm-sets (and the associated fix-sets) in an incremental fashion: a perm-set with k elements is generated by merging k perm-sets with $k - 1$ elements. If no k-element perm-sets are generated, the algorithm will not attempt to generate any $(k + 1)$-element perm-sets.

Let us illustrate the algorithm GREEDY2 with two examples. Consider the graph of Fig. 5. Given the set of its automorphisms as input (enumerated in Eq. 3), the algorithm produces the following 2-permofixes (after executing the procedure CLEANUP):

$$
\begin{array}{lll}
(\{1,2\},\emptyset) & (\{2,3\},\emptyset) & (\{3,6\},\emptyset) \\
(\{1,4\},\emptyset) & (\{2,5\},\emptyset) & (\{4,5\},\emptyset) \\
(\{1,6\},\emptyset) & (\{3,4\},\emptyset) & (\{5,6\},\emptyset) \\
(\{1,3\},\{2,5\}) & (\{1,5\},\{3,6\}) & (\{2,4\},\{3,6\}) \\
(\{2,6\},\{1,4\}) & (\{3,5\},\{1,4\}) & (\{4,6\},\{2,5\})
\end{array}
$$

The procedure EXTEND produces two 3-permofixes: $(\{1,3,5\},\emptyset)$ and $(\{2,4,6\},\emptyset)$. The procedure CLEANUP subsequently removes all permofixes (P, F) with $|P| = |F| = 2$. In the next step, the algorithm selects a permofix with the highest value of $|P|!\,|F|!$. This is either $(\{1,3,5\},\emptyset)$ or $(\{2,4,6\},\emptyset)$. In either case, the fix-set is empty, so the procedure RESTRICT sets all elements of all automorphisms to $\uparrow$. As a result, the algorithm immediately stops with the result $\{1,3,5 \mid 2 \mid 4 \mid 6\}$ (or $\{2,4,6 \mid 1 \mid 3 \mid 5\}$, depending on its selection). Among all exploratory equivalent partitions, these two both have the highest product of the factorials of the cardinalities of their constituent sets and hence represent two optimal solutions to the MAXEXPLOREQ problem.

The graph of Fig. 1 has 8 automorphisms:

$$
\begin{aligned}
a_1 &= (1,2,3,4,5,6) \\
a_2 &= (1,2,3,4,6,5) \\
a_3 &= (1,2,4,3,5,6) \\
a_4 &= (1,2,4,3,6,5) \\
a_5 &= (2,1,5,6,3,4) \\
a_6 &= (2,1,5,6,4,3) \\
a_7 &= (2,1,6,5,3,4) \\
a_8 &= (2,1,6,5,4,3)
\end{aligned}
$$

In the first iteration, the algorithm produces the following permofixes:

$$
\begin{array}{ll}
(\{3,5\},\emptyset) & (\{1,2\},\emptyset) \\
(\{3,6\},\emptyset) & (\{3,4\},\{1,2,5,6\}) \\
(\{4,5\},\emptyset) & (\{5,6\},\{1,2,3,4\}) \\
(\{4,6\},\emptyset) &
\end{array}
$$

The set of automorphisms contains no permofixes (P, F) with $|P| > 2$. Using the highest-potential criterion, the algorithm selects either the permofix $(\{3,4\},\{1,2,5,6\})$ or the permofix $(\{5,6\},\{1,2,3,4\})$. Let us assume that the former is selected; the latter permofix leads to the same output. After the selection, the set of automorphisms is positionally restricted with respect to the fix-set $\{1,2,5,6\}$):

$$
\begin{aligned}
a'_1 &= (1,2,\uparrow,\uparrow,5,6) \\
a'_2 &= (1,2,\uparrow,\uparrow,6,5) \\
a'_5 &= (2,1,\uparrow,\uparrow,3,4) \\
a'_6 &= (2,1,\uparrow,\uparrow,4,3)
\end{aligned}
$$

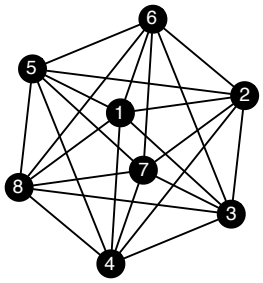

Fig. 6. The smallest graph on which Algorithm 2 returns a suboptimal solution.

