

## Article

# A Symmetry-Breaking Node Equivalence for Pruning the Search Space in Backtracking Algorithms

Uroš Čibej <sup>\*,†</sup> , Luka Fürst <sup>†</sup> and Jurij Mihelič 

Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, 1000 Ljubljana, Slovenia; luka.fuerst@fri.uni-lj.si (L.F.); jurij.mihelic@fri.uni-lj.si (J.M.)

\* Correspondence: uros.cibej@fri.uni-lj.si

† These authors contributed equally to this work.

Received: 6 September 2019; Accepted: 13 October 2019; Published: 15 October 2019



**Abstract:** We introduce a new equivalence on graphs, defined by its symmetry-breaking capability. We first present a framework for various backtracking search algorithms, in which the equivalence is used to prune the search tree. Subsequently, we define the equivalence and an optimization problem with the goal of finding an equivalence partition with the highest pruning potential. We also position the optimization problem into the computational-complexity hierarchy. In particular, we show that the verifier lies between  $\mathcal{P}$  and  $\mathcal{NP}$ -complete problems. Striving for a practical usability of the approach, we devise a heuristic method for general graphs and optimal algorithms for trees and cycles.

**Keywords:** graph equivalence; symmetry breaking; backtracking; monomorphism search; search tree pruning; graph algorithm

## 1. Introduction

The work presented in this paper was primarily motivated by the *subgraph isomorphism problem*, the goal of which is to find a single (or all, depending on the definition) occurrence(s) of a given pattern graph  $G$  in a given host graph  $H$ . This problem has numerous applications in fields such as chemistry, where it is often of interest to search for a specific substructure (e.g., a functional group) within a larger structure (e.g., a molecule) [1]. Subgraph isomorphism is also a key component of graph grammar parsing [2,3]; to determine whether a given graph  $G$  can be generated by a given graph grammar, the graphs on the productions' right-hand sides have to be matched against subgraphs of the graph  $G$ .

The decision version of the subgraph isomorphism problem is  $\mathcal{NP}$ -complete [4], and its counting version is  $\#\mathcal{P}$ -complete [5]. A polynomial-time algorithm is therefore unlikely to exist. Besides that, none of the algorithms developed thus far perform in the worst case any better than a naive occurrence enumeration approach [6]. Most of the exact algorithms are based on backtracking (e.g., [7–12]), utilizing different pruning techniques to reduce the search space. However, when designing a practically efficient backtracking algorithm, pruning is not the only parameter that influences its speed. A very important factor is the overhead that is required for these techniques to work (such as bookkeeping and computations that have to be done at every node of the search tree). Symmetry-breaking is one possible technique that reduces the search space significantly, but has not been utilized in any subgraph-isomorphism solver. The biggest reasons for this are the complexity of implementation and a large overhead which ultimately leads to slower algorithms.

To address this problem, we introduce a novel approach to symmetry-breaking whose main goal is to be lightweight and suitable for implementation in state-of-the-art solvers and not so much to encompass as much symmetries as possible. Our approach is based on a type of equivalence on graph vertices, called *exploratory equivalence* (EE). Exploratory equivalence induces an *exploratory equivalent*

*partition* of the vertex set. As shown below, if a set of vertices (say,  $P$ ) is an equivalence class in an EE partition of the pattern graph  $G$ , then the vertices in  $P$  can be regarded as being interchangeable during the search for occurrences of  $G$  in the host graph  $H$ , reducing the number of distinguishable mappings between the vertices in  $P$  and those of the graph  $H$  by a factor of  $|P|!$ .

Since several EE partitions can be defined for a pattern graph with more than one automorphism, a problem that immediately arises is that of finding an EE partition that brings about the greatest speedup in the subgraph isomorphism search. We define this problem as the *maximum EE partition problem* (MAXEE).

Although the subgraph isomorphism problem was our initial motivation, the concept has a much broader application. As we demonstrate, exploratory equivalence can be used to speed up backtracking algorithms for a wide variety of search problems.

The main contributions of this paper are the following:

- The definition of a novel vertex equivalence on graphs, which is simple to use in various backtracking algorithms and, even more importantly, very lightweight, i.e., it introduces very little computational overhead on the existing backtracking algorithms.
- Even though we start with the subgraph isomorphism, we show that exploratory equivalence can be applied to a broader set of search problems. Whenever the goal of the problem in question is to find an injective mapping between a given pair of sets that satisfies a certain predicate, exploratory equivalence can be used to speed up the backtracking-based search approach. Furthermore, the EE-based performance improvement technique that we describe in this paper can be employed *in addition to* other search-space pruning methods, not just *instead* of them.
- We analyze the computational complexity of the defined optimization problem. More precisely, we show that the verifier for the optimization problem is at least as hard as the graph isomorphism problem and at most as hard as the setwise stabilizer problem. As a consequence, the decision version of MAXEE at present cannot be classified as a member of  $\mathcal{NP}$ .
- We develop a heuristic method for solving the MAXEE problem. We were able to see the practicality of this method in our practical assessment on a set of realistic graphs.
- For two families of graphs, namely trees and cycles, we provide exact and efficient methods for finding the optimal exploratory efficient partition. For the trees we devise a quadratic algorithm and for cycles we provide an analytic solution that gives a formula for the optimal solution for any cycle.

The rest of this paper is structured as follows. We first discuss the related work in Section 2. In Section 3, we define the concepts and notation that is used throughout the paper. Section 4 generalizes the search problem as a monomorphism search problem and provides a framework in which our equivalence will work. In Section 5, we define exploratory equivalence and the MAXEE problem. Section 6 defines the problem of checking whether a given vertex set partition is exploratory equivalent and determines the lower and the upper bound for its computational complexity. In Section 7, we present two algorithms for solving MAXEE: a suboptimal heuristic algorithm for general graphs and an exact algorithm for trees. In addition, we present an analytic solution for MAXEE on cycle graphs. In Section 8, we apply MAXEE to subgraph isomorphism search and show experimental results on real-world host graphs. Section 9 discusses the proposed approach and concludes the paper.

## 2. Related Work

Symmetries and symmetry breaking methods have been studied extensively in several contexts, most notably in graph theory, constraint satisfaction problems (CSP), and integer programming (IP). Graph theory has been interested mostly in identifying and describing different types of symmetries, whereas a great deal of work in CSP and IP has delved more into algorithmic aspects of symmetries, i.e., using them to speed up the solvers for these mathematical programs. Our work touches both

the way to capture these symmetries and methods to use the presented concepts for speeding up computations. In this section, we summarize the most important related concepts and highlight the similarities and differences with our work.

**Graph theory.** The basic problem when dealing with symmetries is the graph isomorphism (or more precisely graph automorphism) problem, which is the cornerstone of theoretical computer science. Even though it is still not known to be in  $\mathcal{P}$ , highly efficient implementations [13] made this problem practical quite some time ago. Even more encouragingly, recent advances by Babai [14] brought this problem much closer to  $\mathcal{P}$  also in theory. The practicality of the graph isomorphism problem is highly important in this context, since we heavily rely on existing solvers to produce graph automorphisms and we strive for immediate practical use of the algorithms presented in this paper as well.

In graph theory, the concepts closest to our work are the notions of *distinguishing number* and *distinguishing coloring*, defined in [15]. In a nutshell, the distinguishing coloring of a graph is the coloring where the only color-preserving automorphism is the identity. The distinguishing number of a graph is the smallest number of colors for such a coloring. Since its definition, it has spawned a large body of work [16–19], mainly elaborating various families of graphs and dealing with the computational complexity of finding this number. In this paper, we also describe a coloring, but our goal differs in several points. We define a weaker mechanism for symmetry breaking, i.e., we do not strive to capture all symmetries. Instead, our main focus is to define a mechanism that is more amenable to algorithmic usage. By contrast, distinguishing coloring is not directly usable in such applications. Let us give a small example to demonstrate the difference between these two notions. As is clear from the definition in Section 5, the partition  $\{1, 4, 5 \mid 2, 3, 6\}$  is a distinguishing coloring for the graph  $G$  in Figure 1, but it is not an exploratory equivalent partition.

Besides distinguishing coloring, graph theory abounds in various other graph equivalences. A graph equivalence is simply a partition (coloration) of the vertex set of a given graph. Although several related notions appear in the literature [20–22], exploratory equivalence is, to the best of our knowledge, a novel concept. Everett and Borgatti [20] surveyed a large class of so-called regular partitions. In what follows, we demonstrate that exploratory equivalence does not belong to this class.

Given a partition  $P$ , let  $C_P(v)$  denote the class to which the vertex  $v$  belongs. Similarly, for a vertex set  $S \subseteq V$ , let  $C_P(S)$  denote the set of classes assigned to the vertices of  $S$ , i.e.,  $C_P(S) = \{C_P(v) \mid v \in S\}$ . A partition  $P$  of a graph is *regular* if the equality of the classes of two vertices implies the equality of the classes of the corresponding neighborhoods. More formally, a partition  $P$  of a graph  $G$  is regular if and only if for all  $i, j \in V$  we have

$$C_P(i) = C_P(j) \implies C_P(\mathcal{N}(i)) = C_P(\mathcal{N}(j)). \quad (1)$$

Many different types of partitions (i.e., colorations) are regular, e.g., strong and weak structural coloration, orbit coloration, perfect coloration, and exact coloration (see [20] for details). For example, the orbit coloration (i.e., the partition consisting of the orbits) of the graph  $G$  in Figure 1 is  $\mathcal{P} = \{1, 2 \mid 3, 4, 5, 6\}$ . Incidentally, note this partition is not exploratory equivalent.

As it turns out, exploratory equivalence is not regular. To demonstrate this, consider again the graph  $G$  in Figure 1 and its exploratory equivalent partition  $\{1, 2 \mid 3, 4 \mid 5, 6\}$  (again, see Section 5 for definition), with different colors assigned to individual equivalence classes. It is easy to see that this partition is not regular, since  $C(1) = C(2)$  but  $C(\{3, 4\})$  is not equal to  $C(\{5, 6\})$ .

**CSP, IP, and SAT** Symmetry-breaking is a common technique in constraint satisfaction problems, integer programming, and satisfiability [23,24]. Similar to the distinguishing coloring for graphs, Crawford et al. [25] tried to capture all symmetries in any propositional satisfiability problem and demonstrated how it can be used in constraint satisfaction problem. The symmetries are described as a tree of symmetry-breaking predicates that can be used as a preprocessing step for speeding up of search algorithms. However, there are many drawbacks to this approach, most notably: (1) the intractability of producing such a tree; and (2) the potentially exponential size of the resulting tree.

Again, our goals are different; we do not wish to capture all the symmetries, but instead strive for the simplicity of the devised mechanism. In addition, we always work with the compact representation of the symmetries, i.e., the polynomial-size generating set of the automorphism group. The resulting exploratory equivalence produces extremely simple symmetry breaking predicates, thus inducing almost no overhead during the search process.

Besides this very general approach, there is an abundance of practical methods in both the area of constraint satisfaction problems, integer programming, and satisfiability (see, e.g., [26,27] for an extensive overview of different methods).

Similar to our idea, the main mechanism for breaking symmetries in CSPs, IPs, and SATs is to apply symmetry breaking constraints, while guaranteeing that no solution (up to automorphisms) is being left out. The methods can roughly be divided into two groups: *dynamic symmetry breaking* and *static symmetry breaking* methods. This distinction depends on when the symmetry breaking predicates are computed and enforced. Dynamic methods generate constraints during the search process and have been devised mainly for CSP problems [28]. These methods have also been successfully applied in practice, most notably in the GAP algebra system [29], where it is called Symmetry Breaking During Search (SBDS) [30]. Our method is not dynamic, in the sense that no recomputations of the symmetries are required during the backtracking search.

In contrast to dynamic methods, static methods add inequalities to the initial formulation of the problem. As a special case, the ones most similar to our work and also most widely applied are the methods that add constraints that guide the search algorithm only to a lexicographically smallest solution. There are three major algorithms that work in this manner: Symmetry Backtracking Search (SBS) [31], Symmetry Breaking via Dominance Detection (SBDD) [32,33], and Isomorphism Pruning (IsoP) [34]. These three algorithms were developed independently, but follow the same basic concepts. All of them prune a branch of the enumeration tree if it leads to a lexicographically larger solution. This is enforced by checking whether a certain permutation exists inside the automorphism group. The automorphism group can be kept in a compact representation (i.e., as a set of generators), but checking whether a certain permutation is contained in the group induces a significant overhead at each node of the search tree. Judging from the positive practical results, the costs of group membership checks pay off, but we show below that similar constraints can be enforced without the significant overhead.

