Faradžev Read-type enumeration of non-isomorphic CC systems

Milan Banković*, Filip Marić

Faculty of Mathematics, University of Belgrade, Studentski trg 16, 11000 Belgrade, Serbia

Abstract

Donald Knuth introduced abstract CC systems to represent configurations of points in a plane with a given orientation (clockwise or counterclockwise) of all triples of points. We present efficient enumeration of all non-isomorphic CC systems with at most 12 points. Our algorithm is based on Faradžev-Read type enumeration, enhanced with the homomorphism principle and SAT solving, enabling us to enumerate more than $1.3 \cdot 10^{12}$ non-isomorphic CC systems with 12 points.

Keywords: point triples orientation, CC systems, Faradžev-Read enumeration, homomorphism principle, SAT solving

1. Introduction

Many conjectures in finite discrete geometry can be tested by examining them on different configurations of points in a plane. Therefore, it is often desirable to list all non-isomorphic configurations of points wrt. some isomorphism relation. In many applications it suffices to consider that two configurations are isomorphic if orientation (clockwise or counterclockwise) of all corresponding triplets of points is the same (without loss of generality, it can be considered that all given points are in a general position i.e., that no three points are collinear). For example, some finite cases of the famous Erdős-Szekeres Happy-Ending conjecture [1] are shown by analyzing all different orientations of triples of points [2, 3]. Orientation can be easily determined from the Cartesian coordinates of the points. However, if the coordinates of points are not given, but only the orientation of all triples of points, the problem of enumerating non-isomorphic configurations becomes a pure combinatorial problem, that may be easier to solve than the more general geometric problem.

Orientations of points cannot be assigned arbitrarily, as in some cases the configurations would not be *geometrically realizable* i.e., it would not be possible to find coordinates of points with such orientation. Knuth [4] introduced *abstract CC systems* to be sets of points with a fixed orientation of all triples of points, where the orientation satisfies some axioms used to rule out configurations that would obviously be geometrically unrealizable. Those axioms give necessary,

*Corresponding author

Email addresses: milan@matf.bg.ac.rs (Milan Banković), filip@matf.bg.ac.rs (Filip Marić)

but not sufficient conditions to realizability, so there exist CC systems that are not geometrically realizable.[1] Nevertheless, for many applications, it is sufficient to enumerate all non-isomorphic CC systems, without testing their realizability, since many interesting conjectures hold for non-realizable CC systems as well.

The problem of enumeration of all non-isomorphic CC systems quickly becomes quite challenging, since the number of non-isomorphic CC systems of a fixed size n (i.e. the number of points) tends to grow very fast as n grows (exponentially in n^2 [4]). Up to our best knowledge, the greatest size for which the number of non-isomorphic CC systems is published is $n = 10$, and that number is 28 627 261 [7].

In this paper we present a novel algorithm for enumerating non-isomorphic CC systems. It is based on Faradžev-Read general algorithm scheme [8, 9]. It is combined with the homomorphism principle [10], that is applied on the structure of nested convex hulls of the set of points in order to reduce the number of tested permutations. An important component of the algorithm is the augmenting operation, which is based on efficient SAT solving, enabling us to enumerate all non-isomorphic CC systems of sizes $n = 11$ and $n = 12$ (there are respectively 4 686 329 954 and 1 382 939 012 729 such systems), using a multi-processor computer. The question of realizability of enumerated CC systems is not considered in this paper. Aside from the obtained numbers (which could be calculated relatively easily from the existing publicly available data generated by other researchers), the main contribution of our work is the method itself, since the presented techniques could be generalized and applied to other problems in the field of combinatorial geometry.

Related work. The concept of abstract CC systems is related to a very similar concept of *abstract order types*. An *order type* of a set of n points in a plane is a mapping that assigns to each triple of points from the set an orientation (clockwise or counterclockwise). The key difference between order types and CC systems is in the way the equivalence relation is defined – while two CC systems are considered equivalent if they can be obtained from each other only by relabeling, two order types are considered equivalent if they can be obtained from each other by both relabeling and mirroring. In other words, each order type may consist of one or two non-isomorphic CC systems, depending on whether the two mirrored CC systems are isomorphic (such CC systems are called *achiral* [7]). Because of the similarity of the two concepts, existing methods for enumeration of order types (as well as publicly available databases) can be employed for enumeration of CC systems and vice versa.