The automorphisms a'_3 and a'_4 are equal to a'_1 and a_2', respectively, and an analogous situation occurs with the automorhpisms a'_7 and a'_8. In the second iteration, only two permofixes are produced: $(\{1,2\},\emptyset)$ and $(\{5,6\},\{1,2\})$. The latter has a greater potential than the former and is hence selected, restricting the set of automorphisms to $\{(1,2,\uparrow,\uparrow,\uparrow,\uparrow), (2,1,\uparrow,\uparrow,\uparrow,\uparrow)\}$. The restricted automorphisms give rise to the sole permofix $(\{1,2\},\emptyset)$, which is selected in the third iteration of the algorithm. The algorithm thus outputs the partition $\{3,4 \mid 5,6 \mid 1,2\}$, which is again an optimal solution to the MAXEXPLOREQ problem.

For a vast majority of input graphs, the algorithm GREEDY2 produces optimal exploratory equivalent partitions. The smallest graph (in terms of vertex count) with a suboptimal result is shown in Fig. 6. For this graph, the algorithm produces the partition $\{1,7 \mid 2,8 \mid 3,5 \mid 4,6\}$ with the target cardinality factorial product being $2!\,2!\,2!\,2! = 16$. The optimal solution, however, is the partition $\{1,2,3,4 \mid 5 \mid 6 \mid 7 \mid 8\}$ with the target product of $4! = 24$. In the first iteration, the algorithm produces 20 permofixes, two of which are $(\{1,2,3,4\},\ \emptyset)$ and $(\{1,7\},\{2,3,4,5,6,8\})$. The former permofix would lead to an optimal solution, but the algorithm chooses the latter, since $2!\,6! > 4!$. However, the fix-set of the selected permofix eventually contributes only $2!\,2!\,2!$ instead of the potential $6!$ to the target product, making the algorithm's first-iteration choice suboptimal.

Interestingly, the graphs of Fig. 4 are not counterexamples for the second greedy algorithm, and the graph of Fig. 6 is not a counterexample for the first algorithm. In contrast to the algorithm GREEDY1, the algorithm GREEDY2 considers the combined sizes of individual perm-sets and fix-sets when making greedy selections. In the right graph of Fig. 4, for example, the algorithm GREEDY2 has to choose between the permofix $(\{2,3\},\{1,4,5,6,7\})$ (or an equivalent permofix with potential $2!\,5!$) and the permofix $(\{2,4,6\},\emptyset)$ (or an equivalent permofix with potential $3!$). The first permofix is obviously preferable, leading to an optimal partition. Conversely, since the algorithm GREEDY1 considers perm-sets without the associated fix-sets, it prefers the perm-set $\{1,2,3,4\}$ over all 2-element perm-sets (regardless of the sizes of their associated fix-sets) when dealing with the graph of Fig. 6.

V. EXPLORATORY EQUIVALENCE AND THE IMPROVED REKERS-SCHÜRR PARSER

As we mentioned in the introduction, the concept of exploratory equivalence was developed by Fürst *et al.* [9] for the purpose of improving the Rekers-Schürr graph grammar parser [10], although the authors did not provide a rigorous graph-theoretic and group-theoretic definition of exploratory equivalence and did not consider the possibility of multiple exploratory equivalent partitions for a single graph. In this section, we show how a proper consideration of exploratory equivalence may lead to immense performance gains when parsing graphs against graph grammars.

