Received February 6, 2020, accepted February 18, 2020, date of publication February 27, 2020, date of current version March 10, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2976876

Efficient Enumeration of Higher Order Algebraic Structures

MAJID ALI KHAN, (Member, IEEE)
College of Computer Engineering and Science, Prince Mohammad Bin Fahd University, Khobar 31952, Saudi Arabia
e-mail: makhan@pmu.edu.sa

This work was partially supported by Prince Mohammad Bin Fahd University (PMU) internal research grant.

ABSTRACT Algebraic structures are widely studied mathematical structures in abstract algebra. Enumerating higher order algebraic structures is a computationally intensive task due to large number of possible permutations and the presence of many symmetrically equivalent redundant structures. This paper describes a comprehensive methodology and a novel algorithm to efficiently enumerate higher order algebraic structures. Enumeration of these structures is performed in two steps, namely: generation of complete set of algebraic structures with given constraints, and then computationally demanding task of performing isomorphism checking to identify isomorphism classes. In this paper we choose to study inverse property loops (IP loops), a particular class of algebraic structures, but the methodology can be applied to enumerate any algebraic structure with given constraints. IP loop constraints are modeled in Google's *or-tools* constraint solver to generate the complete set of IP loops of given order. The paper then discusses and evaluates several techniques to efficiently identify isomorphism classes within these algebraic structures. A novel algorithm is proposed that utilizes valid mappings and a tree data structure for efficient isomorphism checking. The algebraic structures are also modeled as color graphs and isomorphism classes are determined using state of the art graph isomorphism checking tool (*nauty*). The performance of these proposed isomorphism checking approaches is then evaluated using a diverse set of algebraic structure problems. It also presents an efficient use of multi-core systems to further enhance the efficiency of isomorphism checking. The proposed approach was then used to solve the previously computationally unsolved problem of determining isomorphism classes of exponent 3 IP loops of order 15.

INDEX TERMS Algebraic structures, constraint solver, IP loops, isomorphism checking, isomorphism classes, nauty.

I. INTRODUCTION

Algebraic structures are of interest to mathematicians because of their special properties and their practical applications in different areas such as formation of statistical designs, construction of error codes, cryptography, etc [1]–[3]. In abstract algebra, an algebraic structure refers to a set with one or more finite operations defined on it that satisfies a list of axioms. Quasigroups, groups, loops and IP loops are some of the widely studied algebraic structures [4]. A *quasigroup* $(Q, *)$ is a groupoid Q with a binary operation $*$ such that for each $x, y \in Q$, $x * a = y$ and $b * x = y$ have unique solutions. In other words, the main difference between a quasigroup and a group is that quasigroup has no requirement of associativity. The multiplication table of a finite quasigroup is called a *Latin square*. A quasigroup with an identity element e such that for each $x \in Q$, $x * e = x = e * x$ is called a *loop*. An *inverse property (IP) loop* is a loop L if it has a two-sided inverse x^{-1} such that $x^{-1} * (x * y) = y = (y * x) * x^{-1}$ for each $x, y \in L$.

Researchers had interest in counting and enumerating algebraic structures for over three centuries. The On-line Encyclopedia of Integer Sequences (OEIS) maintains a record of currently known counts of algebraic structures such as Latin squares [36] and loops [37]. Earliest history of counting Latin squares (LS) goes back to at least 1782 as the number of reduced LS of order 5 was known to Euler [15] and Cayley [14]. Since that time, researchers have been trying to get the next order algebraic structures [4], [11], [16], [19], [24], [25]. Despite their interest in algebraic structures, there has been considerable delay in achieving consecutive milestones. This was because of sheer computational

The associate editor coordinating the review of this manuscript and approving it for publication was Fatos Xhafa.

complexity of the problem. There are numerous other studies on counting algebraic structures which have reported incorrect counts [13], [17], [18], [20]. These historical results were obtained through deduced mathematical formulas, applying algorithmic approaches or formulating them as constraint programming problems [7], [8], [11], [24], [25]. In this paper we use constraint programming and isomorphism checking techniques to enumerate algebraic structures.

Algebraic structures of any order (n) can be enumerated and counted by modeling them as a finite domain constraint satisfaction problem (CSP). One or more relevant constraints need to be applied on CSP variables to enumerate required algebraic structures. Table 1 shows the constraints for Latin square, loop and IP loop properties. Based on specified CSP constraints the constraint solver efficiently explores the state space and finds all possible solutions that satisfy the required algebraic structure properties.

TABLE 1. Few example algebraic structures and corresponding constraints.

Name	Constraint
Latin square	$\forall row : \forall i,j \in row, x_i = x_j \Rightarrow i = j$ $\forall col : \forall i,j \in col, y_i = y_j \Rightarrow i = j$
Loop	$x * e = x = e * x$
IP loop	$\forall x,y \in L : x^{-1} * (x * y) = (y * x) * x^{-1} = y$
Basic symmetry breaking in IP loop	$\lvert x - x^{-1} \rvert \leq 1$
Exponent 3	$\forall x : (x * x) * x = e = x * (x * x)$

Algebraic structures obtained from constraint solvers have symmetries [5], [6]. In other words, each obtained solution may have several equivalent solutions. For example, there are 161280 Latin squares of order 5, of which only 1411 isomorphism classes exist [4]. As the order of algebraic structure (n) becomes higher the number of symmetries increases enormously. Practically it is sufficient to find only one solution from each class of equivalent solutions. The enumeration time of constraint solvers is greatly reduced if these symmetries are eliminated during the search itself. In [7] some symmetric breaking constraints for IP loops are proposed. Our previous work [21] proposed a mining-based approach to identify additional such constraints. Symmetric breaking constraints such as those suggested in [7], [21] remove many redundant solutions. However, even after applying symmetric breaking constraints the solutions generated by the constraint solver have enormous number of symmetric copies. These redundant symmetric copies are then eliminated using a separate post-processing step in order to get the final unique solutions (isomorphism classes). This paper proposes a novel algorithm to efficiently enumerate algebraic structures in this post-processing step. A previous work [23] has also listed IP loop algebraic structures of order 7, 9, 11 and 13 labelled with their corresponding isomorphism classes.

Unfortunately, most of the earlier work on enumeration of algebraic structures does not provide any details about the performance characteristics of the system. [12] briefly mentions that IP Loop of order 13 were enumerated on an ordinary desktop system in less than a day. In comparison, our proposed algorithm can enumerate IP loop of order 13 in six hours (21952 seconds) on an ordinary desktop system. Due to the lack of prior performance characteristics, we have chosen to compare our work with *nauty*, a state of the art isomorphism checking tool [9].

This paper presents a novel algorithm to efficiently identify the isomorphism classes of algebraic structures by eliminating redundant copies. It also describes the mechanism for modeling the algebraic structures as color graphs and then use state of the art graph isomorphism tool (*nauty*) to identify the isomorphism classes. The paper then evaluates the results obtained using a comparative analysis of the proposed isomorphism checking approaches for a diverse range of algebraic structure problems. It also presents an approach to efficiently utilize multi-core systems to further enhance the performance of isomorphism checking. The proposed parallel implementation is then used to solve the previously unsolved problem of determining isomorphism classes of exponent 3 IP loops of order 15 [22].

The rest of the paper is organized as follows. Section II describes the related background information for constraint programming, isomorphism classes and graph isomorphism. Section III explains the overall methodology used to efficiently generate and then determine isomorphism classes of algebraic structures. It also discusses several isomorphism checking approaches including our novel algorithm (TVMC) and the details of modeling algebraic structures as color graphs to apply graph isomorphism approach and its parallel implementation. Section IV shows the results of the comparative study conducted to evaluate the isomorphism checking approaches on a diverse range of problems. It also evaluates the results of using a parallel implementation on multi-core systems with different hardware architectures. Section V concludes the paper with possible future research directions.

II. BACKGROUND

This section briefly describes the background information about constraint solvers, isomorphism classes and graph isomorphism.

A. CONSTRAINT PROGRAMMING

Constraint programming (CP) is a paradigm where constraints are used to capture desired properties in the solution. These constraints represent relations between variables of the system. Constraint programming aims at finding feasible solutions to the problem that satisfies all the given constraints. CP has been applied in several domains including computer graphics, natural language processing, scheduling, and planning. There are several free and commercial constraint solvers available which allow users to model problems as finite domain constraint satisfaction problem (CSP) such as *JaCoP* [34], *GECODE* [35] and Google's *or-tools* [32]. The enumeration problem is modeled as CSP where the range of

*	0	1	2	3	4
0	2	1	0	3	4
1	4	3	2	0	1
2	1	2	3	4	0
3	3	0	4	1	2
4	0	4	1	2	3

(a) Latin Square

*	0	1	2	3	4
e=0	0	1	2	3	4
1	1	0	3	4	2
2	2	3	4	1	0
3	3	4	0	2	1
4	4	2	1	0	3

(b) Loop

*	0	1	2	3	4
e=0	0	1	2	3	4
1	1	3	0	4	2
2	2	0	4	1	3
3	3	4	1	2	0
4	4	2	3	0	1

x	x^{-1}
0	0
1	2
2	1
3	4
4	3

(c) IP Loop

FIGURE 1. **Examples of algebraic structures.**

the binary operation $*$ is a CSP variable whose domain consists of elements of the algebra. The properties of algebraic structures (Latin square, Loop, and IP loop) are captured in the form of constraints that are applied on CSP variables.

Table 1 shows few algebraic structures and the corresponding constraints. Latin square constraint implies that each symbol (element) is uniquely present in any row and in any column. An example of Latin square of order 5 is shown in Figure 1(a). The loop constraint enforces existence of identity element (e) such that a binary operation ($*$) between e and any other element (x) results in the same element (x). For example, in Figure 1(b) $e = 0$ and $(2 * 0) = (0 * 2) = 2$, whereas $(2 * 0) \neq (0 * 2) \neq 2$ in Latin square shown in Figure 1(a). The IP loop constraint implies existence of left and right inverses (x^{-1}) such that $x^{-1} * x = e$ and $x * x^{-1} = e$ and holds left inverse property ($x^{-1} * (x * y) = y$) and right inverse property ($(y * x) * x^{-1} = y$) for each element of the loop. For example in Figure 1(c), the inverse of element 1 is 2, and $2 * (1 * 3) = (3 * 1) * 2 = 3$. On the contrary in Figure 1(b), the inverse of element 1 is 1, but $1 * (1 * 3) \neq 3$ and $(3 * 1) * 1 \neq 3$. Additionally, symmetry breaking constraint suggests that the difference between any element to its inverse is not more than one. Finally, exponent 3 constraint implies that $x * x = x^{-1}$ for every element. For example $(8 * 8) = 7 = 8^{-1}$ in Figure 4, which shows two examples of exponent 3 IP loops of order 15.