### 3. Preliminaries

Let  $f|_A$  denote the *restriction* of a function  $f$  to a set  $A$ , i.e., the function  $f$  defined on  $A \cap \text{dom } f$ . Let  $G = (V, E)$  denote a *simple undirected graph*, where  $V = \{1, 2, \dots, n\}$  is a set of vertices and  $E \subseteq V \times V$  is a set of edges. A connected acyclic undirected graph is called a *tree*. The set of vertices adjacent to a vertex  $u$  is called the *neighborhood* of  $u$  and is denoted  $\mathcal{N}(u) = \{v \in V \mid (u, v) \in E\}$ . Similarly,  $\mathcal{N}(U) = \cup_{u \in U} \mathcal{N}(u)$  denotes the set of vertices adjacent to any vertex in  $U$ .

Given two graphs,  $G = (V, E)$  and  $H = (U, F)$ , several different homomorphisms from  $G$  to  $H$  can be defined. In the paper, we are mainly interested in an *isomorphism*, which is a bijective mapping  $h: G \rightarrow H$  such that  $(i, j) \in E \iff (h(i), h(j)) \in F$ , and an *automorphism*, which is an isomorphism whose domain is equal to its codomain, i.e.,  $h: G \rightarrow G$ . We write  $G \simeq H$  if there exists an isomorphism from  $G$  to  $H$ ; such graphs  $G$  and  $H$  are called *isomorphic*. A *monomorphism* is an injective mapping  $h: G \rightarrow H$  such that for all  $(i, j) \in E$  it holds that  $(h(i), h(j)) \in F$ . A monomorphism  $h: G \rightarrow H$  is often called *subgraph isomorphism*, since it is equivalent to an isomorphism between the graph  $G$  and a subgraph of the graph  $H$ . A subgraph in  $H$  that is isomorphic to  $G$  is called an *occurrence* of  $G$  in  $H$ .

The goal of the *graph isomorphism problem* (denoted GI) is to determine whether two graphs are isomorphic. Interestingly, GI is one of the few problems belonging to  $\mathcal{NP}$  that are not known to be either  $\mathcal{NP}$ -complete or in  $\mathcal{P}$ .

Note that an automorphism is a *permutation*, i.e., a bijective function of a finite set  $P$  onto itself. The set of all permutations of a set  $P$ , together with function composition, forms a group

called *symmetric group*, denoted with  $\text{Sym}(P)$ . All groups used in this paper are subgroups of a symmetric group. To simplify notation, we represent every group by its underlying set. We also define  $S_n \equiv \text{Sym}(\{1, 2, \dots, n\})$ . Every group can be succinctly described by a set of *generators*, which is a subset of the group such that any group element can be expressed as the combination (under the group operation) of finitely many generators and their inverses. Given the above definitions, the set of all automorphisms of a graph  $G$  on  $n$  vertices is a subgroup of the symmetric group and is defined as

$$\text{Aut}(G) = \{a \in S_n \mid G \simeq a(G)\}. \quad (2)$$

Given a set  $A \subseteq S_n$  and a set  $P \subseteq \{1, \dots, n\}$ , the *pointwise stabilizer* of  $A$  with respect to  $P$  is the set of all permutations in  $A$  that fix all elements of  $P$ , i.e.,

$$\text{PointStab}(A, P) = \{a \in A \mid \forall i \in P: a(i) = i\}. \quad (3)$$

The *setwise stabilizer* of  $A$  with respect to  $P$  is the set of all permutations in  $A$  that permute the elements of  $P$  only with each other, not with elements outside  $P$ . Formally,

$$\text{SetStab}(A, P) = \{a \in A \mid \{a(i) : i \in P\} = P\}. \quad (4)$$

Note that the identity always belongs to any stabilizer and that  $\text{PointStab}(A, P) \subseteq \text{SetStab}(A, P)$ .

A *partition* is a family  $\{P_1, P_2, \dots, P_s\}$  of nonempty subsets of a finite set  $S$  such that every element in  $S$  is a member of exactly one of the subsets, i.e.,  $P_i \subseteq S$ ,  $P_i \neq \emptyset$  for all  $1 \leq i \leq s$ ,  $P_i \cap P_j = \emptyset$  for all  $1 \leq i, j \leq s$  with  $i \neq j$ , and  $\bigcup_{1 \leq i \leq s} P_i = S$ . A partition  $\{P_1, P_2, \dots, P_s\}$  will often be written as  $\{i \in P_1 \mid i \in P_2 \mid \dots \mid i \in P_s\}$ . For example, the partition  $\{\{1, 2\}, \{3\}, \{4\}\}$  may be written as  $\{1, 2 \mid 3 \mid 4\}$ . If the order of the sets in a partition is important, the partition is called *ordered*. An ordered partition will be specified as  $\langle i \in P_1 \mid i \in P_2 \mid \dots \mid i \in P_s \rangle$ , e.g.,  $\langle 1, 2 \mid 3 \mid 4 \rangle$ .

#### 4. Framework for Backtracking Search Algorithms

To demonstrate how to use exploratory equivalence in backtracking-based search, let us first define a more general problem. This definition generalizes many practical problems and also shows the wide variety of applications in which exploratory equivalence can be used.

**Definition 1.** (*Monomorphism search problem*) Given two sets,  $A$  and  $B$ , and a predicate  $P$ , find a monomorphism  $m : A \rightarrow B$  satisfying a predicate  $P(m)$ .

Notice that the subgraph isomorphism problem is an instance of this problem, where  $A$  and  $B$  are the vertex sets of the two graphs and  $P$  is the predicate checking for the connectivity preservation of the monomorphism  $m$ . Another example is the  $N$ -queens problem, where  $A = \{1, \dots, N\}$  are the  $N$  queens to be placed on the chessboard  $B = \{(i, j) : i, j \in \{1, \dots, N\}\}$  and the predicate ensures that the placed queens do not attack each other. A third example is the family of problems called vertex ordering problems [35]. Vertex ordering problems include, e.g., the minimal bandwidth of a graph [36] and two widely used features of a graph, the pathwidth and the treewidth of a graph [37]. For vertex ordering,  $A$  is the vertex set  $V$  of the input graph and  $B = \{1, 2, \dots, |V|\}$ . The predicates depend on the type of the problem, but can usually be easily defined.

Algorithm 1 gives a description of the backtracking framework for the monomorphism search problem that exploits exploratory equivalence. Exploratory equivalence is used to impose an ordering on the monomorphisms, thus significantly reducing the size of search spaces that exhibit many symmetries. The inputs to this algorithm are:

1. A partial monomorphism  $m$ : The current mapping from  $A$  to  $B$ , where only some of the elements of  $A$  are mapped.

2. The index  $idx$  of an element of  $A$ : The depth of the search tree, which also represents the index of the element  $a \in A$  currently being mapped to  $B$ .
3. An ordering of the set  $A$ : A bijective function  $ord_A: A \rightarrow \{1, \dots, |A|\}$ , which represents the order in which the elements of  $A$  are being assigned to the elements of  $B$ . We use  $ord_A(a)$  to find the order of the element  $a$ , and  $ord_A^{-1}(idx)$  to find the element having rank  $idx$ . As a shorthand, when dealing with sets of elements (sets of indices), we use  $ord_A(S)$  to denote the set of indices of the elements  $a \in S$  and  $ord_A^{-1}(S)$  to denote the set of elements at the indices  $i \in S$ .
4. An ordering of the set  $B$ : A bijective function  $ord_B: B \rightarrow \{1, \dots, |B|\}$ , which represents the order in which the elements of  $B$  are selected as the images of  $a \in A$ .
5. An exploratory equivalent partition of the set  $A$ : A set of disjoint sets  $\{P_1, P_2, \dots, P_k\}$  covering the set  $A$ . We use  $eq_A(a)$  to denote the class  $P_i$  that contains the element  $a$ .
6. A predicate  $P: (A \rightarrow B) \rightarrow \{true, false\}$  that defines the goal of the search problem. It describes the problem-specific structure that we are searching for.

---

**Algorithm 1:** Backtracking algorithm FIND for the monomorphism search problem.
 

---

```

1 function FIND( $m, idx, ord_A, ord_B, eq_A, P$ )
   Output: monomorphism  $A \rightarrow B$  satisfying predicate  $P$ 
2   if  $idx > |A| \wedge P(m)$  then
3     return  $m$ ;                                     // a monomorphism found
4   end
5    $a \leftarrow ord_A^{-1}(idx)$ ;
   /* check whether  $a$  is the first element of  $eq_A(a)$  in  $ord_A$  */
6   if  $idx = \min(\{ord_A(x) \mid x \in eq_A(a)\})$  then
7      $start \leftarrow 1$ ;                                // start the search at the first element of  $B$ 
8   else
9      $maxE \leftarrow \max(\{ord_B(m(x)) \mid x \in eq_A(a) \cap ord_A^{-1}(\{1, 2, \dots, idx - 1\})\})$ ;
10     $start \leftarrow maxE + 1$ 
11  end
12  for  $k$  in  $\{start, \dots, |B|\}$  do
13     $m(a) \leftarrow ord_B^{-1}(k)$ ;
14    FIND( $m, idx + 1, ord_A, ord_B, eq_A, P$ );
15    backtrack, i.e., remove  $a$  from the domain of  $m$ ;
16  end

```

---

The main structure of the algorithm is the same as classical backtracking, where  $ord_A$  imposes the order in which the elements of  $A$  are assigned to the elements of  $B$ . Without the symmetry-breaking constraints, every element of  $B$  is tried, but by using an  $eq_A$  partition, the constraints

$$\forall P \in eq_A, a_1, a_2 \in P : ord_A(a_1) < ord_A(a_2) \implies ord_B(m(a_1)) < ord_B(m(a_2)) \quad (5)$$

are imposed. These constraints are enforced by setting the  $start$  variable, i.e., the index of the element in  $B$  for which an assignment is attempted.

Notice that the framework is very general and may be used for any problem that can be described as a monomorphism search. The partition  $eq_A$  can represent different notions for different domains. In this paper, we focus on graph equivalence (i.e.,  $A$  is the set of vertices of a graph), but we do not impose any restrictions on the set  $B$ . Another nice property of the framework is that it does not impose any restrictions on the orderings of  $A$  and  $B$ . This is a very important feature, since ordering is typically one of the most significant optimization techniques. In our framework, any problem-specific ordering can be used. Furthermore, there are no specific interferences to other pruning techniques, which can be easily plugged into this procedure.

## 5. Exploratory Equivalence and Maximum Exploratory Equivalence

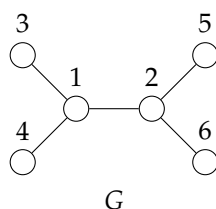
This section formally defines exploratory equivalence on graphs. Exploratory equivalence yields several possible partitions of a graph, with varying effectiveness for symmetry breaking. To formalize this fact, we introduce a score function that defines an optimization problem over the set of all partitions.

### 5.1. Exploratory Equivalence

In what follows, we use a predicate to tell whether a particular set of permutations “includes” all possible permutations of a given set. We use the following definition.

**Definition 2.** (Cover) A set of permutations  $A \subseteq S_n$  covers a set  $P \subseteq \{1, \dots, n\}$  if for every permutation  $\sigma$  of the set  $P$  there exists a permutation  $a \in A$  such that  $a(i) = \sigma(i)$  for all  $i \in P$ , i.e.,

$$\text{cover}(A, P) \equiv \forall \sigma \in \text{Sym}(P) \exists a \in A \forall i \in P: \sigma(i) = a(i). \quad (6)$$



**Figure 1.** A sample pattern graph  $G$  used in the paper to give examples of defined terms.

For example, consider the graph  $G$  in Figure 1: the set  $\text{Aut}(G)$  covers  $\{3, 5\}$ , since it contains the permutation (automorphism) 123456, where  $a(3) = 3$  and  $a(5) = 5$ , as well as 215634, where  $a(3) = 5$  and  $a(5) = 3$ .

**Definition 3.** (Ordered EE partition) Given a graph  $G = (V, E)$ , an ordered partition  $\langle P_1, P_2, \dots, P_s \rangle$  of  $V$  is exploratory equivalent (EE) if for all  $i \in \{1, \dots, s\}$  we have

$$\text{cover}(A_{i-1}, P_i) \text{ and } A_i = \text{PointStab}(A_{i-1}, P_i), \quad (7)$$

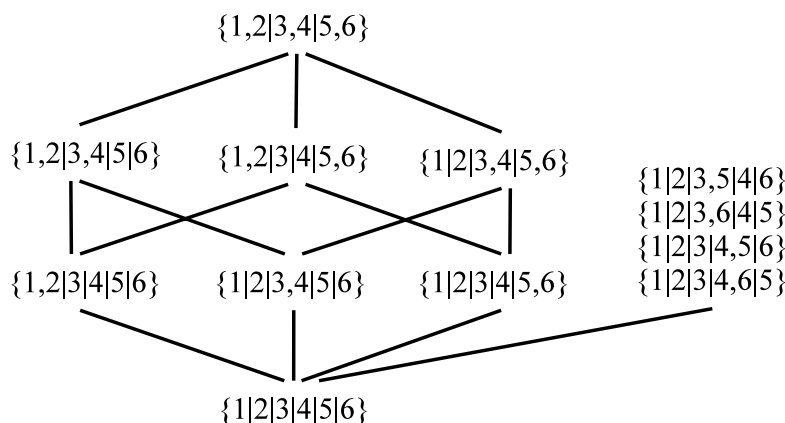
where  $A_0 = \text{Aut}(G)$ .