The Rekers-Schürr graph grammar parser (both the original and the improved version) accepts a graph and a context-sensitive graph grammar on its input. A context-sensitive graph grammar (called just 'grammar' in the sequel) is a quadruple $(\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{A})$, where $\mathcal{N}$ is a set of *nonterminal* labels, $\mathcal{T}$ is a set of *terminal* labels, $\mathcal{P}$ is a set of *productions*, and $\mathcal{A}$ is a set of *axioms*. Each production p is a rule of the form $Lhs[p] ::= Rhs[p]$, where $Lhs[p]$ (the left-hand side – LHS) and $Rhs[p]$ (the right-hand side – RHS) are subgraphs of a graph $Union[p]$ whose elements (vertices and edges) have labels from $\mathcal{N} \cup \mathcal{T}$. The graph $Common[p] = Lhs[p] \cap Rhs[p]$ is called the *context* of the production. Let $Xlhs[p] = Lhs[p] \setminus Common[p]$ and $Xrhs[p] = Rhs[p] \setminus Common[p]$; note that $Xlhs[p]$ and $Xrhs[p]$ might not be proper graphs, since they may contain dangling edges. A sample production, as well as the graphs and the graph element sets associated with it, is shown in Fig. 7. In contrast to the graph depictions shown so far, the inscriptions inside the vertices represent vertex labels rather than vertex indices. The indices are displayed next to individual vertices. The yellow-colored vertices belong to the graph $Common[p]$ and hence to both the LHS and RHS simultaneously; this is also reflected in the fact that such vertices have the same index on both sides of the production.

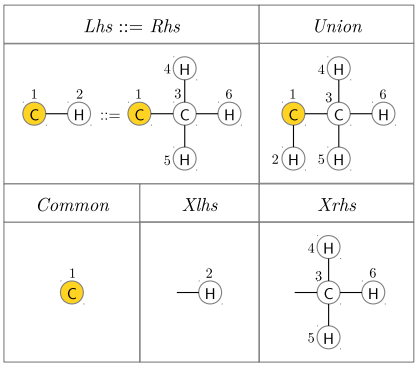

Fig. 7. A sample production and the associated graphs and graph element sets.

An *l-homomorphism* $h\colon Lhs[p] \to G$ for a production p is a graph homomorphism whose restriction to $Xlhs[p]$ is injective. An *l-occurrence* of a production p in a graph G is a graph $L' \sqsubseteq G$ such that $L' = h(Lhs[p])$ for some l-homomorphism h. The terms *r-homomorphism* and *r-occurrence* are defined symmetrically (with $Rhs[p]$ and $Xrhs[p]$ instead of $Lhs[p]$ and $Xlhs[p]$, respectively).

To *apply* a production p to a graph G, the following three steps are performed: (1) find an l-occurrence of p in G (let $h\colon Lhs[p] \to G$ be the associated l-homomorphism); (2) remove the elements $h(Xlhs[p])$ from the graph G; (3) attach fresh copies of the elements $Xrhs[p]$ to the elements

$h(Common[p])$ in the same way as the elements $Xrhs[p]$ are attached to the elements of $Common[p]$ within the graph $Rhs[p]$. A *derivation* of a graph G in a graph grammar is a sequence of production applications beginning with an axiom graph and ending with the graph G. The *language* of a graph grammar GG is the set of all terminally labeled graphs that have a derivation in GG. (A graph is *terminally labeled* if all of its elements are labeled by labels from the set $\mathcal{T}$.) A *parser* is an algorithm that, for a given graph G and a given graph grammar GG, determines whether G belongs to the language of GG and produces a derivation of G in GG if this is the case. Figure 8 shows a grammar for generating the structural formulas of linear alkanes. All graph labels belong to the set $\mathcal{T}$, including the 'non-label' — a fictitious label for unlabeled edges. Figure 9 displays the derivation of the propane graph in that grammar. The derivation starts with the axiom (the methane graph) and passes through the ethane graph.

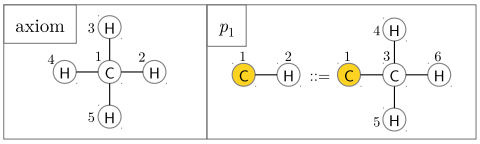

Fig. 8. A grammar for generating the structural formulas of linear alkanes.