We had initially considered *JaCoP* as the constraint solver due to its ease of use and our familiarity with *Java* Language. However, *JaCoP* did not scale quite well and resulted in out of memory errors for IP loop of order 13. We then switched to Google's *or-tools* which is one of the leading constraint solver and performed native computations for *Java* and other programming languages. It turned out to be good enough for our enumeration requirements. For example, it took about 12 minutes (732 seconds) to enumerate 7.8 million IP loops of order 13. The details of our constraints programming implementation using *or-tools* are provided in Section III-A.

B. ISOMORPHISM CLASSES

We call two algebraic structures (e.g. Latin squares, loops or IP loops) $(L_1, *_1)$ and $(L_2, *_2)$ of order n *isomorphic* to each other if there exists a bijective function $f : A \rightarrow B$, where $A = \{0 \ldots n-1\}$ and B is any permutation of A, such that for all indices u and v in L_1:$f(u *_1 v) = f(u) *_2 f(v)$. In our case, L_1 $(n \times n)$ is isomorphic to L_2 $(n \times n)$ if $\forall i, j < n$, $f(L_1[i][j]) = L_2[f(i)][f(j)]$. All those structures that are isomorphic to each other belong to one *isomorphism class*. For example, Figure 2 shows two IP loops L_1 and L_2, which look quite different from each other (as highlighted), but belong to the same isomorphism class because there exists a bijective function, $f : \{0, 1, 2, 3, 4, 5, 6\} \rightarrow \{0, 1, 2, 4, 3, 5, 6\}$ that satisfies the isomorphism property for each element of L_1 and L_2. Please note that $f(0) = 0, f(1) = 1, f(2) = 2, f(3) = 4, f(4) = 3, f(5) = 5$ and $f(6) = 6$. For example, it can be seen that at indices $(i, j) = (1, 3)$, isomorphism property is satisfied as $f(L_1[1][3]) = L_2[f(1)][f(3)] = 5$.

We can also represent the above bijective function f as $f : (3\ \ 4)$, which means that symbols 3 and 4 are swapped. Another way to check isomorphism between two algebraic structures L_1 and L_2 is to generate L_2 from L_1 by swapping particular rows, columns, and the values according to the function f. For example, in Figure 2, L_2 can be generated from L_1 by swapping rows 3 and 4, column 3 and 4, and values 3 and 4. Finding isomorphism in this way, by applying the above formula for all permutations of f is extremely time consuming and involves huge number of possibilities for even slightly large value of n.

0	1	2	3	4	5	6
1	2	0	5	6	4	3
2	0	1	6	5	3	4
3	6	5	4	0	1	2
4	5	6	0	3	2	1
5	3	4	2	1	6	0
6	4	3	1	2	0	5

$f : (3\ \ 4)$

$\approx$

0	1	2	3	4	5	6
1	2	0	6	5	3	4
2	0	1	5	6	4	3
3	5	6	4	0	2	1
4	6	5	0	3	1	2
5	4	3	1	2	6	0
6	3	4	2	1	0	5

FIGURE 2. **IP loop of order 7 (L_1 on the left and L_2 on the right) are isomorphic to each other.**

C. GRAPH ISOMORPHISM

The idea behind isomorphism is helpful in finding structural similarity between two graphs. Two graphs G_1 and G_2 are isomorphic ($G_1 \simeq G_2$) if there exists a bijection between the vertex sets of G_1 and $G_2, f : V(G_1) \rightarrow V(G_2)$ such that any two vertices v and v' in G_1 are adjacent if and only if $f(v)$ and $f(v')$ are adjacent in G_2. The graph isomorphism problem is a computational problem of determining whether two finite graphs are isomorphic. It is used in a variety of applications such as in molecular science [26], [27], the design of electronic circuits [28]–[30], automata theory [31] etc. *nauty* [9] is one of the most efficient graph isomorphism checking tools.

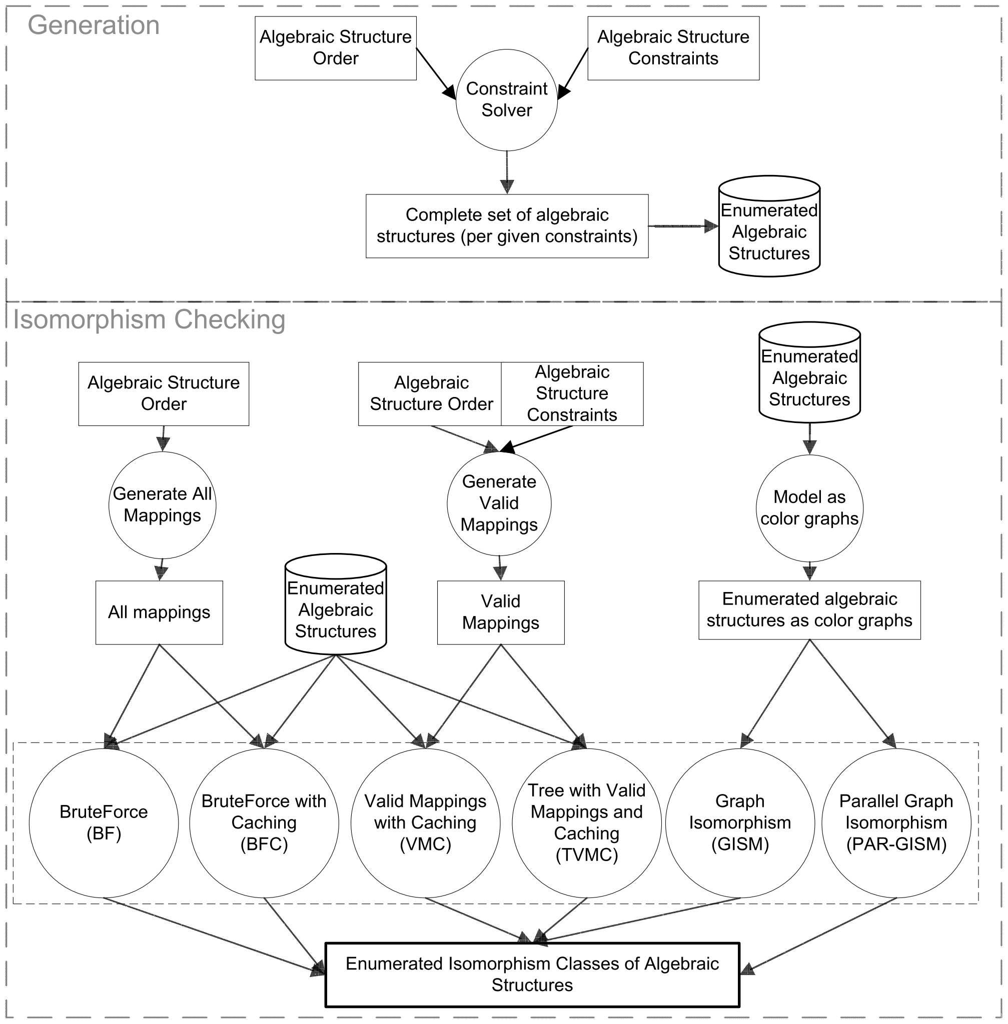

Generation
Algebraic Structure Order
Algebraic Structure Constraints
Constraint Solver
Complete set of algebraic structures (per given constraints)
Enumerated Algebraic Structures
Isomorphism Checking
Algebraic Structure Order
Algebraic Structure Order
Algebraic Structure Constraints
Enumerated Algebraic Structures
Generate All Mappings
Generate Valid Mappings
Model as color graphs
All mappings
Enumerated Algebraic Structures
Valid Mappings
Enumerated algebraic structures as color graphs
BruteForce (BF)
BruteForce with Caching (BFC)
Valid Mappings with Caching (VMC)
Tree with Valid Mappings and Caching (TVMC)
Graph Isomorphism (GISM)
Parallel Graph Isomorphism (PAR-GISM)
Enumerated Isomorphism Classes of Algebraic Structures

FIGURE 3. The process for enumerating isomorphism classes of algebraic structures.

III. PROPOSED METHODOLOGY

The steps for generation of isomorphism classes of algebraic structures of any given order are shown in Figure 3. The input is a set of constraints and the order of the enumerated algebraic structure. The first step in the methodology is to use a *constraint solver* to enumerate complete set of algebraic structures. In the second step, one of the several isomorphism checking approaches can be used to determine isomorphism classes. The input to isomorphism checking approach varies based on its mechanism. The *brute force approach* requires generation of all possible mapping for the given order of algebraic structure. The *tree-based approach* requires as input a set of valid mappings generated based on the constraints and order of algebraic structure. While a *graph isomorphism approach* (such as *nauty*) requires as input the set of algebraic structures modeled as color graph.

The following subsections describe each of these steps in more details.

A. GENERATING ALGEBRAIC STRUCTURES USING CONSTRAINT SOLVER

In this process, we generate all the algebraic structures of any order n with given constraints. We generate the algebraic structures by representing it as a matrix, Mat_n of order n (that is, n rows and n columns) where, each element of the matrix is a domain variable that contains a value in the range $\{0 \ldots n-1\}$. We consider 0 to be the identity element of our algebraic structures. We then apply algebraic structure constraints on Mat_n using a generic constraint solver (Google's *or-tools*), to generate a set $S_n = \{mat_1, mat_2, \ldots mat_p\}$ such that each $mat_i \in S_n$ represents a valid algebraic structure of order n. This set obviously contains many matrices which are isomorphic copies of each other. Table 5 shows the matrix Mat_n (represented as variable x) and the implementation of the algebraic structures constraints of Table 1 using Google's *or-tools*. The interested readers can refer to [33] to get further insights about the functions used in the implementation.