The enumeration of order types seems to be much better covered in the literature than the enumeration of CC systems. One of the first and the most important work on enumeration of order types is done by Aichholzer et al. [5], where this problem is reduced to the problem of enumeration of non-isomorphic arrangements of straight lines by a suitable duality transformation. This problem is, in turn, reduced to the problem of enumeration of *pseudoline arrangements* (i.e. a relaxed version of the problem where simple curves may be used instead of straight lines), which is then solved using the method of *wiring diagrams*

[1] It is known from the literature that all CC systems of up to 8 points are geometrically realizable [5]. The smallest non-realizable CC system consists of nine points, and is constructed from the pseudoline arrangement that violates Pappus' theorem [6].

[11]. Since some pseudoline arrangements may not be *stretchable* (i.e. cannot be transformed to an equivalent straight line arrangement), the corresponding order types may not be geometrically realizable (hence the name *abstract order types*). The authors further consider realizability of the obtained abstract order types to obtain the list of all realizable order types up to the size of 10. The same authors extended their work in [12], where they have provided a complete enumeration of all (abstract and realizable) order types of size 11, using the same method based on the wiring diagrams for pseudoline arrangement enumeration, and the set of enhanced methods for construction of geometric realizations of the obtained abstract order types.

The enumeration of abstract order types of size 11 is considered again recently by Scheucher et al. [13]. Their method is based on augmentation of already enumerated order types of smaller size, in the way that the *signotope axioms* remain satisfied. Their method has shown excellent performance, enumerating all abstract order types of size 11 in just about 20 CPU hours. The problem of realizability of obtained abstract order types is not considered in their work. The obtained database of abstract order types of size 11 is then used to prove that there are no *11-universal sets* of size 11, which is done with the help of a SAT solver.

The problem of enumeration of non-isomorphic CC systems have been also studied in the literature, but to a much lesser extent. For example, Knuth [4] gives quite a thorough analysis of the problem, and gives asymptotic behaviour of the number of non-isomorphic CC systems (exponential in n^2). He also shows the tight connection between CC systems and primitive sorting networks, showing that the number of non-isomorphic CC systems is equal to the number of ,,weakly equivalent reflection networks". He presents the numbers of non-isomorphic CC systems with up to 9 points. A decade later, the number of non-isomorphic CC systems of size 10 is published in [7]. In this work, the techniques based on *halving lines* are used to enumerate all pseudoline arrangements of size 10 (which correspond to abstract order types). Using the number of achiral CC systems of size 10, they obtain the number of all non-isomorphic CC systems of size 10.

Aside from the published work, some data recently made public by Scheucher and Rote suggest the numbers of abstract order types up to the size 13[2], as well as the numbers of mirror-symmetric abstract order types (that is, the number of achiral CC systems) up to the size 13[3]. These numbers allow us to easily calculate the number of non-isomorphic CC systems up to the size 13, using the formula $C_n = 2D_n - R_n$, where C_n is the number of CC systems of size n, D_n is the number of abstract order types of size n, and R_n is the number of achiral CC systems of size n (thus we have C_{11}=4 686 329 954, C_{12} = 1 382 939 012 729, and C_{13} = 732 955 581 630 129, confirming our results for C_{11} and C_{12}). As far as we know, these numbers were not officially published so far.

SAT solving has recently been applied to efficient solving of several problems in discrete finite geometry (e.g., [2, 14, 15, 13]). Marić used a SAT solver to confirm the Erdös-Szekeres Happy-Ending conjecture for polygons with at most 6 points [2], and Balko and Valtr used a SAT solver to refute the conjecture of

[2] `https://oeis.org/A006247`
[3] `http://oeis.org/A325628`

Peters and Szekeres (a stronger variant of Erdös and Szekers conjecture). Beside mentioned work of Scheucher et al. [13], another interesting application is given by Scheucher in [14]. In this work, with computer assistance, it is shown that every set of 17 points in general position in a plane admits two disjoint 5-holes. The allowable configurations of points are encoded using signotope axioms, and additional constraints are added to forbid disjoint 5-holes. The unsatisfiability result for the set of 17 points proves the conjecture.