For the graph  $G$  in Figure 1, one of the ordered EE partitions is  $\langle 1, 2 \mid 3, 4 \mid 5, 6 \rangle$ ; the corresponding stabilizer subgroups are  $A_1 = \{123456, 123465, 124356, 124365\}$ ,  $A_2 = \{123456, 123465\}$ , and  $A_3 = \{123456\}$ .

The above definition depends on the order of the sets in the partition. To use the defined notions in an order-oblivious setting, we also need the following definition.

**Definition 4.** (EE partition) Given a graph  $G = (V, E)$ , a partition  $\{P_1, P_2, \dots, P_s\}$  of  $V$  is exploratory equivalent if there exists an ordered exploratory equivalent partition  $\langle P_{i_1}, P_{i_2}, \dots, P_{i_s} \rangle$  for a set of distinct indices  $i_j \in \{1, \dots, s\}$ .

Notice that there can be many different EE partitions for a given graph. Figure 2 presents an example.



**Figure 2.** The Hasse diagram of all EE partitions of the graph  $G$  in Figure 1. (The four partitions on the right-hand side are actually four separate vertices in the diagram.)

Every partition represents an equivalence relation, thus the members of a partition are called *classes* rather than *sets*. Hence, the partition  $\{1 \mid 2 \mid 3,4 \mid 5 \mid 6\}$  is composed of five equivalence classes and Vertices 3 and 4 are equivalent.

## 5.2. Maximum Exploratory Equivalence

As testified by the example in Figure 2, there may be many EE partitions for a particular graph. To be able to differentiate among partitions in terms of the pruning effectiveness of the search space, we provide a suitable measure. Furthermore, we also define a corresponding optimization problem that captures the main goal of our work, i.e., to speed up various exploration algorithms.

First, consider the monomorphism search problem described above. The size of the search space, i.e., the number of possible monomorphisms from the set  $A$  to the set  $B$ , is  $\binom{|B|}{|A|} |A|!$ . Often, a search algorithm has to generate and process each and every of them. However, when the set  $A$  can be partitioned into the classes  $P_1, P_2, \dots, P_s$  containing equivalent elements (in the problem-specific sense), then the number of possibilities is reduced to

$$\binom{|B|}{|A|} \frac{|A|!}{|P_1|! |P_2|! \dots |P_s|!}. \quad (8)$$

The reduction factor is thus given in the denominator as the product of factorials of set cardinalities. We often call this factor the *score* of a particular partition and use it as the objective in the maximum exploratory equivalent partition problem (MAXEE):

**Definition 5.** (MAXEE) Given a graph  $G = (V, E)$ , find an EE partition  $\{P_1, \dots, P_s\}$  of  $V$  that maximizes the score

$$\text{score}(P) = \prod_{i=1}^s |P_i|!. \quad (9)$$

## 6. Computational Complexity

In this section, we tackle the computational complexity of the defined optimization problem. We first define an auxiliary problem, i.e., a verifier (VERIFYEE) to check whether a given partition is a valid EE partition. We show that VERIFYEE is GI-hard. Next, we define a natural decision version of MAXEE (we call it MAXEEDEC) and reduce VERIFYEE to MAXEEDEC, demonstrating its GI-hardness.

Subsequently, we position VERIFYEE more precisely; we show that the verifier belongs to the class of so-called *intermediate problems*, i.e., problems between  $\mathcal{P}$  and  $\mathcal{NP}$ -complete. This is done by proving it to be at most as hard as the problem of finding the setwise stabilizer, which is currently also believed to be intermediate.

First, let us define the verifier problem.

**Definition 6.** (VERIFYEE) Given a graph  $G = (V, E)$  and an ordered partition  $\mathcal{P} = \{P_1, P_2, \dots, P_s\}$  of the vertex set  $V$ , decide whether  $\mathcal{P}$  is an EE partition of the graph  $G$ .

**Theorem 1.** (GI-hardness) VERIFYEE is GI-hard.

**Proof.** A pair of input graphs to the GI problem,  $G = (V, E)$  and  $H = (U, F)$  (without loss of generality, assume  $|V| = |U|$ ), is transformed into the instance  $(G' = (V', E'), \mathcal{P})$  of VERIFYEE, where

$$V' = V \cup U \cup \{v_G, v_H\},$$

$$E' = E \cup F \cup \{(v_G, v) \mid v \in V\} \cup \{(v_H, u) \mid u \in U\} \cup \{(v_G, v_H)\}, \quad (10)$$

and

$$\mathcal{P} = \{\{v_G, v_H\}\} \cup \{\{v\} \mid v \in V \cup U\}. \quad (11)$$

Now, we argue that  $\mathcal{P}$  is an EE partition of  $G'$  iff  $G$  is isomorphic to  $H$ .

( $\implies$ ) If  $(G', \mathcal{P})$  is in VERIFYEE, then there exists an automorphism on  $G'$  mapping  $v_G$  to  $v_H$ . Furthermore, to preserve connectivity, all the vertices in  $V$  are mapped to the vertices in  $U$ , which is exactly an isomorphism between  $G$  and  $H$ .

( $\impliedby$ ) Let  $h$  denote an isomorphism from  $G$  to  $H$ . Construct an automorphism on  $G'$  that maps  $v_G$  to  $v_H$  and all the other vertices according to  $h$  and  $h^{-1}$ . Henceforth,  $\{v_G, v_H\}$  is an EE class and  $\mathcal{P}$  is an EE partition.  $\square$

To establish the complexity of the MAXEE problem, we define a decision version of this problem.

**Definition 7.** (MAXEEDEC) Given a graph  $G = (V, E)$  and an integer  $k$ , decide whether there exists an exploratory equivalent partition  $\mathcal{P}$  with  $\text{score}(\mathcal{P}) \geq k$ .

We show the GI-hardness of this problem and the proof will require the following auxiliary lemma (i.e., we need to show that more refined partitions can only have smaller scores).

**Lemma 1.**  $\forall (k_1, k_2, \dots, k_l), \sum_{i=1}^l k_i = n, k_i > 0, l > 1 : n! > \prod_{i=1}^l k_i!$

**Proof.** We need to prove this only for  $l = 2$ , since any partition of  $n$  can be written as a sequence  $n = k_1 + k'_2 = k_1 + k_2 + k'_3 = \dots = k_1 + k_2 + \dots + k'_{l-1} = k_1 + k_2 + \dots + k_{l-1} + k_l$ . If we manage to prove the inequality for  $l = 2$ , then the inequality follows for any  $l \geq 2$ .

The proof for  $l = 2$  is a simple observation:

$$n! = (1 \cdot 2 \cdot 3 \cdot \dots \cdot k_1) \cdot ((k_1 + 1) \cdot (k_1 + 2) \cdot \dots \cdot (k_1 + k_2)) > k_1! \cdot k_2!, \quad (12)$$

since  $(k_1 + 1) \cdot (k_1 + 2) \cdot \dots \cdot (k_1 + k_2) > k_2!$  and  $k_1 > 0$ .  $\square$

**Theorem 2.** MAXEEDEC is GI-hard.

**Proof.** We prove this by a reduction from VERIFYEE, which is proven to be GI-hard above.

If we had labeled graphs, then we would simply color each  $P_i$  with color  $i$  and ask whether such a labeled graph has an EE partition with score  $\prod_{i=1}^s |P_i|!$ . However, since we are dealing with unlabeled graphs, we have to simulate the labels with gadgets that we add for each  $P_i$ .

The reduction from VERIFYEE  $(G, \mathcal{P})$  to MAXEEDEC  $(G', k)$  is as follows. We label each class  $P_i \in \mathcal{P}$  by connecting each  $v \in P_i$  to a gadget. Each class needs to have a distinct label, and we also

need to make sure that each gadget cannot be mapped (by an automorphism) onto the rest of the graph. For each  $P_i$ , this gadget is a clique of  $K_{n+i}$ , which is connected to the set  $P_i$  through a connecting vertex  $c_i$ . Each vertex in  $P_i \cup K_{n+i}$  is connected to  $c_i$ . The sketch of this construction can be seen in Figure 3.

We can see that the vertices in each  $K_{n+i}$  can only be isomorphic to each other and no other vertex. These vertices have a different degree than any vertex from the original graph  $G$  and also a different degree from any vertex in other cliques. The vertices  $c_i$  are also unique (non-isomorphic to any other vertex), since they are always connected to non-isomorphic neighbors (those from  $K_{n+i}$ ). Consequently, two vertices  $u, v$  from  $G$  can be isomorphic only if they are in the same partition. These observations are helpful at the end of this proof.

Now, we also need to generate the appropriate score value:

$$k = \prod_{P_i \in P} |P_i|!(n+i)!. \quad (13)$$

We can now prove that:

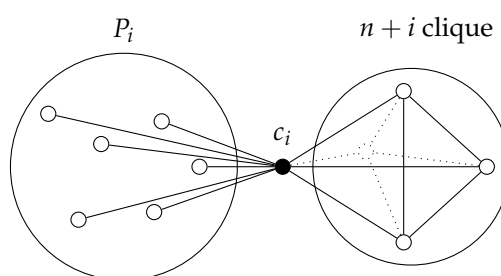
$$\mathcal{P} \in EE(G) \iff \exists \mathcal{P}' \in EE(G') : \text{score}(\mathcal{P}') \geq k. \quad (14)$$

( $\implies$ ) This direction is trivial, since the construction of  $G'$  does not impose any restrictions on isomorphisms in each  $P_i$ . Therefore, each  $P_i$  is a valid EE set also in  $G'$ . However, all vertices in each clique are also a valid EE set, thus we can construct the partition of  $G'$ :

$$\mathcal{P}' = \bigcup_{P_i \in P} \{P_i, V(K_{n+i}), \{c_i\}\}. \quad (15)$$

The score of this partition is exactly  $k$ .

( $\impliedby$ ) We need to show that  $\mathcal{P}'$  must contain all the sets  $P_i \in P$ . Because of the isomorphism restrictions we imposed by adding the gadgets, any EE partition on  $G'$  can only be a refinement of  $\mathcal{P}'$ . Any refined EE partition  $\mathcal{P}'_{ref}$  has a strictly lower score than  $\mathcal{P}'$ , which follows from Lemma 1.  $\square$



**Figure 3.** An overview of the reduction from VERIFYEE to MAXEEDEC.

The next goal of our complexity analysis is to demonstrate that the VERIFYEE problem is (probably) not much harder than GI. To do this, we reduce it to the problem of the setwise stabilizer of permutation groups, which is also GI-hard [38], but widely believed not to be  $\mathcal{NP}$ -hard, for it is efficiently solvable in practice and exhibits similar properties as GI.

The first step of our demonstration is to consider another problem, which is later used as the main ingredient for an algorithm for VERIFYEE. In particular, we consider the problem of deciding the  $\text{cover}(A, P)$  predicate for a given  $A$  and  $P$ . Notice that the necessary condition for  $A$  to cover  $P$  is that  $|A| \geq |P|!$ .

**Lemma 2.** Given a group  $A \subseteq S_n$  and a set  $P \subseteq \{1, \dots, n\}$ , deciding the  $\text{cover}(A, P)$  predicate is polynomial-time reducible to calculating the setwise stabilizer  $\text{SetStab}(A, P)$ .

**Proof.** Let us restrict the set of permutations in  $A$  to those that stabilize the set  $P$ . This operation preserves all permutations that permute the elements of  $P$ . However, there might be more than  $|P|!$  such permutations. Hence, we additionally restrict every generator of the group to the set  $P$ . More formally, we consider the set

$$A' = \{a|_P \mid a \in \text{SetStab}(A, P)\}. \quad (16)$$

Now, we have

$$\text{cover}(A, P) \iff A' = \text{Sym}(P). \quad (17)$$

Notice that it is straightforward to compute  $a|_P$ , since every generator of the stabilizer group consists of two types of cycles: cycles where all elements belong to the set  $P$ , and those where none of the elements belongs to  $P$ .  $\square$

**Theorem 3.** VERIFEE is polynomial-time reducible to the problem of setwise stabilizer.

**Proof.** Following the definition of exploratory equivalence, we give an algorithm to compute VERIFEE (Algorithm 2). We omit checking that  $P_1, P_2, \dots, P_k$  is indeed a partition of a set of vertices.