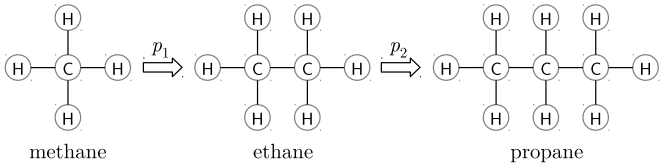

Fig. 9. Derivation of the propane graph in the grammar of Fig. 8.

The Rekers-Schürr parser works as a two-stage process. In the first stage, the input graph G is analyzed in order to obtain a partially ordered redundant set $\mathcal{S}$ of candidate production applications that might take part in a potential derivation of G. In the second stage, the parser tries to find, using backtracking if necessary, a sequence of production applications within the set $\mathcal{S}$ that constitutes a correct derivation of the graph G. The improvement by Fürst *et al.* pertains only to the first stage of the parsing process.

At the beginning of the first stage, the parser creates a graph $\overline{G}$ as a copy of the input graph G. After that, it iteratively searches the graph $\overline{G}$ for all r-occurrences of individual productions. For each discovered r-occurrence of a production p, the graph $\overline{G}$ is augmented by attaching fresh copies of the elements $Xlhs[p]$ to the r-occurrence, giving rise to a *production instance* — a homomorphic image of the entire production p that defines a candidate application of p in a potential derivation of G. The augmentation of the graph $\overline{G}$ might result in new r-occurrences among the added elements. The discover-and-augment cycle finishes once all r-occurrences of all productions have been discovered.

To guarantee the discovery of all r-occurrences, each RHS has to be matched against the graph $\overline{G}$ in all possible ways. In other words, all RHS-to-$\overline{G}$ r-homomorphisms have to be established, including different r-homomorphisms between a production and *each* of its r-occurrences. However, exploratory equivalence can make some (or all) of the r-homomorphisms between a production and its r-occurrence redundant. Let us assume that a production p contains k distinct vertices $v_1, \ldots, v_k$ with the following properties:

- either $v_1, \ldots, v_k \in Xrhs[p]$ or $v_1, \ldots, v_k \in Common[p]$;
- $v_1, \ldots, v_k$ constitute an equivalence class in at least one exploratory equivalent partition of the graph $Rhs[p]$;
- $v_1, \ldots, v_k$ constitute an equivalence class in at least one explorationally equivalent partition of the graph $Union[p]$.

Then it can be shown [9] that the set of r-homomorphisms $h\colon Rhs[p] \to \overline{G}$ established between the production p and the graph $\overline{G}$ can be safely restricted to those r-homomorphisms h for which $index(h(v_1)) < \ldots < index(h(v_k))$, where $index(v)$ is a unique index assigned to a vertex $x \in \overline{G}$. This rule reduces the number of established p-homomorphisms between the production p and each of its occurrences by a factor of $k!$. Since each discovered r-homomorphism is followed by an augmentation of the graph $\overline{G}$, immense performance gains can thus be attained. This rule can be straightforwardly extended to multiple non-singleton classes of an exploratory equivalent partition.

Consider the grammar of Fig. 8. The optimal exploratory equivalent partition for the axiom graph is $\{1 \mid 2, 3, 4, 5\}$. This implies that we can employ the rule $h(2) < h(3) < h(4) < h(5)$ whenever searching for occurrences of the axiom graph in the graph $\overline{G}$. For the RHS of the production p_1, the optimal partition is $\{1 \mid 3 \mid 4, 5, 6\}$. Since the graph $Union[p_1]$ also has an exploratory equivalent partition in which the vertices 4, 5, and 6 are part of the same equivalence class, we can enforce the rule $h(4) < h(5) < h(6)$ for every r-homomorphism established between the RHS of the production p_1 and the graph $\overline{G}$. Because of the interleaved discover-and-augment cycle, the enforcement of these rules may significantly reduce the parsing time.