Outline of the paper. In Section 2 we give some background on Faradžev-Read type algorithms, SAT solvers and CC systems. In Section 3 we describe our algorithm and prove its correctness. In Section 4 we give some details of its C++ implementation and show experimental results. In Section 5, we provide a more detailed comparison of our work to other relevant approaches in the literature. Finally, in Section 6 we draw some conclusions and describe some directions of further work.

2. Preliminaries

In this section we describe some background on Faradžev-Read algorithm, SAT solving and CC systems.

2.1. Faradžev-Read algorithm

Faradžev-Read algorithm is a very general scheme for exhaustive isomorph-free enumeration of combinatorial objects, developed independently by Faradžev [8] and Read [9], and applied on many different problems (e.g., [16, 17]).

Let us assume that each considered combinatorial object has the associated *size* (for instance, this can be the number of nodes in a graph, or the number of elements in a finite set). Let S_n denote the set of all objects of size n. We also assume an equivalence relation $\sim$ on S_n. We say that two objects x and y are *isomorphic* with respect to $\sim$ if and only if $x \sim y$. The goal is to form a list L_n of objects from S_n containing exactly one representative of each equivalence class of $\sim$.

The approach used by Faradžev and Read is based on augmentation of objects of smaller size. Let us assume that we have already constructed a list L_{n-1} of all non-isomorphic objects of size $n-1$. The list L_{n-1} is traversed, and an appropriate *augmenting operation* is applied to each element $x \in L_{n-1}$, producing (zero or more) objects of size n. Each such object is appended to L_n if and only if it is not isomorphic to any of the objects already present in L_n. A naive method would be to compare this new object to all objects already in L_n, but this is usually too expensive, since the number of objects in L_n may grow very fast. In order to cope with that problem, the notion of *canonicity* is introduced – for each equivalence class we choose one canonical representative, and there should exists an effective method for checking whether some object is the canonical representative (*canonical object*) of its class or not. Assuming the list L_{n-1} contains exactly all canonical objects of size $n-1$, we can form the list L_n by augmenting the canonical objects from L_{n-1}, where only the augmented objects that pass the canonicity test are appended to L_n. In order to make such enumeration exhaustive, the following property of the augmenting operation must hold:

Property 1: each canonical object y of size n can be obtained by augmenting some canonical object x of size $n-1$ (otherwise there would certainly exist some canonical objects of size n that would not be added to L_n).

If the same canonical object y of size n can be obtained by augmentation of more than one object of size $n-1$, the problem of duplicates in L_n may arise. Again, checking whether the object is already in L_n before the addition may be too time consuming, so it is important to avoid it. For this reason, we further assume that there exists a linear order over the objects of the same size n, denoted by $\prec_n$ (or $\prec$, if the size is clear from the context). Now each newly constructed canonical object of size n is appended to L_n only if it is greater (with respect to $\prec_n$) than the last object already in the list L_n. This way, we avoid comparing the newly constructed object to all objects present in L_n, and also guarantee that the constructed list L_n will be sorted in the ascending order with respect to $\prec_n$, avoiding duplicates. However, this approach may spoil the exhaustiveness, unless the following properties hold:

Property 2: if x and y are canonical objects of size n, and $x \prec y$, and if x' and y' are, respectively, the smallest canonical objects of size $n-1$ (with respect to $\prec_{n-1}$) from which x and y can be obtained by augmentation, then $x' \preceq y'$.

Property 3: if $y_1, y_2, \ldots, y_k$ is the sequence of objects of size n obtained by repeated augmentation of some fixed object x of size $n-1$, then $y_1 \prec y_2 \prec \ldots \prec y_k$.