$\text{PointStab}$  on Line 5 can be computed in polynomial time [39], so the problem further reduces to the cover problem, which is reducible by the previous lemma to  $\text{SetStab}$ .  $\square$

---

**Algorithm 2:** Algorithm for computing VERIFEE.

---

```

1 function VERIFEE ( $A, \{P_1, P_2, \dots, P_k\}$ )
  Output: True if  $\{P_1, P_2, \dots, P_k\}$  is a valid exploratory equivalence.
2  if  $k = 0$  then
3    return true
4  end
5  if  $\{a|_{P_1} \mid a \in \text{SetStab}(A, P_1)\} = P_1$  then
6     $\text{stab} = \text{PointStab}(A, P_1)$ ;
7    return VERIFEE( $\text{stab}, \{P_2, P_3, \dots, P_k\}$ )
8  end
9  return false

```

---

## 7. Algorithms

Due to the complexity results in the previous section, we do not focus on trying to find an optimal solution for the general MAXEE problem. Instead, we first describe a (relatively) time-efficient heuristic algorithm, making it more suitable for potential practical applications. Subsequently, we demonstrate that, by reducing the problem to trees, we can find an optimal solution in polynomial time.

Since we are dealing with a problem that is currently speculated to be harder than  $\mathcal{NP}$ , we focus on finding a good heuristic method whose time complexity remains feasible.

### 7.1. Heuristics for General Graphs

Here, we present a greedy heuristic algorithm for solving the MAXEE problem. The algorithm is based on the recursive procedure given in Algorithm 3. The procedure receives as an input a current permutation group  $A$  as well as a current (partial) EE partition  $\{P_1, P_2, \dots, P_k\}$ , where the set  $P_k$  is not finalized yet and can be extended with additional elements. On the other hand, the sets  $P_i$  with  $1 \leq i < k$  are final; additionally, the group  $A$  is pointwise stabilized to all the elements in these sets. The algorithm starts with  $A = \text{Aut}(G)$  and an empty partial partition.

**Algorithm 3:** Heuristics for the MAXEE problem.

---

```

1 function MAXEE ( $A, \{P_1, P_2, \dots, P_k\}$ )
  Output: One of the exploratory equivalent partitions.
2    $A' \leftarrow \text{PointStab}(P_k, A);$ 
3   if  $|A'| = 1$  then
4     return  $\{P_1, P_2, \dots, P_k\}$ 
5   end
6   if  $k = 0$  then
7      $P_1 \leftarrow \text{BESTEPAIR}(A);$ 
8     return MAXEE ( $A, \{P_1\}$ )
9   end
10   $P'_k \leftarrow P_k \cup \text{AUGMENT}(A, P_k);$ 
11   $P_{k+1} \leftarrow \text{BESTEPAIR}(A');$ 
12  if  $\varphi(A, P'_k) > \varphi(A', P_{k+1})$  then
13    return MAXEE( $A, \{P_1, P_2, \dots, P_{k-1}, P'_k\}$ )
14  else
15    return MAXEE( $A', \{P_1, P_2, \dots, P_k, P_{k+1}\}$ )
16  end

```

---

The halting condition for this algorithm is satisfied when the automorphism group stabilized to all the elements in the partial partition contains only the trivial automorphism. In this case, the current partition is the final solution (Line 4).

When the partial partition is empty ( $k = 0$ ),  $P_1$  becomes a pair of elements with the best potential for expansion (Line 7). Otherwise, in each recursive step, a choice between two possibilities is made (Line 12):

1. The set  $P_k$  is augmented by a single element (the call to AUGMENT computes the best candidate for augmentation; see Lines 10 and 13).
2.  $P_k$  is finalized, a new pair of elements is selected as the set  $P_{k+1}$ , and  $A$  is pointwise stabilized on  $P_k$  (BESTEPAIR returns the pair of elements that exhibit the best potential; see Lines 11 and 15).

This choice is done greedily, using the following criterion

$$\varphi(A, P) = |P|! \cdot |\text{notFixed}(\text{PointStab}(A, P))|!, \quad (18)$$

which estimates the potential of a set  $P$  in a group  $A$ . Here, the  $|P|!$  factor comes from the MAXEE objective function, and

$$\text{notFixed}(A) = \{i \mid \exists a \in A : a(i) \neq i\} \quad (19)$$

consists of positions not yet fixed.

Now, consider the functions BESTEPAIR and AUGMENT (Algorithm 4). In the former, to select the best pair of vertices in a given group  $A$ , we first find candidate pairs using the orbits of the group (Line 2). Notice that a pair of vertices in the same orbit trivially satisfies the *cover* predicate. Nevertheless, this is basically an optimization, for the set of all pairs could be used instead. From such a candidate set, the best pair is chosen (Line 3) using the potential criterion defined above.

**Algorithm 4:** The definition of two auxiliary functions

---

```

1 function BESTEPAIR( $A$ )
   Output: a pair of vertices in the same orbit.
2    $\mathcal{C} \leftarrow \{\{p, q\} \mid p \text{ and } q \text{ are in the same orbit of } A\}$ ;
3   return  $\operatorname{argmax}_{P \in \mathcal{C}} \varphi(A, P)$ 

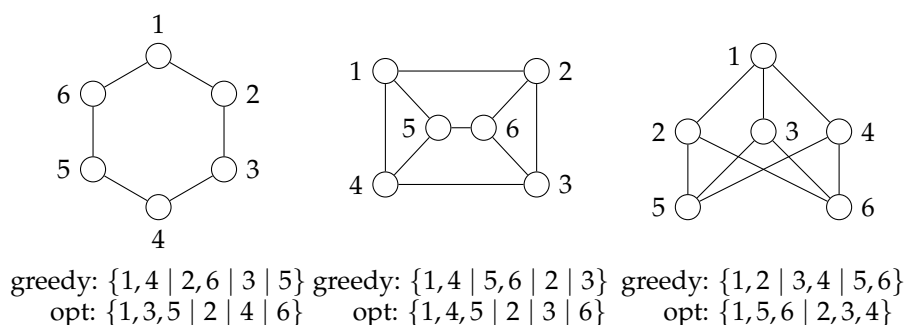
4 function AUGMENT( $A, P$ )
   Output: An augmented exploratory equivalent partition  $P'$ 
5    $\mathcal{C} \leftarrow \{i \mid i \in \text{same orbit as elements of } P\}$ 
6    $\text{VERIFYEE}(A, P \cup \{i\})$ ;
7   return  $\operatorname{argmax}_{i \in \mathcal{C}} \varphi(A, P \cup \{i\})$ 

```

---

A similar idea is behind augmenting an existing EE set  $P$ . The candidates are the elements  $i$  in the same orbit as the elements of  $P$ , and the set  $P \cup \{i\}$  also has to be exploratory equivalent. From this set of candidates, the element with the highest potential criterion function  $\varphi$  is selected.

On certain input instances, the algorithm finds suboptimal partitions. The smallest such inputs are the three six-vertex graphs shown in Figure 4. Nevertheless, we present a small experimental evaluation in Section 9, showing that in most cases the heuristics are indeed highly successful in finding an optimal solution.



**Figure 4.** The graphs on six vertices where the greedy algorithm obtains a suboptimal solution.

### Time Complexity

The time complexity of the described heuristics is dominated by the complexity of the setwise stabilizer. All the other building blocks of this algorithm are of polynomial time complexity. The time complexity of the set stabilizer is still exponential in the classical sense, but that does not do the problem justice. A more informative fact about the set stabilizer problem is that it is not NP-complete and, like GI, this problem is easy on the vast majority of practical instances (for a more detailed description of the context of the set-stabilizer problem, see [40]). This is what makes it a practical option for our heuristics and we could see this almost polynomial behavior also in our practical experiments, which is discussed in Section 9.

#### 7.2. Exact Algorithm on Trees

Since the MAXEE problem is unlikely to be solvable in polynomial time for general graphs, we restrict our attention to trees. We show that for an arbitrary tree, the MAXEE problem can be solved by a polynomial-time algorithm.

To begin with, let us introduce some tree-related concepts. For a given tree  $T = (V, E)$ , the *distance* between vertices  $u$  and  $v$ , denoted  $d(u, v)$ , is the number of edges on the (unique) path from  $u$  to  $v$ . The *height* of a tree  $T$ ,  $h(T)$ , is the maximum distance between a pair of vertices, i.e.,  $h(T) = \max_{\{u, v\} \subseteq V} d(u, v)$ . The *neighborhood subtree* of a vertex  $u$  at a distance  $d$ , denoted  $NS_d(u)$ , is the subtree of  $T$  composed of all vertices whose distance from  $u$  is at most  $d$ . A vertex  $u$  can be automorphically

mapped (a vertex  $u$  can be “automorphically mapped” to a vertex  $v$  if there exists an automorphism that maps  $u$  to  $v$ .) to a vertex  $v$  only if  $NS_d(u) \simeq NS_d(v)$  for all  $d \geq 0$ .

The *eccentricity* of a vertex  $u$ , denoted  $e(u)$ , is the maximum distance between  $u$  and any other vertex in the tree, i.e.,  $e(u) = \max_{v \in V} d(u, v)$ . A *center* of the tree is a vertex with minimum eccentricity. Let  $e = e(c)$ , where  $c$  is a center of the tree.

**Lemma 3.** Every tree has either one or two centers. In the latter case, the centers are connected by an edge.

The proof can be found in [41]. We will denote the centers by  $c_1$  and  $c_2$ , allowing for the possibility that  $c_1 = c_2 = c$ .

Let  $\mathcal{E}(u, v)$  denote the set of edges on the path from  $u$  to  $v$ . For a tree  $T$ , the *centrifugal subtree* of a vertex  $u \notin \{c_1, c_2\}$ , denoted  $CS(u)$ , is a subtree of  $T$  composed of all vertices  $v$  such that  $\mathcal{E}(u, v) \cap (\mathcal{E}(u, c_1) \cup \mathcal{E}(u, c_2)) = \emptyset$ . For a tree with a single center  $c$ ,  $CS(c) = T$ ; for a tree with distinct centers  $c_1$  and  $c_2$ , the tree  $CS(c_i)$  (for  $i \in \{1, 2\}$ ) is induced by the vertex set  $\{v \in V \mid (c_1, c_2) \notin \mathcal{E}(c_i, v)\}$ .

**Lemma 4.** (1) If a tree has a single center  $c$  and if  $e \geq 1$ , there exist distinct vertices  $u'$  and  $v'$  such that  $d(c, u') = d(c, v') = e$  and  $\mathcal{E}(c, u') \cap \mathcal{E}(c, v') = \emptyset$ . (2) If a tree has distinct centers  $c_1$  and  $c_2$  and if  $e \geq 2$ , there exist distinct vertices  $u'$  and  $v'$  such that  $d(c_1, u') = d(c_2, v') = e - 1$  and  $\mathcal{E}(c_1, u') \cap \mathcal{E}(c_2, v') = \emptyset$ .

**Proof.** Let us begin with Property (1). First, if  $e \geq 1$ , the center of the tree has at least two neighbors (say,  $w_1$  and  $w_2$ ); otherwise, it would not be the sole center or a center at all. Now, let us show that the trees  $CS(w_1)$  and  $CS(w_2)$  have an equal height. If  $h(CS(w_1)) \leq h(CS(w_2)) - 2$ , then  $w_2$  has a smaller eccentricity than  $c$ , thus  $c$  cannot be a center. If  $h(CS(w_1)) = h(CS(w_2)) - 1$ , then  $c$  is not the sole center of the tree;  $w_2$  is another. Since the cases  $h(CS(w_2)) \leq h(CS(w_1)) - 2$  and  $h(CS(w_2)) = h(CS(w_1)) - 1$  are symmetric, it follows that  $h(CS(w_1)) = h(CS(w_2)) = e(w_1) = e(w_2) = e - 1$ . Let us pick vertices  $u' \in CS(w_1)$  and  $v' \in CS(w_2)$  such that  $d(w_1, u') = d(w_2, v') = e - 1$  (see Figure 5, left). The distances  $d(c, u')$  and  $d(c, v')$  are both equal to  $e$ , and, since the trees  $CS(w_1)$  and  $CS(w_2)$  are disjoint, we also have  $\mathcal{E}(c, u') \cap \mathcal{E}(c, v') = \emptyset$ .