B. IDENTIFYING ISOMORPHISM CLASSES USING BRUTE FORCE APPROACH (BF)

As described in Section II-B, any two algebraic structures are isomorphic to each other if there exists a mapping which can map one structure to another. The *brute force approach* is based on generating all possible permutations of f (called mappings henceforth) and then using them to check for isomorphism over complete set of algebraic structures. If n is the order of algebraic structure, m is the total number of all possible mappings, p is the total number of algebraic structures in the complete set and i is the total number of isomorphism classes within these algebraic structures, then the computational complexity of brute force approach is $O(n^2mpi)$. Unfortunately, m, p and i increase exponentially as a result of increase in the order n of algebraic structure. For example, the number of possible mappings m increase by a factor of $n!$. So, for order 11 algebraic structures, we need to test 11! (about 4 million mappings) to test for isomorphism between a pair of algebraic structures. The number jumps to 13! (about 6 billion) for order 13 algebraic structures. This is clearly not a computationally feasible solution for higher order algebraic structures.

In the next subsections, we present a series of approaches to improve the performance of isomorphism checking. As the brute force approach has a computational complexity of $O(n^2mpi)$, our proposed approaches are mainly intended at reducing the impact of each parameter. In Section III-C, we propose brute force with caching (BFC) approach to reduce the time to identify an isomorphism class by caching the set of useful mappings. In Section III-D, we present valid mappings with caching (VMC) approach which improves BFC approach by reducing the number of possible mappings (m) in addition to using cached lookup of these mappings. In Section III-E we present a tree-based approach (TVMC) that improves VMC by reducing the impact of i by using a tree data structure to keep the set of known isomorphism classes. In Section III-F we present an approach (GISM) to model algebraic structures in color graph format and then use a graph isomorphism checking tool *nauty*. Lastly, in Section III-G we provide details of an approach (PAR-GISM) that uses multiple parallel instances of *nauty* to further enhance the efficiency of identifying isomorphism classes.

C. BRUTE FORCE WITH CACHING APPROACH (BFC)

Certain insights can be utilized to improve the performance of brute force approach. These insights include:

- The set of isomorphism classes is tiny in comparison to the completely enumerated set of algebraic structures. This highlights the presence of a large number of redundant (isomorphic copies) algebraic structures
- A mapping can be utilized to identify several isomorphic copies
- Not every mapping is useful in identifying isomorphism classes
- All mappings have to be checked only if an algebraic structure is not isomorphic to current set of known isomorphism classes

As the number of isomorphism classes is quite small as compared to the total number of algebraic structures, making a right guess on the mapping can increase system performance (as we do not have to continue the search). A previously discovered mapping can be cached to see if it can be reused for identifying another isomorphism class. This idea is implemented in our BFC approach. As the pair of algebraic structures are tested with available mappings, any mapping which successfully identifies an isomorphism class is stored in a cache. In subsequent tests, the cached mappings are checked before the complete set of mappings. A cache hit occurs when a cached mapping is successful in identifying an isomorphism.

For example, in IP loop of order 11 problem, there are 6464 algebraic structures and 3.6 million mappings. The isomorphism checking using BFC approach resulted in a speedup of 2.7 over BF approach (Table 3), with a cache hit ratio of 77% (i.e. 77% of the time the cached mapping was used to identify the isomorphism).

Although BFC approach is useful in enhancing the performance, it is still not computationally feasible to test billions of mappings on thousands of algebraic structures (e.g. for IP loop order 13).

D. VALID MAPPINGS WITH CACHING APPROACH (VMC)

In this section, we describe a technique used to reduce the set of possible mappings. We observed that there are many permutations (mappings) of f which do not satisfy the isomorphic relation $f(m_1[i][j]) = m_2[f(i)][f(j)]$ for all values of $i, j \leq n$ because of constraints shown in Table 1. We consider these mappings as invalid and discard them. We can use constraint solver to find all valid mappings which satisfy isomorphic relationship between two algebraic structures (IP loops in this case). The constraint solver models the system by specifying the relevant constraints from Table 1. After the constraints are embedded in the model, the constraint solver searches the state space to find those permutations that satisfy these constraints. All such permutations are called ‘‘valid mappings’’. If the set S represents all the permutations of f and the set S_v represents all the valid mappings then $S_v \subseteq S$. If $m = (|S|)$ is the total number of all possible mappings, and $v = (|S_v|)$ is the total number of valid mappings then it turns out that $v \ll m$. This reduced set of mappings (i.e. S_v) is then used for identifying isomorphism classes.

Figure 4 shows an example of invalid mapping $f(4\ 5)$. This mapping, if applied to a valid IP loop structure (shown on left) will produce an algebraic structure (shown on right) which does not satisfy the basic symmetry breaking constraint (i.e. $|x - x^{-1}| \leq 1$). For example, in the algebraic structure on the right side, for $x = 3$; $x^{-1} = 5$ and thus $|x - x^{-1}| > 1$.

Detecting isomorphism classes using valid mappings reduces the time complexity from $O(n^2mpi)$ to $O(n^2vpi)$ where $v \ll m$. For example, for IP loop of order 13, the total

$$\left[\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
1 & 2 & 0 & 5 & 7 & 4 & 9 & 3 & 11 & 8 & 14 & 13 & 6 & 10 & 12 \\
2 & 0 & 1 & 7 & 5 & 3 & 12 & 4 & 9 & 6 & 13 & 8 & 14 & 11 & 10 \\
3 & 6 & 8 & 4 & 0 & 10 & 2 & 13 & 1 & 12 & 11 & 14 & 7 & 5 & 9 \\
4 & 8 & 6 & 0 & 3 & 13 & 1 & 12 & 2 & 14 & 5 & 10 & 9 & 7 & 11 \\
5 & 11 & 10 & 2 & 1 & 6 & 0 & 9 & 14 & 4 & 12 & 7 & 13 & 8 & 3 \\
6 & 4 & 3 & 14 & 9 & 0 & 5 & 11 & 13 & 7 & 2 & 1 & 10 & 12 & 8 \\
7 & 10 & 12 & 1 & 2 & 14 & 13 & 8 & 0 & 11 & 6 & 3 & 5 & 9 & 4 \\
8 & 3 & 4 & 11 & 14 & 12 & 10 & 0 & 7 & 13 & 1 & 9 & 2 & 6 & 5 \\
9 & 14 & 13 & 6 & 12 & 1 & 11 & 2 & 5 & 10 & 0 & 4 & 8 & 3 & 7 \\
10 & 5 & 7 & 13 & 11 & 8 & 3 & 14 & 12 & 0 & 9 & 6 & 4 & 2 & 1 \\
11 & 13 & 5 & 10 & 8 & 9 & 14 & 1 & 6 & 3 & 7 & 12 & 0 & 4 & 2 \\
12 & 7 & 14 & 9 & 13 & 2 & 8 & 10 & 4 & 5 & 3 & 0 & 11 & 1 & 6 \\
13 & 9 & 11 & 12 & 10 & 7 & 4 & 6 & 3 & 2 & 8 & 5 & 1 & 14 & 0 \\
14 & 12 & 9 & 8 & 6 & 11 & 7 & 5 & 10 & 1 & 4 & 2 & 3 & 0 & 13
\end{array}\right]
\begin{array}{c} :f\,(4\quad 5) \\ \approx \end{array}
\left[\begin{array}{ccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
1 & 2 & 0 & 4 & 5 & 7 & 9 & 3 & 11 & 8 & 14 & 13 & 6 & 10 & 12 \\
2 & 0 & 1 & 7 & 3 & 4 & 12 & 5 & 9 & 6 & 13 & 8 & 14 & 11 & 10 \\
3 & 6 & 8 & 5 & 10 & 0 & 2 & 13 & 1 & 12 & 11 & 14 & 7 & 4 & 9 \\
4 & 11 & 10 & 2 & 6 & 1 & 0 & 9 & 14 & 5 & 12 & 7 & 13 & 8 & 3 \\
5 & 8 & 6 & 0 & 13 & 3 & 1 & 12 & 2 & 14 & 4 & 10 & 9 & 7 & 11 \\
6 & 5 & 3 & 14 & 0 & 9 & 4 & 11 & 13 & 7 & 2 & 1 & 10 & 12 & 8 \\
7 & 10 & 12 & 1 & 14 & 2 & 13 & 8 & 0 & 11 & 6 & 3 & 4 & 9 & 5 \\
8 & 3 & 5 & 11 & 12 & 14 & 10 & 0 & 7 & 13 & 1 & 9 & 2 & 6 & 4 \\
9 & 14 & 13 & 6 & 1 & 12 & 11 & 2 & 4 & 10 & 0 & 5 & 8 & 3 & 7 \\
10 & 4 & 7 & 13 & 8 & 11 & 3 & 14 & 12 & 0 & 9 & 6 & 5 & 2 & 1 \\
11 & 13 & 4 & 10 & 9 & 8 & 14 & 1 & 6 & 3 & 7 & 12 & 0 & 5 & 2 \\
12 & 7 & 14 & 9 & 2 & 13 & 8 & 10 & 5 & 4 & 3 & 0 & 11 & 1 & 6 \\
13 & 9 & 11 & 12 & 7 & 10 & 5 & 6 & 3 & 2 & 8 & 4 & 1 & 14 & 0 \\
14 & 12 & 9 & 8 & 11 & 6 & 7 & 4 & 10 & 1 & 5 & 2 & 3 & 0 & 13
\end{array}\right]$$

FIGURE 4. Example of an invalid mapping which produces an algebraic structure (on the right) that does not satisfy the basic symmetry breaking constraint ($|x - x^{-1}| \leq 1$).

number of possible mappings (m) is approximately 480 million but there are only 43, 720 valid mappings (i.e. $v = 0.0000913 \times m$). This results in much faster isomorphic detection. Please note that the valid mappings are cached in the same manner as described in the BFC approach in the previous subsection. The valid mappings with caching (VMC) approach thus augments the brute force with caching (BFC) approach.