These two properties guarantee that the newly generated object y of size n that passed canonicity test is not already present in L_n if and only if $y' \prec y$, where y' is the last object in L_n. Indeed, if $y' \prec y$, then we know that y is not present in L_n, since L_n is sorted in the ascending order. On the other hand, if $y \preceq y'$, then we know that y is already present in L_n, since we know that the object y must have been constructed for the first time before y', because of the Property 2 and Property 3, and the fact that L_{n-1} is sorted in the ascending order.

The general Faradžev-Read algorithm scheme is given in Algorithm 1. It is parameterized by two procedures: `is_canonical`(x) which is used for canonicity testing, and `augment`(x) which represents the augmenting operation. These two procedures, as well as the linear order $\prec$ are specific to the concrete type of objects being enumerated and must satisfy the above three properties.

In a special case, when each canonical object y of size n can be obtained by augmentation of only one canonical object x of size $n-1$, only Property 1 should be satisfied, and the linear order $\prec$ is not needed. In that case, the simplified version of Faradžev-Read algorithm scheme given in Algorithm 2 may be used.

Algorithm 1 and 2 operate in a *breadth-first search* (BFS) manner. Since the whole list L_n is stored in memory at once, this can be very memory consuming. A *depth-first search* (DFS) based variant of Faradžev-Read algorithm is given in Algorithm 3. In essence, the algorithm does not form the lists L_m of canonical objects of size $m < n$. Instead, for each canonical object x of size $m < n$ it creates the list of its augmentations $[y_1, \ldots, y_k]$, and for each of them which passes the canonicity test, recursively invokes the same procedure. Canonical objects of size n are collected and returned in the list L_n.

Require: L_{n-1} is an exhaustive list of canonical objects of size $n-1$, sorted in the ascending order wrt. $\prec$
Require: `is_canonical`(x) returns *true* iff x is the canonical representative of its class
Require: `augment`(x) returns the sorted list of all objects of size n that can be obtained by augmenting the object x of size $n-1$
Ensure: L_n is an exhaustive list of canonical objects of size n, sorted in the ascending order wrt. $\prec$
begin
$L_n = [\,]$ {L_n is initially empty}
for all $x \in L_{n-1}$ **do**
$[y_1, \ldots, y_k] =$ `augment`(x)
for all $y \in [y_1, \ldots, y_k]$ **do**
{Let y' be the last object added to L_n}
if `is_canonical`$(y) \wedge y' \prec y$ **then**
$L_n = L_n, y$
end

Algorithm 1: `faradzev_read`$_{\texttt{is\_canonical},\texttt{augment},\prec}(L_{n-1})$

Require: L_{n-1} is an exhaustive list of canonical objects of size $n-1$
Require: `is_canonical`(x) returns *true* iff x is the canonical representative of its class
Require: `augment`(x) returns the list of all objects of size n that can be obtained by augmenting the object x of size $n-1$
Ensure: L_n is an exhaustive list of canonical objects of size n
begin
$L_n = [\,]$ {L_n is initially empty}
for all $x \in L_{n-1}$ **do**
$[y_1, \ldots, y_k] =$ `augment`(x)
for all $y \in [y_1, \ldots, y_k]$ **do**
if `is_canonical`(y) **then**
$L_n = L_n, y$
end

Algorithm 2: `faradzev_read_simple`$_{\texttt{is\_canonical},\texttt{augment}}(L_{n-1})$

2.2. SAT solvers

In this section we assume the standard syntax and semantics of propositional logic [18]. Let P be a set of *propositional atoms*. A *literal* is either a propositional atom p from P or its negation $\neg p$. A *clause* is a disjunction of literals. Because of the commutativity and associativity of the disjunction, we may consider clauses as sets of literals. A propositional formula in *conjunctive normal form* (*CNF formula*) is a conjunction of clauses. For the similar reason, the CNF formula may be considered as a set of clauses.