As for Property (2), note that  $h(CS(c_1)) = h(CS(c_2))$ : if  $h(CS(c_1))$  were smaller than  $h(CS(c_2))$ , then  $c_1$  could not be a center, and symmetrically for the case  $h(CS(c_2)) < h(CS(c_1))$ . Given that  $h(CS(c_1)) = h(CS(c_2)) = e - 1$ , there is a vertex  $u'$  at a distance  $e - 1$  from  $c_1$  and a vertex  $v'$  at the same distance from  $c_2$  (see Figure 5, right). Since the trees  $CS(c_1)$  and  $CS(c_2)$  are disjoint,  $\mathcal{E}(c_1, u') \cap \mathcal{E}(c_2, v') = \emptyset$ .  $\square$

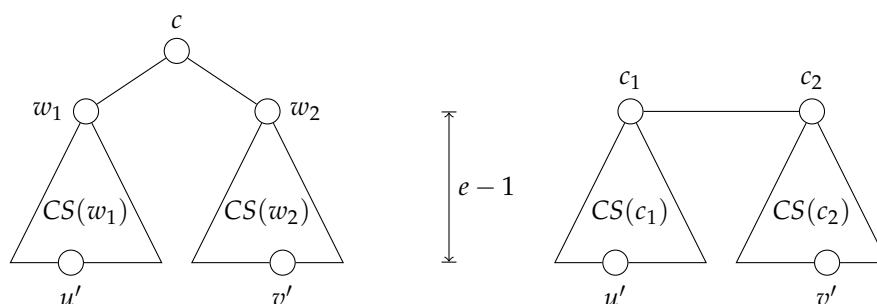


Figure 5. An illustration of Lemma 4: (Left) Property (1); and (Right) Property (2).

**Lemma 5.** If vertices  $u$  and  $v$  belong to the same orbit of the given tree, then all of the following hold:

- (1) If the tree has a single center, then  $d(u, c) = d(v, c)$ .
- (2) If the tree has two distinct centers, then  $d(u, c_1) = d(v, c_1)$  if  $d(u, v)$  is even and  $d(u, c_1) = d(v, c_2)$  if  $d(u, v)$  is odd.

$$(3) \quad CS(u) \simeq CS(v).$$

**Proof.** For trees consisting of at most two vertices, the lemma can easily be verified. We may thus assume that the tree has at least three vertices.

- (1) By Lemma 4, a tree with a single center  $c$  contains distinct vertices  $u'$  and  $v'$  such that  $d(c, u') = d(c, v') = e$  and the paths from  $c$  to  $u'$  and from  $c$  to  $v'$  are disjoint. Without loss of generality, we may assume that  $d(u, c) < d(v, c)$  and that the vertex  $v$  does not lie on the path from  $c$  to  $u'$  (if it does, we can swap  $u'$  and  $v'$ ). The distance from  $v$  to  $u'$  is  $D = d(v, c) + d(c, u') = d(v, c) + e$ . However, the most distant vertex from  $u$  can be at most  $d(u, c) + e \leq D - 1$  edges away from  $u$ . Therefore, the tree  $NS_D(v)$  has more vertices than  $NS_{D-1}(v)$ , but the tree  $NS_D(u)$  is the same as  $NS_{D-1}(u)$ . Consequently,  $u$  and  $v$  cannot be automorphically mapped to each other and hence cannot belong to the same orbit.
- (2) Lemma 4 ensures the existence of vertices  $u'$  and  $v'$  such that  $d(c_1, u') = d(c_2, v') = e - 1$  and the paths from  $c_1$  to  $u'$  and from  $c_2$  to  $v'$  are disjoint. Let the vertices  $u$  and  $v$  belong to the same orbit, and let us assume that  $d(u, v)$  is even. If  $u$  and  $v$  belong to the centrifugal subtrees of different centers (say,  $u$  belongs to  $CS(c_1)$  and  $v$  to  $CS(c_2)$ ), then  $d(u, c_1) \neq d(v, c_2)$ , contradicting our assumption that  $u$  and  $v$  are both part of the same orbit:  $d(u, c_1) < d(v, c_2)$  implies that the vertex  $u'$  is located  $d(v, c_2) + e$  edges away from  $v$  but there is no such vertex at the same distance from  $u$ , and a symmetrical conclusion follows from  $d(u, c_1) > d(v, c_2)$ . Therefore, an even value of  $d(u, v)$  means that  $u$  and  $v$  both belong to either  $CS(c_1)$  or  $CS(c_2)$  and (by applying a distance-based argument once again) that  $d(u, c_1) = d(v, c_1)$ . In a similar fashion, we can show that an odd value of  $d(u, v)$  means that  $u$  belongs to  $CS(c_1)$  and  $v$  to  $CS(c_2)$  (or the other way around) and that  $d(u, c_1) = d(v, c_2)$ .
- (3) If the subtrees  $CS(u)$  and  $CS(v)$  are non-isomorphic, there exists some distance  $d$  at which the neighborhood subtrees of  $u$  and  $v$  are non-isomorphic. Therefore, no automorphism can map  $u$  to  $v$  and vice versa, and so these two vertices do not belong to the same orbit.

□

The following lemma specifies a sufficient condition for a vertex set partition to be exploratory equivalent.

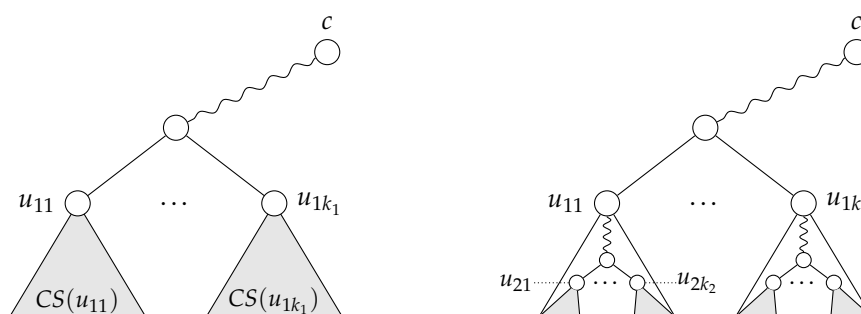
**Lemma 6.** An ordered partition  $\mathcal{P} = \langle P_1, \dots, P_s \rangle$  of the vertex set  $V$  of a given tree  $T$  is exploratory equivalent if all of the following conditions hold:

- (1) for each  $i \in \{1, \dots, s\}$ , all vertices in  $P_i$  belong to the same orbit of  $T$ ;
- (2) for each  $i \in \{2, \dots, s\}$  and for all pairs  $u, v \in P_i$ , we have  $d(u, v) = 2$ ;
- (3) either (i)  $d(u, v) = 2$  for all pairs  $u, v \in P_1$  or (ii)  $P_1 = \{c_1, c_2\}$  and  $c_1 \neq c_2$ ;
- (4) if  $i \neq j$  and there exists a vertex  $v \in P_i$  such that  $P_j \subseteq CS(v)$ , then  $j > i$ .

**Proof.** To verify that the partition  $\mathcal{P}$  is exploratory equivalent, we first have to show that the automorphism group of the tree,  $\text{Aut}(T)$ , covers the set  $P_1$ . By Lemma 5, the fact that all vertices in  $P_1$  belong to the same orbit implies that they are located at the same distance from the center(s) and that their centrifugal subtrees are isomorphic. If  $c_1 \neq c_2$ ,  $P_1 = \{c_1, c_2\}$ , and  $CS(c_1) \simeq CS(c_2)$ , then  $\text{Aut}(T)$  clearly covers  $P_1$ , since  $c_1$  can be automorphically mapped to  $c_2$  and vice versa. If, however, all vertices of  $P_1$  are located at the pairwise distance of 2, then they are connected with a common neighbor, in addition to having isomorphic centrifugal subtrees (see Figure 6, left). Consequently, the vertices of  $P_1$  can be automorphically mapped to each other in all possible ways, i.e., the set  $\text{Aut}(T)$  contains an automorphism for all  $|P_1|!$  permutations of  $P_1$ . In other words,  $\text{Aut}(T)$  covers  $P_1$ .

Now, let us pointwise-stabilize the set  $\text{Aut}(T)$  with respect to the set  $P_1$  and check whether the reduced set of automorphisms covers  $P_2$ . The pointwise stabilization can be interpreted as making the

vertices in  $P_1$  distinguishable (e.g., by assigning distinct colors or labels to them). After this operation, can the vertices of  $P_2$  still be automorphically mapped to each other? Condition (4) in the lemma allows for two possibilities: (i) the set  $P_2$  is disjoint from all centrifugal subtrees of the vertices in  $P_1$  (in which case the pointwise stabilization with respect to  $P_1$  has absolutely no impact on  $P_2$ ); and (ii) the set  $P_2$  is part of the centrifugal subtree of some vertex in  $P_1$ . In the latter case, as depicted in Figure 6 (right), the vertices of  $P_2$  can still be automorphically mapped to each other in every possible way, since they are connected to the same neighbor and since their centrifugal subtrees remain unaffected by the pointwise stabilization. In the same way, we can argue that the set  $P_3$  remains covered after pointwise-stabilizing the current set of automorphisms with respect to the set  $P_2$ . Since the same logic applies all the way to  $P_s$ , we can conclude that the partition  $\mathcal{P}$  is exploratory equivalent.  $\square$



**Figure 6.** An illustration of the proof of Lemma 6: **(Left)**  $\text{Aut}(T)$  covers  $P_1 = \{u_{11}, \dots, u_{1k_1}\}$  provided that  $\text{CS}(u_{11}) \simeq \dots \simeq \text{CS}(u_{1k_1})$ ; and **(Right)**  $\text{Aut}(T)$  covers  $P_2 = \{u_{21}, \dots, u_{2k_2}\}$  even after  $\text{Aut}(T)$  is pointwise-stabilized with respect to  $P_1$ .

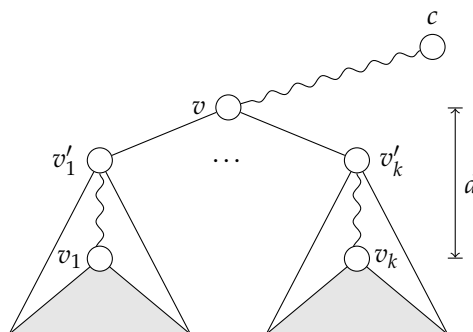
The next lemma forms the basis for an algorithm to find a maximum EE partition of a given tree.

**Theorem 4.** For any tree, there exists a maximum EE partition that satisfies Conditions (1)–(3) from Lemma 6.

**Proof.** Every EE partition has to satisfy Condition (1); if vertices  $v_1, \dots, v_k$  do not belong to the same orbit, they cannot be automorphically mapped to each other and hence cannot possibly constitute a class in an EE partition.

As for Conditions (2) and (3), let  $P = \{v_1, \dots, v_k\}$  be a class in a maximum EE partition  $\mathcal{P}$ , and let  $d(v_i, v_j) > 2$  for some  $i$  and  $j$ . For the time being, let us assume that the tree has only one center, i.e.,  $c = c_1 = c_2$ . Since the vertices in  $P$  belong to the same orbit, Lemma 5 ensures that  $d(v_1, c) = \dots = d(v_k, c)$  and  $\text{CS}(v_1) \simeq \dots \simeq \text{CS}(v_k)$ . Now, let  $v$  be the vertex with the property  $d(v_1, v) = \dots = d(v_k, v) = d$  that minimizes the distance  $d$ , and let  $v'_i$ , for  $i \in \{1, \dots, k\}$ , be the neighbor of  $v$  on the path from  $v$  to  $v_i$  (Figure 7). We claim that the partition  $\mathcal{P}'$  obtained by replacing the set  $P$  with the set  $P' = \{v'_1, \dots, v'_k\}$  (and adding the vertices  $v_1, \dots, v_k$  as singletons) is still a maximum EE partition. First, if the automorphism group of the tree covers  $P$ , it also covers  $P'$ , since an automorphism that maps  $v_i$  to  $v_j$  maps the entire centrifugal subtree of  $v'_i$  to that of  $v'_j$  (and  $v'_i$  itself to  $v'_j$ ). Second, in the original partition ( $\mathcal{P}$ ), the vertices  $v'_1, \dots, v'_k$  had to occur as singletons; otherwise,  $\mathcal{P}$  would not be EE, since an automorphism mapping  $v'_i$  to  $v'_j$  also maps  $v_i$  to  $v_j$ . Therefore, the score of  $\mathcal{P}'$  is the same as that of  $\mathcal{P}$ , and if  $\mathcal{P}$  is maximum, so is  $\mathcal{P}'$ . To see that the partition  $\mathcal{P}'$  remains to be EE, consider that the replacement of the constituent class  $P$  with the set  $P'$  could rule out only classes composed of vertices on the paths from  $v'_1, \dots, v'_k$  to  $v_1, \dots, v_k$ . However, in the partition  $\mathcal{P}$ , these vertices could only occur as singletons, since an automorphism that interchanges  $v_i$  with  $v_j$  interchanges the entire path from  $v'_i$  to  $v_i$  with the path from  $v'_j$  to  $v_j$ . Therefore, every EE class containing vertices at a pairwise distance greater than 2 can be replaced by one where the pairwise distance is exactly 2.