E. TREE-BASED APPROACH WITH VALID MAPPINGS AND CACHING (TVMC)

In the approaches mentioned in the previous subsections, the identification of a new isomorphism class requires checking an algebraic structure against all the previously discovered isomorphism classes using the available set of mappings. As a result, the computational complexity increases with the increased number of discovered isomorphism classes. A careful analysis reveals that many of these isomorphism classes share similar structure (elements) and many redundant computations could be avoided with a suitable re-organization of isomorphism classes elements. We propose to represent the isomorphism classes using a tree-based structure to reduce the computational complexity. Figure 5 shows an example on how seven isomorphism classes (labelled (a) to (g)) are represented within our proposed tree structure. The tree structure is built such that each branch of the tree represents one isomorphism class, and each level in the tree represents a unique position of the matrix element. Every newly discovered isomorphism class is added to the existing tree as shown in Algorithm. 1. As long as two isomorphism classes have the same element values, they are represented by a single branch in the tree. If element values differ at any depth in a branch, a new offshoot is created to represent all the subsequent values.

The tree representation of isomorphism classes allows the identification of isomorphism class of an algebraic structure with a single traversal of the tree. The traversal begins from the root and continues to traverse all nodes at each level (left to right). At every level in the tree, if the algebraic structure does not map to any node then the complete subtree of the node is discarded from the search space. For example in Figure 5, as the algebraic structure does not map to the first node at level m_{01}, the isomorphism classes (a) and (b) are discarded from search space. Also, if the algebraic structure maps to one of the nodes in tree then rest of the siblings at current level and their corresponding sub-trees are discarded from search space. For example in Figure 5, as the algebraic structure maps to the second node at level m_{01}, the isomorphism classes (d), (e), (f) and (g) are discarded from search space. If it does not match any node in the tree at current level then the whole search space is discarded and the search restarts with a new valid mapping. The isomorphism checking algorithm using tree-based structure is shown in Algorithm. 3. The tree representation of the isomorphism classes reduces the computational complexity of checking for isomorphism from $O(n^2 vpi)$ to $O(n^3 vp)$, thus reducing the time complexity exponentially (as $n \ll i$ for higher order algebraic structures). This is possible because in TVMC approach an algebraic structure is checked against all isomorphism classes by traversing the tree structure only once. Thus, in the worst case, n elements are checked at each level which makes the time complexity of checking a single algebraic structure with a single mapping to $O(n^3)$. Whereas, in VMC approach a single algebraic structure is checked for a single mapping against all i isomorphism classes making it $O(n^2 i)$.

F. GRAPH ISOMORPHISM APPROACH (GISM)

Graph isomorphism is a well established research area with many state of the art graph isomorphism checking tools (e.g. *nauty*). These approaches use graph canonization to efficiently identify isomorphism classes. However, the use of these approaches requires that algebraic structures be represented in an equivalent graph format. In order to represent an algebraic structure of order n as a graph,

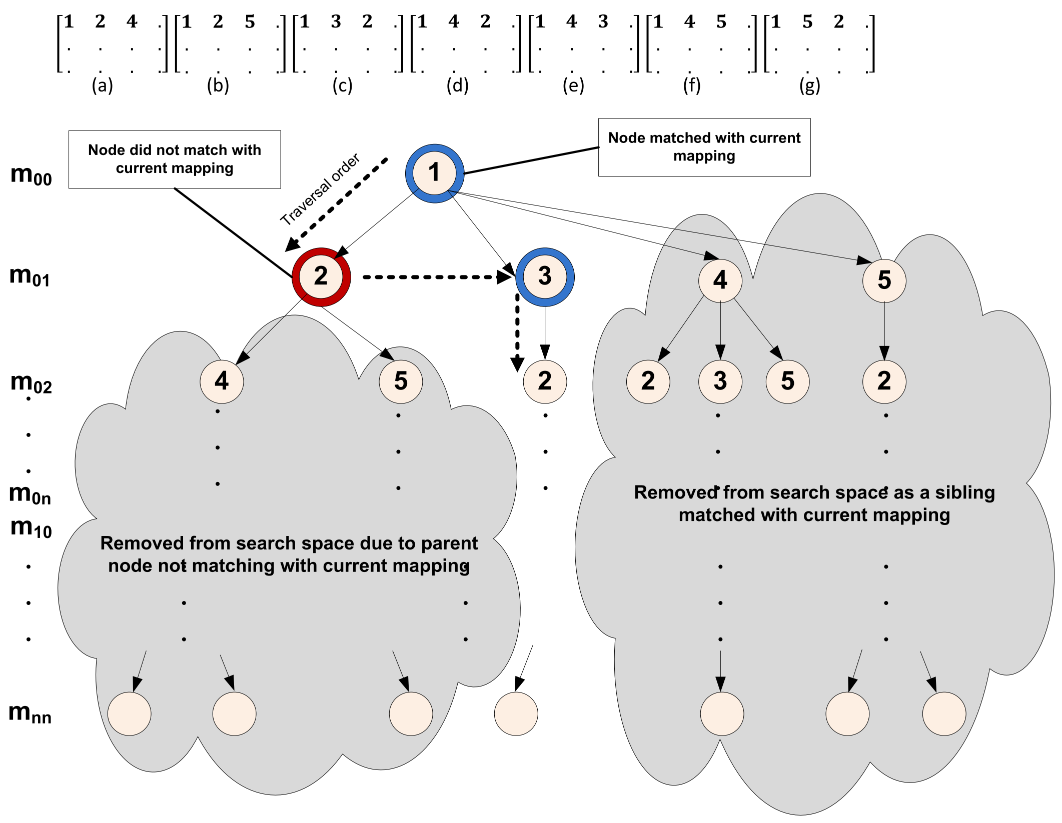

(a)
(b)
(c)
(d)
(e)
(f)
(g)
Node did not match with current mapping
Node matched with current mapping
Traversal order
m00
m01
m02
m0n
m10
mnn
Removed from search space due to parent node not matching with current mapping
Removed from search space as a sibling matched with current mapping

FIGURE 5. Tree traversal for identifying the isomorphism class. The tree structure helps in eliminating large part of the search space to efficiently identify the isomorphism class.

we model it as a color graph [10]. The rows $(r_0, r_1, \ldots, r_n)$ are represented by n same color vertices. Similarly, columns $(c_0, c_1, \ldots, c_n)$ and symbols $(s_0, s_1, \ldots, s_n)$ are represented by their corresponding same color vertices. The matrix positions $(a_{00}, a_{01}, \ldots, a_{ij})$ are represented by n^2 same color vertices. The graph has a total of $n^2 + 3n$ vertices with four different kinds of color vertices.

Figure 6 shows how an order 3 algebraic structure is modeled as a graph. Each vertex color is represented by a unique shape. There is an edge between every matrix position vertex (a_{ij}) and its corresponding row r_i, column c_j and its value $s_{matrix[i][j]}$. Please note that all the edges are the same (i.e. there is no edge coloring). The edges are shown with different styles for clarity in the diagram to differentiate between overlapping edges. Additionally, edges have been added between every triplet of r_i, c_i, s_i to disallow any paratopism. That is, although the graph representation would allows swapping of columns with each other, rows with each other and symbols with each other, it will not allow swapping of a row with column or row with a symbol etc. This would prevent formation of conjugate algebraic structures. Please note that edges between every triplet of r_i, c_i, s_i are not shown as connected in the diagram for clarity purpose.

The algebraic structures generated using constraint solver are modeled into their corresponding graph representation and stored in a file (g6 or s6 format). These graphs can be used

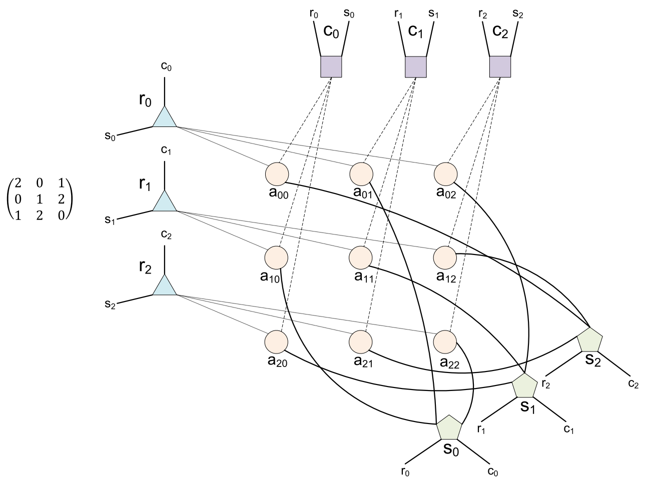


FIGURE 6. Modeling algebraic structure as a graph representation.

by *nauty* to identify isomorphism between these graphs by generating their canonical labels. Figure 7 shows an example of how two algebraic structures which are represented by our graph representation are identified as isomorphic due to bijective mapping shown in the figure. Note that, we have rearranged the graph vertices to clearly visualize that the given bijective mapping indeed produces an identical isomorphic

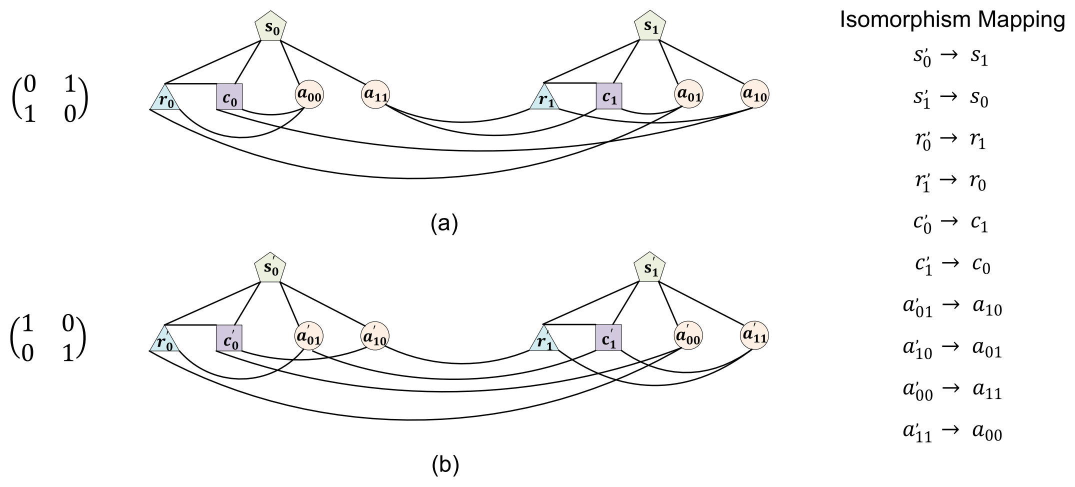


FIGURE 7. Ismorphism checking between two isomorphic algebraic structures using *nauty*.