For the task of parsing the graphs of methane, ethane, and propane against the grammar of Fig. 8, Table I compares the duration of parsing without considering exploratory equivalence (EE) and the duration of parsing when exporatory equivalence is taken into account in the form of imposing constraints on r-homomorphisms between the RHSs and the graph $\overline{G}$. The experiments were conducted on a 3.40-GHz Intel Core i7 machine.

The difference between the two versions of the parser is striking. Without using the rules based on exploratory equivalence, the parser quickly succumbs to a combinatorial explosion as the size of the input graph increases; it took more than 11 hours to parse the graph of propane with 3 vertices C and 8 vertices H. By contrast, when exploratory equivalence is taken into account, the parser takes less than one second (0.989 seconds) even when parsing the graph $C_{30}H_{62}$ (30 vertices C, 62 vertices H). Asymptotically, for a graph with n vertices C, the original parser creates $\Omega(6^n)$ production instances (possibly much more than that), while the version that makes use of exploratory equivalence generates exactly

$12n - 7$ production instances. For many grammars containing symmetries in the sense of exploratory equivalence, the use of exploratory equivalence can reduce the asymptotical parsing time from exponential to polynomial (see [9] for additional examples).

TABLE I. THE TIME REQUIRED TO PARSE THE INDIVIDUAL GRAPHS OF FIG. 9 AGAINST THE GRAMMAR OF FIG. 8.

Graph	Without EE	With EE
methane (CH_4)	0.16	0.14
ethane (C_2H_6)	1.03	0.15
propane (C_3H_8)	41000.	0.16

VI. CONCLUSION

We introduced a novel type of graph equivalence, called exploratory equivalence because of its applicability to various graph search algorithms. Exploratory equivalence was defined as an automorphism-based equivalence relation on graph vertices. In contrast to our usual perceptions about equivalence, exploratory equivalence may induce several distinct vertex set partitions for a given graph.

In addition to defining exploratory equivalence itself, we have also introduced the concept of an optimal exploratory equivalent partition for a given graph. We presented two greedy algorithms for finding such a partition. Both algorithms produce optimal results for a vast majority of input graphs. For instance, considering all non-isomorphic graphs on 8 vertices, the second greedy algorithm produces an optimal partition for 11116 graphs out of 11117, the sole exception being the graph of Fig. 6. Among all non-isomorphic 9-vertex graphs, the algorithm produces suboptimal results for only 2 graphs out of 261080.

In subgraph search algorithms, exploratory equivalence can be employed to prevent or at least reduce multiple discoveries of individual occurrences of graph patterns in a given host graph. In the Rekers-Schürr graph grammar parser, this strategy may bring about immense performance gains, since each discovery of a graph in a host graph results in an augmentation of the same host graph.

A possible direction for the future work is a generalization of exploratory equivalence. As defined in this paper, exploratory equivalence can be regarded as a global relation between vertices. Informally, a pair of vertices may potentially belong to the same exploratory equivalence class only if the entire graph 'looks the same' from the viewpoint of both vertices. For this reason, exploratory equivalence is a fairly infrequent phenomenon for large random graphs, except for sets of leaf vertices attached to the same internal vertex. A natural generalization of 'global' exploratory equivalence is therefore a 'local' version of this concept, where only a limited neighborhood is inspected when determining the equivalence of a set of vertices. However, practical implications of such a definitions have yet to be discovered.

As shown in Section V, exploratory equivalence can be used to impose constraints on graph homomorphisms when searching for occurrences of a given pattern graph inside a given host graph. The purpose of such constraints is to eliminate multiple discoveries of the same occurrence. However, in some cases, the constraints induced by exploratory equivalence do not suffice to cover all automorphisms of the pattern graph. Consider, for example, the graph of Fig. 5. This graph has 12 automorphisms, but the optimal exploratory equivalent partition ($\{\{1, 3, 5 \mid 2 \mid 4 \mid 6\}\}$) only covers half of them. Consequently, the rule $h(1) < h(3) < h(5)$ still allows for two different isomorphisms between a pair of 6-cycles. Besides the constraints induced by the optimal exploratory equivalence, we would need another constraint to cover the rotational symmetry of the graph. The relationship between exploratory equivalence (and other types of equivalence) and graph search constraints is thus another promising direction for the future work.