If the tree has two distinct centers and if  $d = d(v_1, c_1) = \dots = d(v_k, c_1)$  is even, the set  $P' = \{v'_1, \dots, v'_k\}$  can be constructed from the set  $P = \{v_1, \dots, v_k\}$  in exactly the same way as in the single-center case. However, if  $d$  is odd, then  $P$  contains exactly two vertices (say,  $u$  and  $v$ ), and  $d(u, c_1) = d(v, c_2)$ . In a similar manner as above, we can replace the class  $P = \{u, v\}$  with the class  $P' = \{c_1, c_2\}$  in the partition  $\mathcal{P}$ , and the resulting partition is still a maximum EE partition.  $\square$



**Figure 7.** An illustration of the situation in which the class  $\{v_1, \dots, v_k\}$  in an EE partition can be replaced by the class  $\{v'_1, \dots, v'_k\}$ . It holds that  $CS(v_1) \simeq \dots \simeq CS(v_k)$  and consequently also  $CS(v'_1) \simeq \dots \simeq CS(v'_k)$ .

Incidentally, note that the classes in an unordered maximum EE partition that fulfills Conditions (1)–(3) from Lemma 6 can always be ordered in such a way that the resulting ordered partition fulfills Condition (4), too.

We have seen that the search for a maximum EE partition of a given tree can be restricted to those partitions where all constituent classes consist solely of vertices at a pairwise distance of at most 2. Since every class in an EE partition has to be a subset of some orbit, an algorithm for finding a maximum EE partition can easily be devised: start with the set of orbits of the given tree  $(O_1, \dots, O_r)$ , and partition each orbit  $O_i$  into sets  $P_{i1}, \dots, P_{ik_i}$  such that for each  $j \in \{1, \dots, k_i\}$  the distance between each pair of vertices in  $P_{ij}$  is at most 2. The sets  $P_{ij}$  jointly form a maximum EE partition of the input tree. This procedure is presented as Algorithm 5.

---

**Algorithm 5:** A polynomial-time algorithm to find a maximum EE partition for a tree  $T$ .

---

```

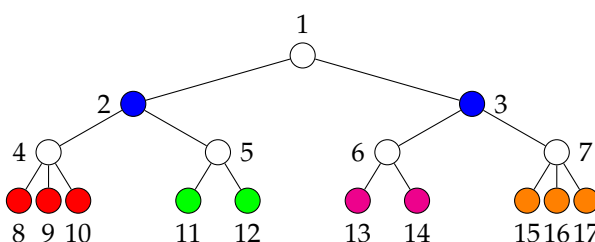
1  function MAXEETREE( $T$ )
2      Output: an optimal EE partion of tree  $T$ 
3       $O_1, \dots, O_r \leftarrow$  the orbits of  $T$ ;
4       $\mathcal{P} \leftarrow \emptyset$ ;
5      for  $i \in \{1, \dots, r\}$  do
6          while  $O_i \neq \emptyset$  do
7               $P \leftarrow \emptyset$ ;
8              while  $\exists v \in O_i: \forall u \in P: d(u, v) \leq 2$  do
9                   $P \leftarrow P \cup \{v\}$ ;
10                  $O_i \leftarrow O_i \setminus \{v\}$ ;
11             end
12              $\mathcal{P} \leftarrow \mathcal{P} \cup \{P\}$ ;
13         end
14     end
15     return  $\mathcal{P}$ 

```

---

The set of orbits of the input tree can be produced in time  $O(n)$  [42]. The rest of the algorithm can be performed in time  $O(n^2)$ ; in the worst case, the algorithm has to determine for each pair of tree vertices whether the distance between them is equal to 2. Therefore, the entire algorithm runs in time  $O(n^2)$ .

For example, the tree in Figure 8 has the following orbits:  $O_1 = \{1\}$ ,  $O_2 = \{2, 3\}$ ,  $O_3 = \{4, 7\}$ ,  $O_4 = \{5, 6\}$ ,  $O_5 = \{8, 9, 10, 15, 16, 17\}$ , and  $O_6 = \{11, 12, 13, 14\}$ . Algorithm 5 splits the orbits  $O_3$ ,  $O_4$ ,  $O_5$ , and  $O_6$ , each into two sets. The resulting maximum EE partition is thus  $\{1 \mid 2, 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8, 9, 10 \mid 11, 12 \mid 13, 14 \mid 15, 16, 17\}$ .



**Figure 8.** The maximum EE partition produced by Algorithm 5 for a sample tree. Different equivalence classes are highlighted by different colors. The uncolored vertices (1, 4, 5, 6, and 7) occur in the maximum EE partition as singletons.

### 7.3. Exact Algorithm on Cycles

Cycles deserve special attention because they represent a frequent counterexample for the proposed heuristic algorithm (Section 7.1) and because they belong to a class of graphs for which maximum EE partitions do not capture the entire set of automorphisms (Section 9). Nevertheless, it is possible to identify a set of rules that determine a maximum EE partition for any given cycle. In what follows, the  $n$ -cycle (with  $n \geq 3$ ) is a graph composed of vertices  $1, 2, \dots, n$  and edges  $(1, 2), (2, 3), \dots, (n-1, n), (n, 1)$ . Note that the automorphism group of the  $n$ -cycle comprises the automorphisms  $(i, i+1, \dots, n, 1, 2, \dots, i-1)$  and  $(i, i-1, \dots, 1, n, n-1, \dots, i+1)$  for all  $i \in \{1, \dots, n\}$ . A cycle with an even (odd) number of vertices is also called an *even cycle* (an *odd cycle*, respectively).

To simplify notation in the following proofs, let  $m_0(k) = k \bmod n$ , and let

$$m(k) = \begin{cases} m_0(k) + n & \text{if } m_0(k) < 0, \\ n & \text{if } m_0(k) = 0, \\ m_0(k) & \text{if } m_0(k) > 0 \end{cases} \quad (20)$$

for a fixed positive integer  $n$  and an arbitrary integer  $k$ . Note that  $m(k) = k$  for  $k \in \{1, \dots, n\}$  and  $m(k + in) = m(k)$  for an arbitrary integer  $i$ .

**Lemma 7.** In an odd  $n$ -cycle, an automorphism that swaps the vertices  $u$  and  $v$  fixes the vertex  $(u + v) / 2$  if  $u + v$  is even and  $m((u + v + n) / 2)$  if  $u + v$  is odd.

**Proof.** Without loss of generality, we may assume that  $u < v$ . If  $u + v$  is even, the path  $\langle u, m(u+1), m(u+2), \dots, m(v-2), m(v-1), v \rangle$  contains an odd number of vertices. Consequently, an automorphism that swaps the vertices  $u$  and  $v$  also swaps the vertices  $m(u+i)$  and  $m(v-i)$  for an arbitrary  $i$ . The vertex  $m(u + (v-u)/2) = m(v - (v-u)/2) = m((u+v)/2) = (u+v)/2$ , which lies in the middle of the path, is swapped with itself, i.e., kept fixed.

If  $u < v$  and  $u + v$  is odd, the path  $\langle u, m(u-1), m(u-2), \dots, m(v+2), m(v+1), v \rangle$  (e.g.,  $\langle 3, 2, 1, 7, 6 \rangle$  if  $n = 7, u = 3$ , and  $v = 6$ ) contains an odd number of vertices. An automorphism that swaps  $u$  and  $v$  also swaps the vertices  $m(u-i)$  and  $m(v+i)$  for an arbitrary  $i$ . For  $i = (u-v-n)/2$ , we obtain  $m(u-i) = m(v+i) = m((u+v+n)/2)$ , thus this vertex is fixed.  $\square$

**Lemma 8.** In an even  $n$ -cycle, an automorphism that swaps the vertices  $u$  and  $v$  fixes no vertices if  $u + v$  is odd and the vertices  $(u + v) / 2$  and  $m((u + v + n) / 2)$  if  $u + v$  is even.

**Proof.** As in the proof of Lemma 7, let  $u < v$ . If  $u + v$  is odd, then both the path  $\langle u, m(u + 1), \dots, m(v - 1), v \rangle$  and the path  $\langle u, m(u - 1), \dots, m(v + 1), m(v) \rangle$  contain an even number of vertices. In this case, swapping the vertices  $u$  and  $v$  does not fix any vertex; we cannot choose an integer  $i$  such that  $m(u + i) = m(v - i)$ . However, if  $u + v$  is even, both paths contain an odd number of vertices. The automorphism fixes the midpoints of the two paths, i.e., the vertices  $(u + v) / 2$  and  $m((u + v + n) / 2)$ .  $\square$

**Lemma 9.** A maximum EE partition of an odd cycle contains exactly one nonsingleton class, and that class consists of at most three vertices.

**Proof.** By Lemma 7, an automorphism that swaps a pair of vertices  $u$  and  $v$  fixes exactly one vertex. Consequently, a maximum EE partition contains at most one nonsingleton class, and that class consists of at most three vertices: the vertices  $u$  and  $v$  and, as shown below, in some cases also the vertex fixed by the automorphism that swaps  $u$  and  $v$ . However, in any cycle, any two vertices can be automorphically mapped to each other, which means that a maximum EE partition contains exactly one nonsingleton class.  $\square$

**Lemma 10.** In any  $(3k)$ -cycle, the partition  $\mathcal{P}_3 = \{k, 2k, 3k \mid \text{singletons}\}$  is exploratory equivalent. (If a partition of a set  $P$  is written as  $\mathcal{P} = \{P_1 \mid \dots \mid P_s \mid \text{singletons}\}$ , then all elements from the set  $P \setminus (P_1 \cup \dots \cup P_s)$  occur in  $\mathcal{P}$  as singletons.)

**Proof.** The partition  $\mathcal{P}_3$  is EE because the automorphism group of the  $(3k)$ -cycle covers all  $3!$  permutations of the set  $\{k, 2k, 3k\}$ , as shown in Table 1.  $\square$

**Table 1.** The cover of all  $3!$  permutations of the set  $\{k, 2k, 3k\}$ .

(	1,	...	$k - 1,$	<b><math>k,</math></b>	...	$2k - 1,$	<b><math>2k,</math></b>	...	$3k - 1,$	<b><math>3k</math></b>	)
(	$2k - 1,$	...	$k + 1,$	<b><math>k,</math></b>	...	1,	<b><math>3k,</math></b>	$3k - 1,$	...	<b><math>2k</math></b>	)
(	$3k - 1,$	...	$2k + 1,$	<b><math>2k,</math></b>	...	$k + 1,$	<b><math>k,</math></b>	...	1,	<b><math>3k</math></b>	)
(	$k + 1,$	...	$2k - 1,$	<b><math>2k,</math></b>	...	$3k - 1,$	<b><math>3k,</math></b>	1,	...	<b><math>k</math></b>	)
(	$2k + 1,$	...	$3k - 1,$	<b><math>3k,</math></b>	1,	...	<b><math>k,</math></b>	...	$2k - 1,$	<b><math>2k</math></b>	)
(	$k - 1,$	...	1,	<b><math>3k,</math></b>	...	$2k + 1,$	<b><math>2k,</math></b>	...	$k + 1,$	<b><math>k</math></b>	)

**Lemma 11.** For an odd  $n$ -cycle where  $n$  is not divisible by 3, a maximum EE partition is  $\{1, 2 \mid \text{singletons}\}$ .

**Proof.** By Lemma 9, the sole nonsingleton set of a maximum EE partition for an odd cycle has at most three vertices. However, for an automorphism group to cover a set of three vertices (as is the case in the proof of Lemma 10), these vertices have to be evenly spaced along the cycle. This is only possible if the number of vertices of the cycle is divisible by 3.  $\square$

**Lemma 12.** For a  $(2k)$ -cycle, a maximum EE partition is  $\{1, 3 \mid 2, k + 2 \mid \text{singletons}\}$  if  $k$  is not divisible by 3 and  $\{2k / 3, 4k / 3, 6k / 3 \mid \text{singletons}\}$  if it is.

**Proof.** Since the sum of  $u = 1$  and  $v = 3$  is even, Lemma 8 guarantees that the automorphism that swaps the vertices  $u$  and  $v$  fixes both the vertex  $w = (u + v) / 2 = 2$  and the vertex  $z = (u + v + 2k) / 2 = k + 2$ . The partition  $\{1, 3 \mid 2, k + 2 \mid \text{singletons}\}$  is thus a candidate EE partition, and it can be readily verified that it is indeed EE: by pointwise-stabilizing the initial set of automorphisms with respect to the set  $\{2, k + 2\}$ , we obtain a set of automorphisms that still covers the set  $\{1, 3\}$

(the corresponding ordered EE partition is hence  $\langle 2, k+2 \mid 1, 3 \mid \text{singletons} \rangle$ ). If  $k$  is not divisible by 3, this partition has the greatest possible score ( $2!2! = 4$ ). In the opposite case, the alternative EE partition  $\{2k/3, 4k/3, 6k/3 \mid \text{singletons}\}$  has a greater score and is hence maximum.  $\square$

The results given above can be summarized as follows:

**Theorem 5.** A maximum EE partition for an  $n$ -cycle is:

- $\{n/3, 2n/3, n \mid \text{singletons}\}$  if  $n = 3k$ ;
- $\{1, 3 \mid 2, n/2 + 2 \mid \text{singletons}\}$  if  $n = 6k - 4$  or  $n = 6k - 2$ ; and
- $\{1, 2 \mid \text{singletons}\}$  if  $n = 6k - 5$  or  $n = 6k - 1$ .