TABLE 2. Time taken to generate complete set of algebraic structures, all mappings, valid mappings and graphs.

Problem	**Time to Generate (in seconds)**			
	All Solutions	**All Mappings (S)**	**Valid Mappings (S_v)**	**Graphs**
IP loop of order 7	0.029	0.023	0.016	0.011
IP loop of order 9	0.047	0.062	0.059	0.072
IP loop of order 11	0.95	1.692	1.094	1.714
IP Loop of exponent 3 (order 13)	2.49	206.835	8.883	4.907
Loop of order 7	120.6	0.016	0.064	481.804
IP loop of order 13	732.2	206.835	9.852	1583.069

graph. We have used the small order 2 algebraic structure in this figure for clarity purpose.

G. PARALLEL GRAPH ISOMORPHISM APPROACH (PAR-GISM)

The graph isomorphism based approach provides an efficient way to perform isomorphism checking of higher order algebraic structures. However, it does not make use of multiple cores available on most of the system. In order to take advantage of the multiple cores, we propose a data parallelism approach called PAR-GISM. In this approach, the set of 4-color graphs is equally divided between multiple parallel instances of *nauty*. Each instance determines the respective isomorphism classes within its assigned set of color graphs. The results are then merged to form a single file which can easily be analyzed by a single instance *nauty* to determine the final set of isomorphism classes.

This resulted in a highly scalable and efficient mechanism for enumeration of higher order algebraic structures using multi-core systems. For example, we observed a speedup of 13 by using this approach on a 36-core machines with 64 threads for enumerating IP loops of order 13. More importantly, this helped us to solve the previously computationally unsolved problem of determining isomorphism classes of exponent 3 IP loops of order 15 [22].

IV. RESULTS

A series of experiments were conducted to evaluate and compare effectiveness of the proposed approaches. The initial set of experiments were conducted on a regular desktop system (dual core Intel i7 processor @ 2.8 GHz and 8 GB of RAM) with Windows 10 OS. In Section IV-A, we present the computational requirements for completing the pre-processing steps before conducting isomorphism checking. Section IV-B presents a comparative analysis of BF, BFC, VMC, TVMC and GISM isomorphism checking approaches. In Section IV-C, we present the results of a more rigorous set of experiments conducted on several hardware platforms to evaluate and compare the performance of GISM and its parallel implementation (PAR-GISM).

A. PRE-PROCESSING STEPS

As shown in Figure 3, certain pre-processing steps (i.e. to generate the complete set of algebraic structures, all mappings (S), valid mappings (S_v) and to convert algebraic structures into graphs) are required before conducting isomorphism checking using our proposed approaches. Table 2 shows the computational time required to complete these pre-processing steps.

The generation of complete set of algebraic structures is required for every approach. Other pre-processing steps

TABLE 3. Isomorphism checking time and speedup (in brackets) of proposed approaches for enumerating algebraic structures.

Algebraic Structure	Total Solutions	Isomorphism Classes	All Mappings $\lvert S \rvert$	Valid Mappings $\lvert S_v \rvert$	Isomorphism Checking Time in seconds (speedup)				
					BF	BFC	VMC	TVMC	GISM
IP Loop Order 7	4	2	720	48	0.086	0.078 (1.1)	0.03 (2.6)	0.033 (0.91)	0.062 (0.53)
IP Loop Order 9	64	7	40320	366	0.916	0.72 (1.3)	0.047 (15.3)	0.062 (0.76)	0.073 (0.85)
IP Loop Order 11	6464	49	$\approx 3.63 \times 10^6$	3654	3986	1459 (2.7)	5.5 (265.3)	2.3 (2.4)	10.6 (0.22)
IP Loop of exponent 3 (Order 13)	22000	64	$\approx 4.79 \times 10^8$	34804	NF	864020	324 (2666.7)	41 (7.9)	181 (0.23)
Loop Order 7	16942080	23746	720	720	NF	NF	2053624	817 (2513)	2458 (0.33)
IP loop Order 13	7853368	10342	$\approx 4.79 \times 10^8$	43720	NF	NF	NF	21952	32090 (0.68)

are based on particular isomorphism checking approach. For higher order algebraic structures, the pre-processing time is quite small as compared to isomorphism checking time (shown in Table 3). The computational time required to complete pre-processing steps can be further optimized by using a parallelized approach but we did not focus on this aspect of the problem in this paper.

We have excluded the pre-processing time from our comparative analysis of the isomorphism checking approaches in the subsequent sections. The execution times in the subsequent sections refers to the time required for isomorphism checking only.

B. COMPARATIVE ANALYSIS OF PROPOSED APPROACHES

Table 3 shows the isomorphism checking time (in seconds) and speedup gained by the proposed approaches for an increasing order of algebraic structures. The speedup is computed relative to the previous approach (i.e. speedup of BFC is relative to BF approach while speedup of VMC is relative to BFC etc.). It can be seen that with the increase in order of algebraic structures, the number of algebraic structures, isomorphism classes and the mappings required to check for isomorphism increase exponentially. The BF approach works quite well for enumerating lower order IP loops of order 7 and 9, but the enumeration time increases by a factor of 4000 for enumerating IP loops of order 11 due to 100 fold increase in the number of solutions and about 100 fold increase in the number of mappings. Note that the enumeration time likely to exceed a week is shown as not feasible (NF).

Figure 8 provides a comparison of isomorphism checking times (Y-axis drawn on $\log_{10}$ scale) with the increased computational complexity of algebraic structures. BFC performs slightly better than BF approach due to increased cache hit ratio. The performance gain becomes larger with higher order algebraic structures (IP loop of order 11). Figure 9 shows the number of cache hits and misses encountered during the enumeration of isomorphism classes for IP loop of order 11 and exponent 3 IP loop of order 13 respectively. It can be seen that the cache hits dominate cache misses by a factor of 3 and thus reduce the time to identify an isomorphism class using cached mappings. As shown in Table 3, these cache hits result in a speedup for enumerating isomorphism classes for IP loop of order 11 by almost a factor of 3.

The VMC approach provides considerably larger speedup over BFC due to drastically reduced number of valid mappings. The speedup increases with the increased order of algebraic structures, as the difference between number of valid mappings ($|S_v|$) and all possible mappings ($|S|$) grows exponentially wider. For example, the time taken to enumerate exponent 3 IP loops of order 13 is drastically reduced by a factor of 2666 with VMC approach as the number of mappings get drastically reduced by a factor of about 14000. The VMC approach performs better than all other approaches for lower order algebraic structures (IP loop of order 7 and 9). However, the performance of VMC degrades drastically for higher order algebraic structures (IP loop of order 11 and beyond) due to increased number of isomorphism classes and valid mappings.

TVMC approach performs better than VMC for higher order algebraic structures due to efficient use of tree structure to reduce the computations in the presence of a large number of isomorphism classes. It can be seen that the speedup of TVMC against VMC increases with the increase in number of isomorphism classes. For example, there is

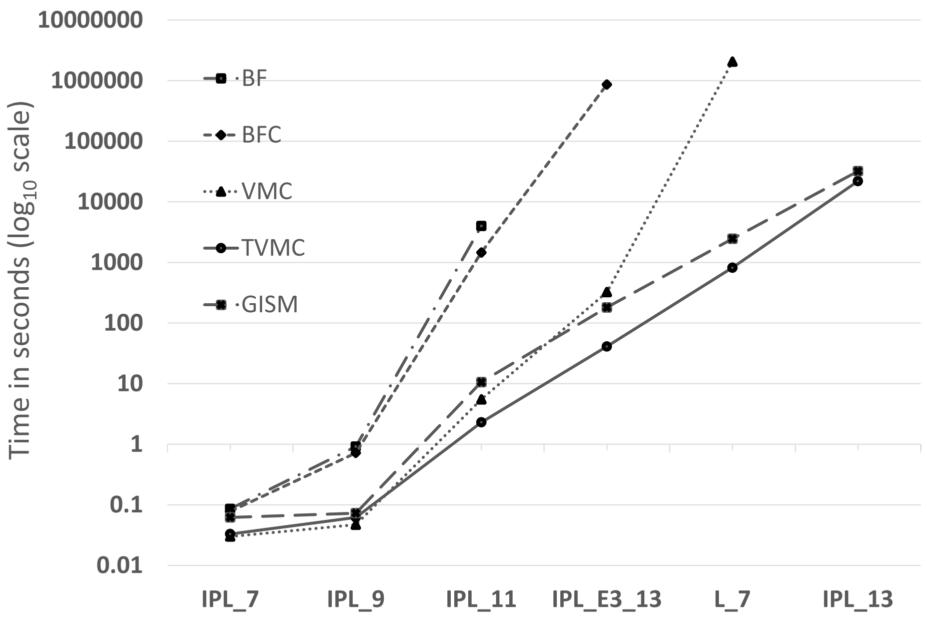
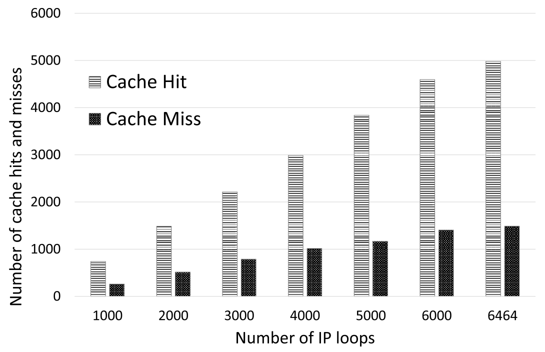
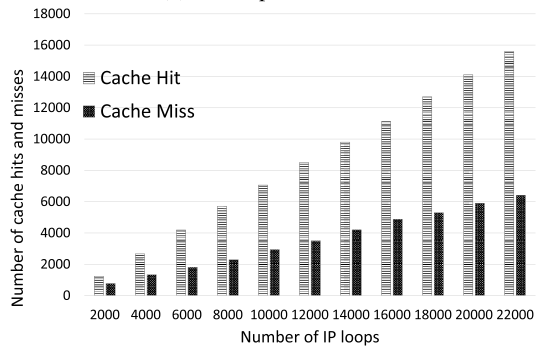


FIGURE 8. A comparison of execution times with BF, BFC, VMC, TVMC and GISM approaches for different problem sizes.