REFERENCES

[1] D. K. Agrafiotis, V. S. Lobanov, M. Shemanarev, D. N. Rassokhin, S. Izrailev, E. P. Jaeger, S. Alex, and M. Farnum, "Efficient Substructure Searching of Large Chemical Libraries: The ABCD Chemical Cartridge," *J. Chem. Inf. Model.*, 2011. doi: 10.1021/ci200413e

[2] J. M. Barnard, "Substructure searching methods: Old and new," *J. Chemical Information and Computer Sciences*, vol. 33, no. 4, pp. 532–538, 1993. doi: 10.1021/ci00014a001

[3] M. O. Jackson, *Social and Economic Networks*. Princeton, NJ, USA: Princeton University Press, 2008. ISBN 0691134405, 9780691134406

[4] D. Knoke, *Political Networks: The Structural Perspective*, ser. Structural Analysis in the Social Sciences. Cambridge University Press, 1994. ISBN 9780521477628

[5] B. Hopkins, "Kevin Bacon and graph theory," *PRIMUS*, vol. 14, no. 1, pp. 5–11, 2004. doi: 10.1080/10511970408984072

[6] F. V. Fomin and D. Kratsch, *Exact Exponential Algorithms*. Springer, 2011.

[7] J. R. Ullmann, "An Algorithm for Subgraph Isomorphism," *J. Assoc. for Computing Machinery*, vol. 23, pp. 31–42, 1976. doi: 10.1145/321921.321925

[8] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs." *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–72, Oct. 2004. doi: 10.1109/TPAMI.2004.75

[9] L. Fürst, M. Mernik, and V. Mahnič, "Improving the graph grammar parser of Rekers and Schürr," *IET Software*, vol. 5, no. 2, pp. 246–261, 2011. doi: 10.1049/iet-sen.2010.0081

[10] J. Rekers and A. Schürr, "Defining and parsing visual languages with Layered Graph Grammars," *Journal of Visual Languages and Computing*, vol. 8, no. 1, pp. 27–55, 1997. doi: 10.1006/jvlc.1996.0027

[11] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, and U. Montanari, Eds., *Handbook of graph grammars and computing by graph transformation (Vols. 1.–3.)*. World Scientific, 1997–1999.

[12] G. Rozenberg and E. Welzl, "Boundary NLC graph grammars – basic definitions, normal forms, and complexity," *Information and Control*, vol. 69, no. 1–3, pp. 136–167, 1986. doi: 10.1016/S0019-9958(86)80045-6

[13] L. Liberti, "Automatic generation of symmetry-breaking constraints," in *COCOA*, ser. Lecture Notes in Computer Science, B. Yang, D.-Z. Du, and C. Wang, Eds., vol. 5165. Springer, 2008. doi: 10.1007/978-3-540-85097-7_31 pp. 328–338.

[14] A. Mucherino, C. Lavor, and L. Liberti, "Exploiting symmetry properties of the discretizable molecular distance geometry problem," *J. Bioinformatics and Computational Biology*, vol. 10, no. 3, 2012. doi: 10.1142/S0219720012420097

[15] B. D. McKay and A. Piperno, "Practical graph isomorphism, ii," *J. Symbolic Computation*, vol. 60, pp. 94–112, 2013. doi: 10.1016/j.jsc.2013.09.003

[16] M. G. Everett and S. P. Borgatti, "Regular equivalence: General theory," *Journal of mathematical sociology*, vol. 19, no. 1, pp. 29–52, 1994. doi: 10.1080/0022250X.1994.9990134