## 8. Application to the Subgraph Isomorphism Problem

As mentioned in the first paragraph of the Introduction, our approach was initially inspired by the subgraph isomorphism problem, which seeks to identify the occurrences of a pattern graph  $G$  in a host graph  $H$ . This problem can be readily applied to fields such as chemistry, where one may look for specific functional groups within a large molecule, or network analysis, where searching for particular community patterns might be of interest. Given the ever-increasing number, size, and impact of social networks in today's world, we can be sure that the applicability of the subgraph isomorphism search problem may only grow.

In this section, we consider the version of the subgraph isomorphism problem in which the goal is to enumerate *all* occurrences of a pattern graph in a host graph. As mentioned in Section 4, the subgraph isomorphism problem is a special case of a more general monomorphism search problem. Therefore, if the pattern graph has at least one non-identity automorphism and hence at least one nonsingleton class in its maximum exploratory equivalent partition, we can use Algorithm 1 to speed up the search.

Algorithm 1 is independent of the subgraph isomorphism search method. It only assumes that the search method constructs (partial and total) monomorphisms between the pattern graph and the host graph (which, by definition, is what every search algorithm does) but does not require anything else. The search algorithm may produce monomorphisms in any order.

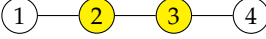
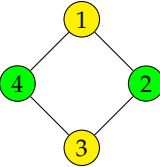
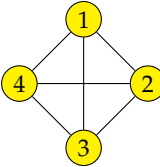
We applied Algorithm 1 to a search method based on *search plans* [2,43]. Search plans were used, for instance, to perform subgraph isomorphism search in the graph grammar parser of Rekers and Schürr [2,3]. A search plan of a given pattern graph  $G$  is a sequence of instructions for a systematic traversal of the vertices and edges of an occurrence of  $G$  in an arbitrary host graph  $H$ . The first instruction of a search plan always takes the form  $[u]$  and is read as “pick an arbitrary vertex  $w \in V_H$  and map it to the vertex  $u \in V_G$  (i.e., establish  $h(u) = w$ )”. All other instructions take either the form  $[u \rightarrow v]$  (“start at the vertex already mapped to  $u \in V_G$ , visit one of its as-yet unvisited neighbors together with the edge connecting the two vertices, and map the neighbor to the vertex  $v \in V_G$ ”) or the form  $[u ? v]$  (“verify the existence of an edge between the vertices already mapped to  $u$  and  $v$ , and visit that edge”). Note that any sequence of instructions that visits all vertices and edges constitutes a valid search plan.

A naive subgraph isomorphism search algorithm simply tries to carry out each instruction in all possible ways using backtracking. It thus finds all occurrences of  $G$  in  $H$ , but each occurrence is discovered  $|\text{Aut}(G)|$  times. To reduce the number of discoveries, we employ exploratory equivalence, as dictated by Algorithm 1. For each nonsingleton class  $\{u_1, \dots, u_k\}$  in the optimal EE partition  $\mathcal{P}$  of  $G$ , we impose the constraint  $h(u_1) < \dots < h(u_k)$ ; the search algorithm is allowed to produce and extend only those partial and total monomorphisms that satisfy this restriction. In this way, the algorithm will still find all occurrences of  $G$  in  $H$ , but the number of discoveries will be reduced by a factor of  $\text{score}(\mathcal{P})$ .

For instance, if we were searching for the occurrences of the graph  $C_4$  (Table 2) in a copy of itself, the naive algorithm would establish all eight monomorphisms (1234, 2341, 3412, 4123, 4321, 3214, 2143,

and 1432) ( $abcd$  represents the mappings  $h(1) = a$ ,  $h(2) = b$ ,  $h(3) = c$ , and  $h(4) = d$ ), whereas the EE-aware algorithm would only produce the monomorphisms 1234 and 2143, since these are the only two that satisfy the requirements  $h(1) < h(3)$  and  $h(2) < h(4)$  (note that the optimal EE partition of  $C_4$  is  $\{\{1, 3\}, \{2, 4\}\}$ ).

**Table 2.** Pattern graphs used in our experiments. In the same fashion as in Figure 8, different colors indicate different equivalence classes in the optimal EE partition, and the uncolored vertices constitute the singleton classes.

Pattern Graph ( $G$ )	$L_4$	$C_4$	$K_4$
Structure and maximum EE partition			
$ \text{Aut}(G) $	2	8	24
Score of max. EE part.	2	4	24
Search plan used in the experiments	$[1]$ $[1 \rightarrow 2]$ $[2 \rightarrow 3]$ $[3 \rightarrow 4]$	$[1]$ $[1 \rightarrow 2]$ $[2 \rightarrow 3]$ $[3 \rightarrow 4]$ $[1 ? 4]$	$[1]$ $[1 \rightarrow 2]$ $[2 \rightarrow 3]$ $[1 ? 3]$ $[3 \rightarrow 4]$ $[1 ? 4]$ $[2 ? 4]$

To see how this redundancy reduction technique manifests itself in practice, we compared the naïve search algorithm with its EE-aware version using the graphs  $L_4$ ,  $C_4$ , and  $K_4$  in Table 2 as the pattern graph set and six real-world graphs drawn from the datasets KONECT [44] and SNAP [45] as the host graph set. Each host graph was converted to an unlabeled simple undirected graph by removing possible labels, loops, and parallel edges and by treating directed edges as undirected.

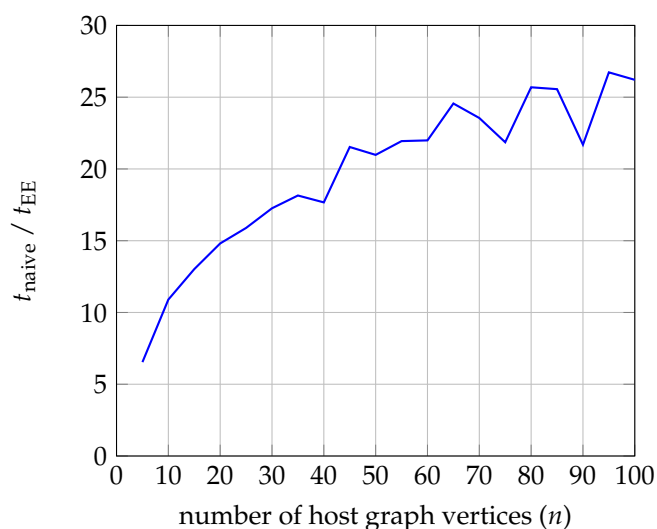
The experimental results are gathered in Table 3. Therein,  $N$  represents the number of occurrences of the corresponding pattern graph ( $G$ ) in the corresponding host graph ( $H$ ), whereas  $t_{\text{naive}}$  and  $t_{\text{EE}}$  denote the time (in milliseconds) required to find all occurrences using the naïve version of the search algorithm and the version that employs EE-based constraints, respectively. The experiments were conducted on a 3.40 GHz 8-core Intel Core i7-3770 machine.

Since all three pattern graphs contain nontrivial automorphisms, the EE-aware algorithm is significantly faster in all cases. As expected, the difference is most pronounced when searching for the occurrences of the graph  $K_4$ . The optimal EE partition of this graph has a score of 24; the corresponding values for the graphs  $L_4$  and  $C_4$  are 2 and 4, respectively. Theoretically, these are the upper bounds for the ratio  $t_{\text{naive}} / t_{\text{EE}}$ . However, owing to idiosyncrasies associated with implementation, execution, and time measurements, the actual ratio may also be slightly larger.

In the case of  $L_4$  and  $C_4$ , the actual ratios are fairly close to the upper bounds. In the case of  $K_4$ , the ratio tends to increase as the number of occurrences grows. This trend has been confirmed by an experiment in which we searched for the occurrences of the graph  $K_4$  within the graph  $K_n$  (the full graph on  $n$  vertices) for  $n \in \{5, 10, 15, \dots, 100\}$ . Figure 9 displays a plot of the ratio  $t_{\text{naive}} / t_{\text{EE}}$  as a function of  $n$ . Note that the number of occurrences of  $K_4$  in  $K_n$  is  $\binom{n}{4}$ .

**Table 3.** Subgraph isomorphism search using a search-plan-based algorithm: comparing a naïve version and a version that employs exploratory equivalence. All the times are in milliseconds.

<i>H</i>	<i>G</i>	<i>N</i>	$t_{\text{naive}}$	$t_{\text{EE}}$	$t_{\text{naive}} / t_{\text{EE}}$
Les Misérables $ V  = 77$ $ E  = 254$	$L_4$	26,784	3.47	1.94	1.8
	$C_4$	2672	8.88	3.71	2.4
	$K_4$	639	7.62	0.77	9.9
US Power Grid $ V  = 4941$ $ E  = 6594$	$L_4$	52,556	13.1	8.49	1.5
	$C_4$	979	11.6	4.51	2.6
	$K_4$	90	6.63	2.36	2.8
David Copperfield $ V  = 112$ $ E  = 425$	$L_4$	61,254	7.86	4.53	1.7
	$C_4$	2579	9.36	2.92	3.2
	$K_4$	58	5.80	0.94	6.2
Jazz Musicians $ V  = 198$ $ E  = 2742$	$L_4$	3,850,915	483	256	1.9
	$C_4$	406,441	866	245	3.5
	$K_4$	78,442	667	42.6	15.7
ca-HepTh $ V  = 9877$ $ E  = 25,973$	$L_4$	4,207,311	545	305	1.8
	$C_4$	239,081	627	198	3.2
	$K_4$	65,592	364	35.8	10.2
ca-CondMat $ V  = 23,133$ $ E  = 93,439$	$L_4$	50,543,325	6390	3520	1.8
	$C_4$	1,505,383	9920	2570	3.9
	$K_4$	294,008	3720	266	14.0

**Figure 9.** The ratio  $t_{\text{naive}} / t_{\text{EE}}$  when searching for the occurrences of  $K_4$  in  $K_n$ .

## 9. Conclusions and Discussion

In this paper, we present a new view on graph symmetries in the form of a graph partition called exploratory equivalence. We show how such partitions can be used for symmetry breaking without imposing any other restrictions or overhead during the execution of backtracking algorithms. Since there are many possible exploratory equivalent partitions in a graph, we introduce the so-called MAXEE optimization problem, which captures the efficiency of symmetry breaking. We manage to position the verifier for the optimization problem between the graph isomorphism and set stabilizer problems. This makes the optimization problem not yet known to be in  $\mathcal{NP}$ . Despite this pessimistic result, we propose optimal polynomial-time algorithms for cycles and trees and a practical heuristic algorithm for general graphs.

As part of the presented results, we show that MAXEE does not capture all possible symmetries in graphs and also show examples of suboptimal results of the given heuristics. However, we also conducted a small experiment, which indicates that in practice MAXEE captures a vast majority of symmetries and also that the greedy algorithm produces optimal results in most cases.

The settings of the experiment are as follows. We generated all connected (non-isomorphic) graphs of sizes 4–10 using geng [46]. For each of them, we computed the number of automorphisms (the number of all symmetries), the optimal solution to MAXEE using backtracking search, and the solution given by the greedy heuristics. Table 4 summarizes these results. For each size (4–10), we show the cumulative number of all symmetries (sym.), the number of symmetries captured by MAXEE (opt.), and the number of symmetries captured by the greedy heuristics (greedy). We have to point out that finding the optimal solution for all graphs of size 10 took a substantial computational effort. We used 77 computers (off-the-shelf personal computers), and it took us nearly one day to obtain the results. On the other hand, the heuristic method was run on a single computer, and the results were obtained in a couple of hours. This is a simple indication of the practical feasibility of the heuristic method.

**Table 4.** Empirical evaluation of MAXEE on the set of all connected simple graphs of sizes 4–10 nodes. We counted the cumulative number of all symmetries on these sets (sym.), the sum of all optimal goal values for MAXEE (opt.), and the sum of all goal values obtained by the greedy heuristics (greedy).

Size	# of Graphs	sym.	opt.	$\frac{\text{opt.}}{\text{sym.}}$	Greedy	$\frac{\text{greedy}}{\text{opt.}}$
4	6	46	42	0.91	42	1.0
5	21	242	226	0.93	226	1.0
6	112	1650	1522	0.92	1490	0.97
7	853	11,338	10,910	0.96	10,850	0.99
8	11,117	100,648	96,896	0.96	96,588	0.99
9	261,080	1,154,556	1,133,514	0.98	1,131,216	0.99
10	11,716,571	24,724,920	24,529,766	0.99	24,515,292	0.99

In this simple experiment, two observations can be made. The first is that, even though MAXEE does not capture all the symmetries present in graphs, it does capture a huge percentage of them (see  $\frac{\text{opt.}}{\text{sym.}}$  column in Table 4). The second observation is the quality of the greedy algorithm, which on average produces nearly optimal results (see  $\frac{\text{greedy}}{\text{opt.}}$  column in Table 4). The percentage of captured symmetries increases with the size of the graphs, which also means that the optimal solution captures more than 99% (on average) of symmetries in the graphs. These results are very encouraging, demonstrating the practical feasibility of the methods presented in this paper.