(a) IP Loop of order 11



(b) Exponent 3 IP Loop of order 13

FIGURE 9. Number of cache hits and misses of mappings during enumeration of isomorphism classes of various algebraic structures.

a speedup of 2.4 for IP Loop of order 11 which contains 49 isomorphism classes. However, for loop of order 7, the number of isomorphism classes is comparatively larger

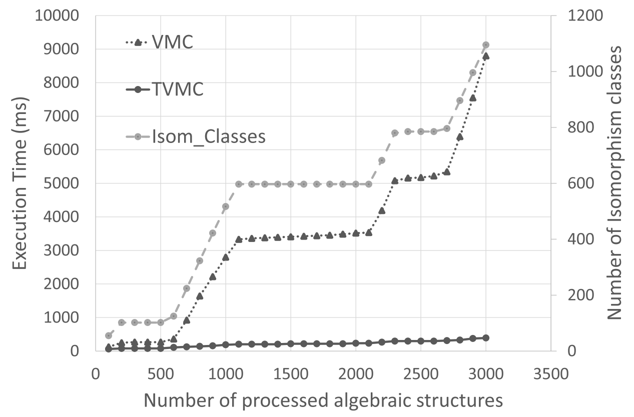


FIGURE 10. The performance improvement of TVMC over VMC approach for Loop 7 as a function of increased number of isomorphism classes. For brevity purpose, the plot only shows data for first 3000 Loop 7 algebraic structures. The increase in execution time is minimal in TVMC as compared to VMC approach because of proposed tree structure.

(23746 isomorphism classes). This results in a large speedup gain (by a factor of 2513) for TVMC over VMC approach. As expected, TVMC approach does not perform better than VMC approach for lower order algebraic structures (when $i \leq n$). This can partially also be attributed to the overhead of creating and managing tree data structure for a small number of isomorphism classes. We also evaluated the performance impact of increasing number of isomorphism classes on VMC and TVMC approaches. Figure 10 compares the execution time performance for VMC and TVMC approaches during the enumeration of initial 3000 algebraic structures of loop of order 7. The x-axis shows the number of algebraic structures processed in an increment of 100. The left vertical axis shows the execution time (in ms) for VMC and TVMC approaches, while the right vertical axis shows the number of isomorphism classes identified during enumeration. It can be seen that as the number of discovered isomorphism classes increase, the execution time of VMC increases almost linearly with it. The execution time of TVMC, as expected, is largely unaffected by the increase in number of discovered isomorphism classes.

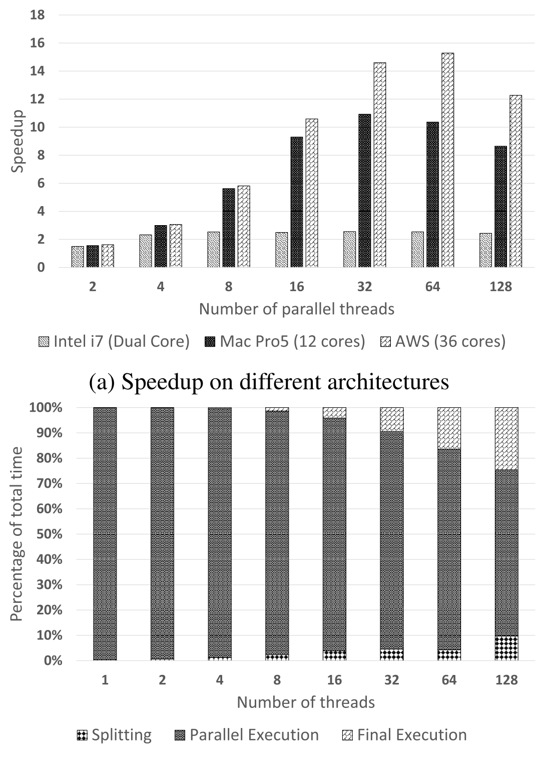
Although GISM approach does not perform better than TVMC for the algebraic structure problems analyzed in this comparative study. But it can be clearly seen that the performance of GISM approach closes in to TVMC with the increase in the order of algebraic structures and it is expected to outperform TVMC approach for higher order algebraic structures (e.g. for IP loops beyond order 13). Another nice aspect of GISM approach is that it consumes relatively small amount of memory in comparison to TVMC approach. For example, for enumerating IP loops of order 13, GISM approach used on average about 4.6MB of memory as compared to the 305MB used by the TVMC. This makes TVMC a less scalable choice for parallel implementation (as 64 instances would require around 20GB of RAM). These two insights (i.e. better performance expectation in enumerating higher order algebraic structures and relatively small memory consumption) led us to consider using GISM approach for attempting to enumerate even higher order algebraic structures with a parallel implementation (as described previously in Section III-G). In the next section, we describe the results obtained using parallel implementation of GISM (PAR-GISM) and its comparison with single instance of GISM.

C. AN EVALUATION OF GISM AND PAR-GISM APPROACHES FOR ENUMERATING HIGHER ORDER ALGEBRAIC STRUCTURES

The experiments described in the previous section showed that GISM is expected to perform better than other approaches for enumeration of higher order algebraic structures and it has relatively small memory consumption. We conducted further experiments to determine the potential of speedup by using parallel instances of GISM (PAR-GISM). We also desired to use the PAR-GISM approach to solve a previously unknown and computationally challenging problem of enumerating exponent 3 IP loops of order 15. These experiments were conducted on the following machine architectures:

- Intel i7 (dual core with 4 logical processors @ 2.8 GHz)
- Mac Pro 5,1 (12 cores Intel Xeon @ 2.4 GHz))
- Amazon Web Services (AWS) EC2 c4.8xlarge instance (36 cores @ 3.5 GHz)

In order to gauge the potential of speedup by using the parallel GISM approach (PAR-GISM), we first evaluated the performance improvements in enumerating IP loops of order 13 on all three hardware architectures with varying number of threads.



(b) Breakdown of total execution time on Mac Pro

FIGURE 11. **Performance evaluation of PAR-GISM on three different hardware architectures with an increasing number of threads for enumerating IP loops of order 13.**

Figure 11(a) shows the speedup gained on these machines with an increasing number of parallel executing threads. As expected, the speedup increases with increased number of threads on multi core systems. The speedup is initially linear but slowly degrades as it reaches an optimal point. The optimal point depends upon the number of available cores. For example, the optimal point for AWS 36-core machine is reached with 64 threads while for 12-core Mac Pro 5 it is reached with 32 threads. The reason for continuous loss in speedup is the increased amount of time spent in the final execution (with merged file) and in chunking (splitting the graphs). The final execution time increases since size of the merged file (containing output graphs from each thread) increases with increased number of threads. This, in turn, makes it more time consuming for final instance of *nauty* to determine the isomorphism classes. Figure 11(b) shows the time break-down for chunking (splitting), parallel and final execution time as the percentage of total time on Mac Pro 5. It can be seen that as the number of threads are increased the time to split and the time of final execution starts to increase, thus reducing the benefit gained from parallel threads.

Encouraged with these results, we attempted to solve the computationally challenging task of determining previously

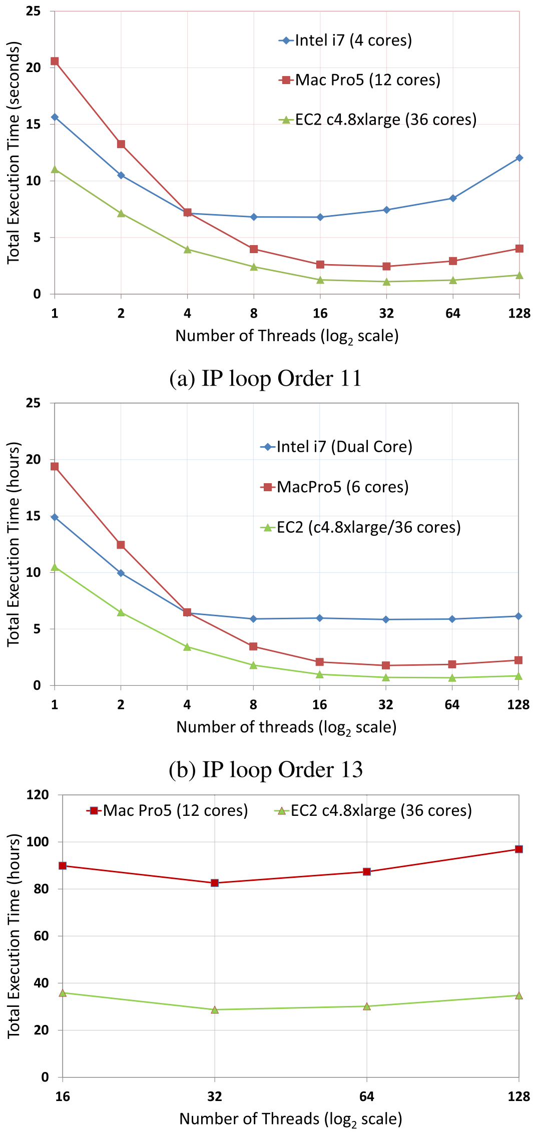
TABLE 4. Performance gain by using parallel *nauty* instances for enumeration of IP loops on AWS 36-core machine.

Problem	Total No. of Algebraic Structures $\lvert S_n \rvert$	No. of Isomorphism Classes	Time with single *nauty* instance (T_o)	Time with parallel instances of *nauty* (T_p)	Speedup (T_o/T_p)
IP loop of order 11	6464	49	10 seconds	1.1 second (32 threads)	9
IP loop of order 13	7853368	10342	9 hours	41 minutes (64 threads)	13
Exponent 3 IP loop of order 15	71149968	27698	NF	28.7 hours (32 threads)	NA

unknown set of isomorphism classes for exponent 3 IP loops of order 15. We were able to successfully determine the isomorphism classes on AWS 36-core machine in approx. 29 hours with 32 threads [22]. Table 4 shows the computational problems and the corresponding maximum speedup obtained by using PAR-GISM approach on AWS 36-core machine. Note that the enumeration time with GISM exceeding one week is shown as not feasible (NF).