A *valuation* v over P is a (partial) assignment of boolean values to the propositional atoms of P. A valuation can be naturally extended to literals over P: if p is *true* in v, then the literal $\neg p$ is *false* in v, and vice-versa. Since the valuation may be partial, some atoms (and literals) may be *undefined* in v. The valuation v may be identified with the set of literals that are true in v. For

Require: x is a canonical object of size m
Ensure: L_n is an exhaustive list of canonical objects of size n that are descendants of x
  begin
  $L_n = [\,]$ {L_n is initially empty}
  $[y_1, \ldots, y_k] =$ `augment`(x)
  for all $y' \in [y_1, \ldots, y_k]$ **do**
    if `is_canonical`(y') **then**
      if $m + 1 = n$ **then**
        $L_n = L_n \cup \{y'\}$
      else
        $L_n = L_n \cup$ `faradzev_read_dfs`$(y', m + 1, n)$
  return L_n
  end

Algorithm 3: `faradzev_read_dfs`(x, m, n)

instance, if $P = \{p, q, r\}$ and p is true, q is false, and r is undefined in v, such valuation may be represented by the set of literals $\{p, \neg q\}$.

Problem of *boolean satisfiability* (or *SAT problem*) is the problem of checking if there exists a valuation that satisfies a given CNF formula. A CNF formula is satisfied in a valuation v if all its clauses are true in v (i.e. all its clauses contain at least one literal that is true in v). SAT problem is the one of the most famous NP-complete problems [19], and also the problem with a great number of applications in different domains [18].

The software systems that implement decision procedures for SAT problem are called *SAT solvers*. Most of the state-of-the-art SAT solvers are based on CDCL algorithm (*conflict driven clause learning*) [20], which is an improved version of DPLL algorithm from 1962 [21]. DPLL algorithm tries to incrementally build a satisfying valuation, by adding literals one by one to the partial valuation stack, and backtracking when the current partial valuation falsifies some clause of the formula being solved. The algorithm also incorporates methods of inference, such as *unit propagation* and *pure literal*. CDCL algorithm additionally includes *conflict analysis* based on *resolution*, *non-chronological backtracking*, *clause learning* and *restarting*. Finally, efficient implementation techniques such as *two-watched-literal scheme* for fast exploration of the clause database and smart branching heuristics are also an important part of modern SAT solvers [22]. Thanks to all the algorithmic and implementational improvements, modern SAT solvers are able to solve problems with thousands of propositional atoms involved, and hundreds of thousands of clauses.

A variation of the basic SAT problem, known as *All-SAT*, is the problem of enumeration of all satisfying valuations for a given CNF formula. The simplest way to enumerate all solutions in case of pure DPLL-based solver is to explicitly backtrack the solver whenever it finds a complete satisfying valuation. In case of CDCL SAT solvers, the simplest way is to add *the blocking clause* each time a satisfying valuation v is found: such clause consists of all literals that are false in v. That way, solver is forced to search for a different solution after being restarted. There are also more efficient approaches that avoid addition of long blocking clauses that tend to slow down the solver, some of them are presented in [23].

2.3. CC system

In this section we introduce a notion of an abstract *CC system*, first defined by Donald Knuth [4]. Assume an universe P whose elements are called *points*, and a ternary relation *ccw*, satisfying the following axioms:

Ax0: $\forall pqr.\ ccw(pqr) \Rightarrow p \neq q \wedge q \neq r \wedge p \neq r$
Ax1: $\forall pqr.\ ccw(pqr) \Rightarrow ccw(qrp)$
Ax2: $\forall pqr.\ ccw(pqr) \Rightarrow \neg ccw(prq)$
Ax3: $\forall pqr.\ p \neq q \wedge p \neq r \wedge q \neq r \Rightarrow ccw(pqr) \vee ccw(prq)$
Ax4: $\forall pqrt.\ ccw(pqt) \wedge ccw(qrt) \wedge ccw(rpt) \Rightarrow ccw(pqr)$
Ax5: $\forall pqrts.\ ccw(tsp) \wedge ccw(tsq) \wedge ccw(tsr) \wedge ccw(tpq) \wedge ccw(tqr) \Rightarrow ccw(tpr)$

Any such structure $\mathbb{P} = (P, ccw)$ is called a *CC system*. The most natural model of the above axioms is a non-empty set P of points in a plane *in a general position* (meaning that there are no three collinear points in P), where the predicate *ccw* is defined as follows:

$$ccw(pqr) \quad \Leftrightarrow \quad \begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix} > 0$$

assuming that (x_p, y_p), (x_q, y_q) and (x_r, y_r) are the coordinates of the points p, q and r, respectively. In this interpretation, $ccw(pqr)$ denotes that the triple (p, q, r) is counter-clockwise oriented in the plane (hence the name of the predicate *ccw*).