From our entire work and the empirical feature of subgraph isomorphism, we can summarize the properties of our proposed method as follows:

- The approach is suitable when mapping small graphs onto large graphs, since the overhead of searching for good partitions is minimal. Such application arise often in the field of bioinformatics [47] and in many applications that use graph databases [48].
- Since it was empirically established in [49], symmetries arise surprisingly often in real graphs, which makes it a compelling argument to finally introduce a symmetry-breaking method into state-of-the-art solvers.
- As mentioned above, one of the goals to design a lightweight mechanism, which would induce a minimal overhead in the backtracking algorithm. We achieved this goal, since after the exploratory equivalent partition is detected (as part of the preprocessing), the backtracking algorithm just utilizes the induced constraints which follow from the partition.
- As a downside, the method might become unfeasible when very large graphs are being mapped. However, in such cases, other methods might also fail to find solutions in reasonable time.

In our future work, we will incorporate our equivalence into state-of-the-art solvers and conduct extensive testing on a variety of benchmark problems. This will show us the empirical usability of the presented concept and finally establish symmetry-breaking as a standard component of subgraph isomorphism solvers. Since our framework is designed for a wide variety of problems, we intend to integrate exploratory equivalence into some of these as well, e.g., largest clique problem, tree-width, etc. In the more theoretical part of our future work, we will investigate the time complexity of the MAXEE problem further and design new heuristics for finding exploratory equivalent partitions even more quickly.

**Author Contributions:** All authors contributed equally to this work. All authors read and approved the final manuscript.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

EE	Exploratory Equivalence
GI	Graph Isomorphism
MAXEE	Maximum Exploratory Equivalence
MAXEEDEC	Decision version of MAXEE
VERIFYEE	problem of checking if a partition is EE

## References

1. Leach, A.R.; Gillet, V.J. *An Introduction to Chemoinformatics*; Springer: Berlin, Germany, 2007.
2. Rekers, J.; Schürr, A. Defining and Parsing Visual Languages with Layered Graph Grammars. *J. Vis. Lang. Comput.* **1997**, *8*, 27–55. [\[CrossRef\]](#)
3. Fürst, L.; Mernik, M.; Mahnič, V. Improving the graph grammar parser of Rekers and Schürr. *IET Softw.* **2011**, *5*, 246–261. [\[CrossRef\]](#)
4. Cook, S.A. The complexity of theorem-proving procedures. In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC), Shaker Heights, OH, USA, 3–5 May 1971; pp. 151–158. [\[CrossRef\]](#)
5. Arora, S.; Barak, B. *Computational Complexity: A Modern Approach*; Cambridge University Press: Cambridge, UK, 2009.
6. Fomin, F.V.; Kratsch, D. *Exact Exponential Algorithms*; Springer: Berlin, Germany, 2011.
7. Ullmann, J.R. An Algorithm for Subgraph Isomorphism. *J. Assoc. Comput. Mach.* **1976**, *23*, 31–42. [\[CrossRef\]](#)
8. Cordella, L.P.; Foggia, P.; Sansone, C.; Vento, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* **2004**, *26*, 1367–1372. [\[CrossRef\]](#)
9. Čibej, U.; Mihelič, J. Improvements to Ullmann’s Algorithm for the Subgraph Isomorphism Problem. *Int. J. Pattern Recognit. Artif. Intell.* **2015**, *29*, 1550025. [\[CrossRef\]](#)
10. Bonnici, V.; Giugno, R.; Pulvirenti, A.; Shasha, D.; Ferro, A. A subgraph isomorphism algorithm and its application to biochemical data. *BMC Bioinform.* **2013**, *14*, S13. [\[CrossRef\]](#)
11. Kotthoff, L.; McCreesh, C.; Solnon, C. Portfolios of Subgraph Isomorphism Algorithms. In *Learning and Intelligent Optimization*; Festa, P., Sellmann, M., Vanschoren, J., Eds.; Springer International Publishing: Cham, Switzerland, 2016; pp. 107–122.
12. Carletti, V.; Foggia, P.; Saggese, A.; Vento, M. Introducing VF3: A New Algorithm for Subgraph Isomorphism. In *Graph-Based Representations in Pattern Recognition*; Foggia, P., Liu, C.L., Vento, M., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 128–139.
13. McKay, B.D. Practical Graph Isomorphism. *Congr. Numer.* **1981**, *30*, 45–87. [\[CrossRef\]](#)
14. Babai, L. Graph Isomorphism in Quasipolynomial Time. In Proceedings of the forty-eighth annual ACM Symposium on Theory of Computing, Portland, OR, USA, 14–17 June 2015; pp. 684–697.
15. Albertson, M.O.; Collins, K.L. Symmetry breaking in graphs. *Electron. J. Combin.* **1996**, *3*, R18.
16. Cheng, C.T. On computing the distinguishing numbers of trees and forests. *Electron. J. Combin.* **2006**, *13*, R11.

17. Imrich, W.; Klavžar, S. Distinguishing Cartesian powers of graphs. *J. Graph Theory* **2006**, *53*, 250–260. [[CrossRef](#)]
18. Russell, A.; Sundaram, R. A note on the asymptotics and computational complexity of graph distinguishability. *Electron. J. Combin.* **1998**, *5*, R23.
19. Arvind, V.; Cheng, C.T.; Devanur, N.R. On computing the distinguishing numbers of planar graphs and beyond: A counting approach. *SIAM J. Discret. Math.* **2008**, *22*, 1297–1324. [[CrossRef](#)]
20. Everett, M.G.; Borgatti, S.P. Regular equivalence: General theory. *J. Math. Sociol.* **1994**, *19*, 29–52. [[CrossRef](#)]
21. Everett, M.G.; Borgatti, S.P. Computing Regular Equivalence: Practical and Theoretical Issues. *Metodološki Zvezki* **2002**, *17*, 31–42.
22. Sailer, L.D. Structural Equivalence: Meaning and Definition, Computation and Application. *Soc. Netw.* **1978**, *1*, 73–90. [[CrossRef](#)]
23. Knuth, D.E. *The Art of Computer Programming. Volume 4B. Combinatorial Algorithms: Part 2; The Art of Computer Programming*; Addison-Wesley Professional: Boston, MA, USA, 2016.
24. Junttila, T.; Karppa, M.; Kaski, P.; Kohonen, J. An Adaptive Prefix-Assignment Technique for Symmetry Reduction. In *Theory and Applications of Satisfiability Testing—SAT 2017*; Gaspers, S., Walsh, T., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 101–118.
25. Crawford, J.; Ginsberg, M.; Luks, E.; Roy, A. Symmetry-Breaking Predicates for Search Problems. In *Proceedings of the Fifth International Conference Principles of Knowledge Representation and Reasoning, (KR '96)*, Cambridge, MA, USA, 5–8 November 1996.
26. Margot, F. Symmetry in Integer Linear Programming. In *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*; Jünger, M., Liebling, M.T., Naddef, D., Nemhauser, L.G., Pulleyblank, R.W., Reinelt, G., Rinaldi, G., Wolsey, A.L., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 647–686.
27. Gent, I.P.; Petrie, K.E.; Puget, J.F. Symmetry in constraint programming. *Handbook of Constraint Programming*; Elsevier Science: Amsterdam, The Netherlands, 2006; Volume 10, pp. 329–376.
28. Petrie, K.E.; Smith, B.M. Comparison of symmetry breaking methods in constraint programming. *Proc. SymCon05* **2005**.
29. The GAP Group. *GAP—Groups, Algorithms, and Programming, Version 4.8.3*; The GAP Group: Glasgow, UK, 2016.
30. Gent, I.P.; Harvey, W.; Kelsey, T. Groups and constraints: Symmetry breaking during search. In *Principles and Practice of Constraint Programming—CP 2002*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 415–430.
31. Brown, C.A.; Finkelstein, L.; Purdom, P.W. Backtrack searching in the presence of symmetry. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, Proceedings of the 6th International Conference, AAEC-6, Rome, Italy, 4–8 July 1988*; Mora, T., Ed.; Springer: Berlin/Heidelberg, Germany, 1989; pp. 99–110. [[CrossRef](#)]
32. Puget, J.F. Symmetry Breaking Revisited. In *Principles and Practice of Constraint Programming—CP 2002, Proceedings of the 8th International Conference, CP 2002, Ithaca, NY, USA, 9–13 September 2002*; Hentenryck, P., Ed.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 446–461. [[CrossRef](#)]
33. Fahle, T.; Schamberger, S.; Sellmann, M. Symmetry Breaking. In *Principles and Practice of Constraint Programming—CP 2001, Proceedings of the 7th International Conference, CP 2001, Paphos, Cyprus, 26 November–1 December 2001*; Walsh, T., Ed.; Springer: Berlin/Heidelberg, Germany, 2001; pp. 93–107. [[CrossRef](#)]
34. Margot, F. Exploiting orbits in symmetric ILP. *Math. Program.* **2003**, *98*, 3–21. [[CrossRef](#)]
35. Bodlaender, H.L.; Fomin, F.V.; Koster, A.M.C.A.; Kratsch, D.; Thilikos, D.M. A Note on Exact Algorithms for Vertex Ordering Problems on Graphs. *Theory Comput. Syst.* **2011**, *50*, 420–432. [[CrossRef](#)]
36. Garey, M.R.; Graham, R.L.; Johnson, D.S.; Knuth, D.E. Complexity results for bandwidth minimization. *SIAM J. Appl. Math.* **1978**, *34*, 477–495. [[CrossRef](#)]
37. Bodlaender, H.L.; Fomin, F.V.; Koster, A.M.C.A.; Kratsch, D.; Thilikos, D.M. On Exact Algorithms for Treewidth. *ACM Trans. Algorithms* **2012**, *9*, 12:1–12:23. [[CrossRef](#)]
38. Luks, E.M. Permutation groups and polynomial-time computation. In *Groups and Computation: Workshop on Groups and Computation, October 7–10, 1991*; American Mathematical Society: Providence, RI, USA, 1993; Volume 11, p. 139.
39. Seress, Á. *Permutation Group Algorithms*; Cambridge Tracts in Mathematics, Cambridge University Press: Cambridge, MA, USA, 2003.
40. Arvind, V.; Torán, J. Isomorphism Testing: Perspective and Open Problems. *Bull. EATCS* **2005**, *86*, 66–84.

41. Knuth, D.E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed.; Addison-Wesley Professional: Boston, MA, USA, 1997.
42. Colbourn, C.J.; Booth, K.S. Linear Time Automorphism Algorithms for Trees, Interval Graphs, and Planar Graphs. *SIAM J. Comput.* **1981**, *10*, 203–225. [[CrossRef](#)]
43. Fürst, L.; Čibej, U.; Mihelič, J. Iskanje vzorčnih grafov s pomočjo iskalnega načrta ob prisotnosti avtomorfizmov [Searching for pattern graphs using a search plan in the presence of automorphisms]. *Elektrotehniški Vestnik* **2018**, *85*, 162–168.
44. Kunegis, J. KONECT: The Koblenz network collection. In Proceedings of the International Conference on World Wide Web Companion, Rio de Janeiro, Brazil, 13–17 May 2013; pp. 1343–1350. [[CrossRef](#)]
45. Leskovec, J.; Krevl, A. SNAP Datasets: Stanford Large Network Dataset Collection. 2014. Available online: <http://snap.stanford.edu/data> (accessed on 5 October 2019).
46. McKay, B.D.; Piperno, A. Practical graph isomorphism, II. *J. Symb. Comput.* **2014**, *60*, 94–112. [[CrossRef](#)]
47. Melckenbeeck, I.; Audenaert, P.; Colle, D.; Pickavet, M. Efficiently counting all orbits of graphlets of any order in a graph using autogenerated equations. *Bioinformatics* **2017**, *34*, 1372–1380. [[CrossRef](#)]
48. McCreesh, C.; Prosser, P.; Solnon, C.; Trimble, J. When subgraph isomorphism is really hard, and why this matters for graph databases. *J. Artif. Intell. Res.* **2018**, *61*, 723–759. [[CrossRef](#)]
49. Ball, F.; Geyer-Schulz, A. How Symmetric Are Real-World Graphs? A Large-Scale Study. *Symmetry* **2018**, *10*, 29. [[CrossRef](#)]



© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).