We conducted a series of experiments to evaluate the performance gain on all three architectures, with problems of varying complexity and with increased number of threads. Figure 12a shows the results for IP loop of order 11. The horizontal axis showing the number of threads is drawn on $\log_2$ scale. It can be seen that parallelization provides almost linear improvement in execution time on all architectures which subsequently turns into sub-linear improvement and then starts to deteriorate. The optimal point depends on the number of available cores. Intel i7 machine initially performs better than Mac Pro 5 since its core is operating at higher speed. However, after 4 threads the performance of Mac Pro 5 gets better because of available cores. AWS 36-core machine outperforms due to large number of cores and higher operating speed of each core. A similar trend is visible for IP loop of order 13 in Figure 12b. The execution time of this problem was reduced from about 10 hours (with single thread) to less than one hour (with 64 threads) on AWS 36-core machine. Figure 12c shows the execution time for enumerating exponent 3 IP loop of order 15. Due to the large amount of time (in weeks) needed to solve this problem on a single *nauty* instance, we chose to perform experiment with 16 threads and higher. The best case execution time for this problem was 29 hours with 32 threads on AWS 36-core machine.

Figure 13a shows the box plots of parallel execution time variation on 12-core Mac Pro 5 for enumerating IP loop of order 13. It can be seen that there is huge difference in parallel execution times with fewer threads. As the number of threads are increased the variations become smaller but even in the best case (with 64 threads) there is substantial (by a factor of 3) difference in parallel execution times. This implies that a suitable load balancing scheme can be considered to improve the overall total execution time. Figure 13b shows the parallel execution time variations for enumerating exponent 3 IP loops of order 15. In this case, the difference in parallel



(a) IP loop Order 11

(b) IP loop Order 13

(c) Exponent 3 IP loops of Order 15

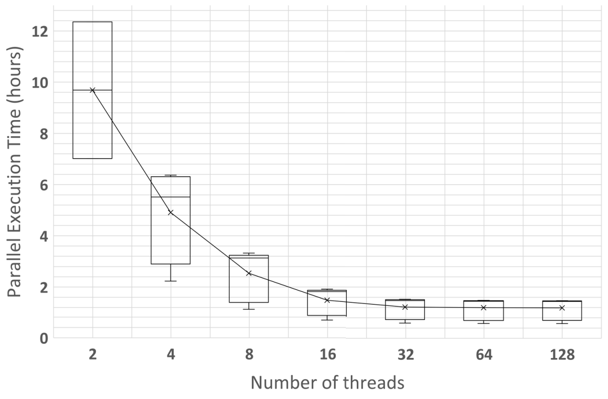
FIGURE 12. Total execution time for enumeration of different algebraic structures with increased number of parallel threads on three different machine architectures.

execution times is marginal compared to the average parallel execution time and thus indicating lesser benefit for any load balancing scheme.

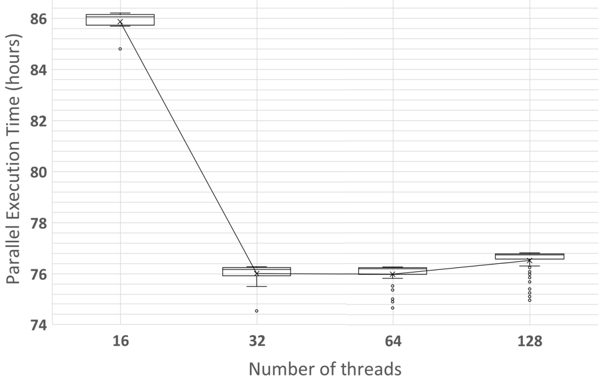
TABLE 5. **Implementation of algebraic structures constraints in Google *or-tools*.**

```
import com.google.ortools.constraintsolver.Solver;
import com.google.ortools.constraintsolver.IntVar;
// Create matrix where each element is a domain variable with range from 0 to n-1
IntVar[][] x = new IntVar[n][n];
solver = new Solver("My Constraint Solver");
for i:=0 to n-1 {
  for j:=0 to n-1 {
    x[i][j] := solver.makeIntVar(0, n-1, "x[" + i + "," + j + "]");
}}
//Implementation of Latin square constraint:
// Create a consraint for each row and column to contain unique values
for i:=0 to n-1 {
  IntVar[] row := new IntVar[n];
  for j:= 0 to n-1 {
    row[j] := x[i][j];
  }
  solver.addConstraint(solver.makeAllDifferent(row));
}
for j:= 0 to n-1 {
  IntVar[] col := new IntVar[n];
  for i:= 0 to n-1 {
    col[i] := x[i][j];
  }
  solver.addConstraint(solver.makeAllDifferent(col));
}
// Implementation of loop constraint:
// First row and first column have fixed values (equal to row or column index)
for i:=0 to n-1 {
  x[0][i] := i;  x[i][0] := i;
}
// Implementation of IP loop constraint:
IntVar b,c, d;
for i := 0 to n-1 {
  for k := 0 to n-1 {
    for q:= 0 to n-1 {
      for m:= 0 to n-1 {
        b := solver.makeIsEqualCstVar(x[i][k],0); // if part
        c := solver.makeIsEqualCstVar(x[i][q], m); // then part
        d := solver.makeIsEqualCstVar(x[k][m], q); // then part
        solver.addConstraint(solver.makeLessOrEqual(b, solver.makeIsLessOrEqualVar(c, d)));

        b := solver.makeIsEqualCstVar(x[i][k],0); // if part
        c := solver.makeIsEqualCstVar(x[q][i], m); // then part
        d := solver.makeIsEqualCstVar(x[m][k], q); // then part
        solver.addConstraint(solver.makeLessOrEqual(b, solver.makeIsLessOrEqualVar(c, d)));
}}}}
// Implementation of basic symmetry breaking in IP loop constraint:
IntVar d := solver.makeIntVar(2, 2, "d");
IntVar b, c;
for i := 1 to n-1 {
  for j := 1 to n-1 {
    b := solver.makeIsEqualCstVar(x[i][j],0); // if part
    c := solver.makeIsGreaterCstVar(d, Math.abs(i - j)); // then part
    solver.addConstraint(solver.makeLessOrEqual(b,c));
}}
// Implementation of Exponent 3 constraint:
IntVar b, c;
for i := 0 to n-1 {
  for m := 0 to n-1 {
    b := solver.makeIsEqualCstVar(x[i][i], m); // if part
    c := solver.makeIsEqualCstVar(x[m][i], 0); // then part
    solver.addConstraint(solver.makeLessOrEqual(b,c));
}}
```



(a) Parallel execution time variation in threads for IP loop of order 13



(b) Parallel execution time variation in threads for exponent 3 IP loop of order 15

FIGURE 13. Parallel execution time variations on 12-core Mac Pro 5.

It can also be observed in Figure 13a that the parallel execution time drops considerably even with larger number of threads than the available cores (16-32 threads on 12-core system). This indicates that a single instance thread is spending considerable time in I/O (reading and sorting graphs from disk) providing ample opportunities to other threads to simultaneously continue their execution. This behavior continues, to a smaller extent, up to 128 threads. A similar trend is visible in Figure 13b for exponent 3 IP loop of order 15 with considerable improvement in parallel execution time from 16 to 32 threads. The parallel execution time, however, starts to increase with 128 threads indicating higher competition among threads for available cores. There are also more outliers in this case, indicating that some threads finish earlier while others have to wait for core availability and end up finishing later.

In general, we have observed that PAR-GISM approach can be scaled higher to solve other computationally challenging problems. The speedup factor is more profound when compared across different machine architectures. For example, it took about 19 hours to solve IP loop of order 13 problem on Mac Pro 5 with a single *nauty* instance, while the same problem was solved on 36-core machine with 32 threads in 0.7 hours (speedup of 27).

Algorithm 1 Add a Matrix Representing an Isomorphism Class to the Tree Structure

```
function AddMatrixToTree (m : matrix to be added);
Input  : Matrix m to be added into the tree t
Output: Tree t
Node node = null;
if root = null then
    root = AddSubTree(0, 0, m);
else
    /* root already exists; skip first
       element as it is always 0        */
    row = 0, col = 1, t = root;
    /* identify a child node that has
       same value as the element at
       current m[row,col]               */
    while t ≠ null do
        found = false;
        list = t.GetAllChildren();
        for each child in the list do
            if child.value == m[row][col] then
                found = true;
                break;
            end
        end
        if found then
            /* found a child having same
               value at m[row][col]     */
            if child is leaf then
                return;
            end
            /* traverse next level of the
               tree                     */
            t = child, col++;
            if col == n then
                col = 0, row++;
            end
        else
            /* no child found, we have to
               add matrix m to the tree
               at this level            */
            break;
        end
    end
    /* add remaining columns in the
       current row to tree              */
    parent_node = child.Parent();
    node = AddSubTree(n-1, col, m);
    parent_node.Add(node);
    /* add all columns of each
       remaining row to tree            */
    parent_node = node;
    node = AddSubTree(row + 1, 0, m);
    parent_node.Add(node);
end
```

Algorithm 2 Add Subtree in *m* From Position (Row, Col)

```
function AddSubTree (Add subtree in $m$ from position (row, col));
Input  : position:(row, col), matrix $m$
Output: Node
Node node = null;
for $i = row$ to $n - 1$ do
    for $j = col$ to $n - 1$ do
        if $node == null$ then
            /* first node in the tree */
            node = new Node(m[i][j]);
        else
            /* get last child */
            leaf = node.GetFirstLeaf();
            leaf.add(new Node(m[i][j]));
        end
    end
end
return node;
```

Algorithm 3 Find Isomorphism With Given Mappings

```
function FindIsomorphism ($m$ : matrix to be checked for isomorphism);
Input  : Matrix m to be checked for isomorphism
         against mapping f
Output: true or false
Node t = root;
for $i = 0$ to $n - 1$ do
    for $j = 0$ to $n - 1$ do
        list = t.GetAllChildren();
        rhs = m[f(i)][f(j)];
        for each child in the list do
            lhs = child.value;
            if $lhs == rhs$ then
                matched = true;
                /* no need to check other
                   children */
                break;
            end
        end
        if not matched then
            /* must match at least one
               child at each level */
            return false;
        else
            /* the child that matched */
            t = child;
        end
    end
end
```

V. CONCLUSION

Algebraic structures are widely used in several scientific disciplines. Enumerating higher order algebraic structures and identifying their isomorphism classes is a computationally challenging task. This paper presents a methodology for efficiently enumerating isomorphism classes of algebraic structures. A novel algorithm (TVMC) is proposed that utilizes valid mappings, caching and a tree-based data structure to efficiently enumerate the isomorphism classes for given algebraic structures. The paper also presents the mechanism to model algebraic structures as color graphs and then use a state of the art graph isomorphism algorithm to identify the isomorphism classes. A comparative analysis is then performed to evaluate the performance of the isomorphism checking approaches. The enumeration of IP loops of order 13 which took 9 hours with state of the art graph isomorphism approach (*nauty*) on a regular desktop system was enumerated by our proposed algorithm (TVMC) in about six hours. A similar performance improvement is observed in several other enumeration problems. The paper also presents a parallel implementation to further enhance the isomorphism checking approach on multi-core systems. The proposed parallel implementation was able to enumerate IP loops of order 13 within 41 minutes on a 36-core machine. The parallel implementation was then used to enumerate the previously unsolved problem of determining isomorphism classes of exponent 3 IP loops of order 15.