Two CC systems $\mathbb{P}_1 = (P_1, ccw_1)$ and $\mathbb{P}_2 = (P_2, ccw_2)$ are *isomorphic* if there is a bijective function $\pi : P_1 \to P_2$ such that for each three distinct points p,q and r in P_1 it holds that $ccw_1(pqr) \Leftrightarrow ccw_2(\pi(p)\pi(q)\pi(r))$.

3. Enumeration of non-isomorphic finite CC systems

In this section, an instance of the Faradžev-Read algorithm scheme for enumeration of non-isomorphic CC system of finite sizes is described. First, a compact way for representing finite CC systems is defined. Such representations will be called *configurations*. In order to shrink the search space, we will analyze the structures of CC systems in more details. The results of such analysis will enable us to focus on a special class of configurations, which will be called *regular configurations*. We will also define the notion of *canonical configurations*, which will be regular configurations that correspond to the canonical representatives of the isomorphism classes of CC systems. Then we will present the algorithm for checking the canonicity of a configuration in an efficient way. Finally, we will describe the augmenting operation, which will be implemented by utilizing an All-SAT solver.

3.1. Finite CC systems and their configurations

In case of finite CC systems, we will always assume that the points are denoted by the natural numbers $0, 1, \ldots, n-1$, where n is the number of points in the system. In other words, we will always assume the CC systems of the form $\mathbb{P} = (P_n, ccw)$, where $P_n = \{0, 1, \ldots, n-1\}$. This means that distinct finite CC systems of the same size differ only in the way their *ccw* predicates

are defined, and in order to represent a finite CC system (as a combinatorial object), it is sufficient to represent its *ccw* predicate. In the following text we formally define the representation that is used in this work.

Let us denote the set of all *triples pqr* (where *pqr* is just a shorthand notation for (p, q, r)) of distinct points from P_n by $\mathcal{T}_n$. We say that a triple $pqr \in \mathcal{T}_n$ is *normalized*, if $p < q < r$. The set of all normalized triples of P_n will be denoted by $\mathcal{N}_n$. It holds that $|\mathcal{N}_n| = \binom{n}{3} = \frac{n \cdot (n-1) \cdot (n-2)}{6}$. We denote by $normalize(pqr)$ the triple $p'q'r'$ obtained from pqr by sorting its points in the ascending order. The triple pqr is *positive* if it can be obtained from $normalize(pqr)$ by cycling its points, and *negative* otherwise. For instance, triples 012, 120 and 201 are positive, and 021, 102 and 210 are negative (the normalized form of all six triples is 012). From Ax1-Ax3, it follows that if a triple pqr is positive, then $ccw(pqr) \Leftrightarrow ccw(p'q'r')$, and if pqr is negative, then $ccw(pqr) \Leftrightarrow \neg ccw(p'q'r')$, where $p'q'r' = normalize(pqr)$. In both cases, the *ccw*-value of a triple is reduced to the *ccw*-value of its normalized form.

For a fixed CC system $\mathbb{P}$ of n points, we define its *configuration* as a function $c : \mathcal{N}_n \to \{0, 1\}$ defined as follows:

$$c(pqr) \quad = \quad \begin{cases} 1, & \text{if } ccw(pqr) \\ 0, & \text{if } \neg ccw(pqr) \end{cases}$$

From the above discussion it follows that the configuration of a CC system fully defines its *ccw* predicate (and, therefore, the CC system itself), so we can identify finite CC systems with their configurations. We say that the configuration c is *of size* n if it corresponds to a CC system of n points.