This work can be extended in multiple directions. The parallel approach can be utilized to enumerate other currently unsolved algebraic structures like IP loops of higher order (order 15 and beyond), symmetric IP loops, C-loops and flexible loops. The proposed TVMC algorithm can be further optimized to make better memory usage and to develop a parallel implementation of TVMC with a shared tree structure. The parallel implementations of TVMC or GISM based approaches can be further enhanced with a distributed mechanism to dynamically allocate tasks among nodes based on their capabilities and current workload. The storage footprint for enumerating higher order algebraic structures in the distributed systems can be reduced by performing algebraic structure generation and isomorphism checking simultaneously on different nodes with proper communication and coordination between these nodes. Another interesting direction is to determine the upper bound of the proposed TVMC algorithm. It is quite challenging to determine the upper bound in the presence of multiple factors including number of enumerated algebraic structures (p), number of valid mappings (v), order of algebraic structure (n), the complex relationship between these algebraic structures and the machine architecture.

ACKNOWLEDGMENT

The views and conclusions herein are those of the author and do not represent the official policies of the university. The author would like to acknowledge Dr. A. Ali of Quaid-e-Azam University for introducing the problem domain of algebraic structures. The author would also like to acknowledge constant support and feedback throughout the project from Dr. S. Muhammad and Dr. N. Mohammad at PMU. Finally, the author would like to acknowledge Prof. B. D. McKay of Australian National University (ANU) for his insightful feedback on properly modelling algebraic structures using *nauty*.

REFERENCES

[1] A. D. Keedwell and J. Dénes, *Latin Squares and Their Applications*. Amsterdam, The Netherlands: Elsevier, 2015.

[2] M. Battey and A. Parakh, ''An efficient quasigroup block cipher,'' *Wireless Pers. Commun.*, vol. 73, no. 1, pp. 63–76, Dec. 2012.

[3] A. Krapez, ''An application of quasigroups in cryptology,'' *Math. Maced*, vol. 8, pp. 47–52, Aug. 2010.

[4] B. D. McKay, A. Meynert, and W. Myrvold, ''Small latin squares, quasigroups, and loops,'' *J. Combinat. Des.*, vol. 15, no. 2, pp. 98–119, 2007.

[5] I. P. Gent and B. Smith, ''Symmetry breaking during search in constraint programming,'' in *Proc. ECAI*, 1999, pp. 599–603.

[6] I. P. Gent, W. Harvey, and T. Kelsey, ''Groups and constraints: Symmetry breaking during search,'' in *Principles and Practice of Constraint Programming*. Berlin, Germany: Springer, 2002.

[7] A. Ali and J. Slaney, ''Counting loops with the inverse property,'' *Quasigroups Rel. Syst.*, vol. 16, no. 1, p. 13, 2008.

[8] K. Hermiston, ''The largest critical sets of latin squares,'' in *Proc. 53rd Annu. Conf. Inf. Sci. Syst. (CISS)*, Mar. 2019, p. 1–6.

[9] B. D. McKay and A. Piperno, ''Practical graph isomorphism, II,'' *J. Symbolic Comput.*, vol. 60, pp. 94–112, Jan. 2014.

[10] B. D. McKay and A. Piperno. *Nauty and Traces User's Guide (Version 2.5)*. Accessed: Mar. 1, 2020. [Online]. Available: http://users.cecs.anu.edu.au/~bdm/nauty/

[11] S. E. Bammel and J. Rothstein, ''The number of 9 × 9 latin squares,'' *Discrete Math.*, vol. 11, no. 1, pp. 83–95, 1975.

[12] J. Slaney and A. Ali, ''Generating loops with the inverse property,'' in *Proc. ESARM*, G. Sutcliffe, S. Colton, and S. Schulz, Eds., 2008, pp. 55–66.

[13] J. W. Brown, ''Enumeration of Latin squares with application to order 8,'' *J. Combinat. Theory*, vol. 5, no. 2, pp. 177–184, Sep. 1972.

[14] A. Cayley, ''On Latin squares,'' *Oxford Camb. Dublin Messenger Math.*, vol. 19, no. 1, pp. 85–239, 1890.

[15] L. Euler, ''Recherches sur une nouvelle espéce de quarrés magiques,'' *Verhandelingen/Uitgegeven Door Het Zeeuwsch Genootschap der Wetenschappen te Vlissingen*, vol. 9, pp. 85–239, 1782.

[16] R. A. Fisher and F. Yates, ''The 6×6 Latin squares,'' in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 30. Cambridge, U.K.: Cambridge Univ. Press, 1934, pp. 492–507.

[17] S. M. Jacob, ''The enumeration of the latin rectangle of depth three by means of a formula of reduction, with other theorems relating to non-clashing substitutions and Latin squares,'' *Proc. London Math. Soc.*, vol. 31, no. 1, pp. 329–354, 1930.

[18] P. A. MacMahon, *Combinatory Analysis*, vol. 1. Cambridge, U.K.: Cambridge Univ. Press, 1915.

[19] B. D. McKay and E. Rogoyski, ''Latin squares of order 10,'' *Electron. J. Combinatorics*, vol. 2, no. 1, Aug. 1995.

[20] H. W. Norton, ''The 7 × 7 squares,'' *Ann. Eugenics*, vol. 9, no. 3, pp. 269–307, 1939.

[21] M. A. Khan, N. Mohammad, S. Muhammad, and A. Ali, ''A mining-based approach for efficient enumeration of algebraic structures,'' *Int. J. Data Sci. Anal.*, vol. 1, no. 2, pp. 89–98, Apr. 2016.

[22] M. A. Khan, N. Mohammad, S. Muhammad, and A. Ali, ''Enumeration of exponent three inverse property loops,'' *Quasigroups Rel. Syst.*, vol. 25, no. 1, pp. 73–86, 2017.

[23] M. A. Khan, ''IP loop algebraic structures of order 7, 9, 11 and 13 with labelled classes, v1,'' Mendeley Data, Elsevier, Mendeley, London, U.K., 2019, doi: 10.17632/bywsggk65r.1.

[24] A. Sade, ''Morphismes de quasigroupes: Tables,'' *Revista da Faculdade de Ciências de Lisboa, 2, A-Ciências Matemáticas*, vol. 13, no. 1, pp. 149–172, 1971.

[25] M. B. Wells, ''The number of Latin squares of order eight,'' *J. Combinat. Theory*, vol. 3, no. 1, pp. 98–99, Jul. 1967.

[26] H. Ehrlich and M. Rarey, ''Maximum common subgraph isomorphism algorithms and their applications in molecular science: A review,'' *WIREs Comput. Mol. Sci.*, vol. 1, no. 1, pp. 68–79, Jan. 2011.

[27] V. R. Rosenfeld, ''Looking into the future of molecules with novel topological symmetries,'' *J. Math. Chem.*, vol. 57, no. 7, pp. 1850–1867, Jun. 2019.

[28] M. Meissner, O. Mitea, L. Luy, and L. Hedrich, ''Fast isomorphism testing for a graph-based analog circuit synthesis framework,'' in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2012, p. 757–762.

[29] P. Wang, M. Niamat, and S. Vemuru, ''Majority logic synthesis based on nauty algorithm,'' in *Field-Coupled Nanocomputing*, Cham, Switzerland: Springer, 2014, p. 111–132.

[30] Y. Li, L. Ding, and Y. W. Li, ''Isomorphic relationships between voltage-source and current-source converters,'' *IEEE Trans. Power Electron.*, vol. 34, no. 8, pp. 7131–7135, Aug. 2019.

[31] T. Boykett, ''Efficient exhaustive listings of reversible one dimensional cellular automata,'' *Theor. Comput. Sci.*, vol. 325, no. 2, pp. 215–247, Oct. 2004.

[32] *OR-Tools: Google's Operations Research Tools*. Accessed: Mar. 1, 2020. [Online]. Available: https://github.com/google/or-tools

[33] *Google's Operations Research Tools Documentation Reference*. Accessed: Mar. 1, 2020. [Online]. Available: https://google.github.io/or-tools/java

[34] *JaCoP: Java Constraint Programming Library*. Accessed: Mar. 1, 2020. [Online]. Available: https://www.lth.se/jacop/

[35] *GECODE: Generic Constraint Development Environment*. Accessed: Mar. 1, 2020. [Online]. Available: https://www.gecode.org/

[36] *A000315: The Online Encyclopedia of Integer Sequences (Latin squares)*. Accessed: Mar. 1, 2020. [Online]. Available: https://oeis.org/A000315

[37] *A057771: The Online Encyclopedia of Integer Sequences (Loop)*. Accessed: Mar. 1, 2020. [Online]. Available: https://oeis.org/A057771

MAJID ALI KHAN (Member, IEEE) received the B.S. degree in computer system engineering from the Ghulam Ishaq Khan Institute (GIKI), Pakistan, in 1997, and the Ph.D. degree from the University of Central Florida (UCF), Orlando, USA, in 2007.

He has extensive software development experience at various multinational companies, including NetSol Technologies Inc., Lahore, Pakistan, and Hewlett-Packard (HP Software), Roseville, CA. He has also worked as the main contributor at several start-up companies. He is currently working as an Assistant Professor with the College of Computer Engineering and Science, Prince Mohammad Bin Fahd University (PMU). He has research interests in machine learning and distributed/parallel computing with applications in several domains, including software engineering, algebraic structures, and modeling of large scale systems.