We further define a linear order $(\mathcal{N}_n, <)$ over normalized triples by using reverse lexicographic comparison in the following way: $p_1q_1r_1 < p_2q_2r_2$ iff:

- $r_1 < r_2$, or
- $r_1 = r_2$ and $q_1 < q_2$, or
- $r_1 = r_2$ and $q_1 = q_2$ and $p_1 < p_2$

For instance, for the CC system of 5 points (denoted by $0, 1, 2, 3, 4$), the normalized triples are ordered in the following fashion: 012, 013, 023, 123, 014, 024, 124, 034, 134, 234. Notice that the order is such that for each point p, the triples containing p are placed after all the triples composed of the points smaller than p are exhausted. By $position(pqr)$ we denote the index of the position of the normalized triple pqr in the above linear order, assuming that the indexing is zero-based. It can be easily shown that $position(pqr) = \binom{r}{3} + \binom{q}{2} + \binom{p}{1}$.

Having in mind the order of normalized triples, configurations of size n may be written as binary strings of length $\binom{n}{3}$, where the i-th bit (counted from left to right, starting from zero) corresponds to the value $c(pqr)$, where $position(pqr) = i$ (e.g. leftmost bit corresponds to $c(012)$, the next bit corresponds to $c(013)$), etc). Such string will also be denoted by c, and its i-th bit by c_i. Notice that not all binary strings of length $\binom{n}{3}$ correspond to legal configurations, since the axioms Ax4 and Ax5 of CC systems may be violated. We say that a configuration is *feasible* if the *ccw* predicate it defines satisfies the axioms of CC systems. We will denote the set of all feasible configurations of size n by S_n.

If a configuration c of size n is given as a binary string, then its prefix c' of length $\binom{n-1}{3}$ corresponds to a configuration of size $n - 1$ of the CC system obtained by removing the point $n - 1$ from the CC system determined by c,

due to the defined order of normalized triplets (i.e. the first $\binom{n-1}{3}$ positions in c correspond to the triplets not containing the point $n-1$ which we removed from the system). We say that c' is *the parent configuration* of c, and also that c is *a child configuration* of c'. Obviously, each configuration has unique parent, while it may have more than one child configurations.

Lemma 3.1. *If c is a feasible configuration of size n, then its parent c' is a feasible configuration of size $n-1$.*

Proof. Since c is a feasible configuration of size n, it satisfies all the axioms of CC systems for the points $0, 1, \ldots, n-1$. Its prefix of length $\binom{n-1}{3}$ represents the configuration c' of size $n-1$ obtained from c by removing the point $n-1$, so it still satisfies all the axioms of CC systems for the points $0, 1, \ldots, n-2$. Thus, it is a feasible configuration. □

Two configurations c_1 and c_2 of size n are *isomorphic* if the CC systems they define are isomorphic. Isomorphisms between configurations correspond to permutations of the set of points $\{0, 1, \ldots, n-1\}$ that are homomorphic with respect to the *ccw* predicate.

Lemma 3.2. *Two configurations c_1 and c_2 of size n are isomorphic if and only if there exists a permutation π of $P_n = \{0, 1, \ldots, n-1\}$, such that:*

$$c_1(pqr) = \begin{cases} c_2(normalize(\pi(p)\pi(q)\pi(r))), & \text{if } \pi(p)\pi(q)\pi(r) \text{ is positive} \\ 1 - c_2(normalize(\pi(p)\pi(q)\pi(r)), & \text{otherwise} \end{cases}$$

Proof. Let $\mathbb{P}_1 = (P_n, ccw_1)$ and $\mathbb{P}_2 = (P_n, ccw_2)$ be the CC systems that correspond to the configurations c_1 and c_2, respectively. By definition, if c_1 and c_2 are isomorphic, then there exists a permutation π of the set P_n such that $ccw_1(pqr) \Leftrightarrow ccw_2(\pi(p)\pi(q)\pi(r))$ for any triple pqr. Assume that pqr is a normalized triple, and let $p'q'r' = normalize(\pi(p)\pi(q)\pi(r))$. If $\pi(p)\pi(q)\pi(r)$ is positive, then $ccw_2(\pi(p)\pi(q)\pi(r)) \Leftrightarrow ccw_2(p'q'r')$, and if $\pi(p)\pi(q)\pi(r)$ is negative, then $ccw_2(\pi(p)\pi(q)\pi(r)) \Leftrightarrow \neg ccw_2(p'q'r')$. The lemma now follows from the definition of a configuration. □

Lemma 3.3. *For each configuration c of size n and for each permutation π of the set $P_n = \{0, 1, \ldots, n-1\}$ there exists a unique configuration c' of size n such that π is the isomorphism from c to c'.*

Proof. Let $\mathbb{P}_1 = (P_n, ccw_1)$ be the CC system determined by the configuration c, and let $\mathbb{P}_2 = (P_n, ccw_2)$ be the CC system such that for all triples pqr it holds:

$$ccw_2(pqr) \Leftrightarrow ccw_1(\pi^{-1}(p)\pi^{-1}(q)\pi^{-1}(r))$$

where π^{-1} is the inverse of the permutation π. The permutation π will be the isomorphism from $\mathbb{P}_1$ to $\mathbb{P}_2$, since for each triple pqr, and their images $p' = \pi(p)$, $q' = \pi(q)$ and $r' = \pi(r)$ it holds:

$$\begin{aligned} ccw_1(pqr) &\Leftrightarrow ccw_1(\pi^{-1}(p')\pi^{-1}(q')\pi^{-1}(r')) \\ &\Leftrightarrow ccw_2(p'q'r') \\ &\Leftrightarrow ccw_2(\pi(p)\pi(q)\pi(r)) \end{aligned}$$

It is easy to see that the above definition of the ccw_2 predicate is the only possible definition such that π is a homomorphism between $\mathbb{P}_1$ and $\mathbb{P}_2$, hence the uniqueness. The configuration c' will be the configuration corresponding to the CC system $\mathbb{P}_2$. □

The configuration c' obtained from c by applying the permutation π will be denoted by $c\pi$.

3.2. Checking canonicity

The isomorphism relation partitions the set of all feasible configurations S_n into the isomorphism classes. For each isomorphism class, we must choose its *canonical representative*. One way to do it is to choose the configuration that corresponds to the lexicographically smallest binary string in the class. In that case, in order to prove the canonicity of a configuration c, it is sufficient to prove that for any permutation of points π, the configuration $c\pi$ is not lexicographically smaller than c. The problem with this approach is that the number of such permutations is $n!$, which grows exponentially with n. Therefore, it would be beneficial if we could reduce the number of permutations that need to be tested.

A general approach is to find some properties that are invariant under isomorphism relation and than to consider only the permutations that maintain such invariant properties. For example, node input and output degrees are invariant under graph isomorphisms, so only permutations of nodes that maintain their degrees need to be considered when checking graph canonicity. An invariant that we found useful for checking canonicity of CC systems is the structure of their nested convex hulls [2].

3.2.1. Configuration hull structures and regular configurations

Assume a CC system $\mathbb{P} = (P, ccw)$ and a non-empty set of points $S \subseteq P$. The *convex hull* of S is the *list* of distinct points $H = [p_0, p_1, \ldots, p_k]$ from S such that for each pair of consecutive points p_i, p_{i+1} (and also for the pair p_k, p_0) and for each point $r \in S$ distinct from these two points it holds that $ccw(p_i p_{i+1} r)$ (and also $ccw(p_k p_0 r)$). It can be shown that every finite set of points always has a convex hull [4, 2]. Specially, if S is a singleton set, or has two elements, its convex hull is S itself, by definition. Otherwise, the convex hull of S has at least three points.

After we find the convex hull H_0 of a finite set S, we can proceed to find the convex hull H_1 of the set $S \setminus H_0$, then the convex hull H_2 of the set $S \setminus (H_0 \cup H_1)$, etc. This process can continue until the empty set is reached. Convex hulls $H_0, H_1, H_2, \ldots, H_m$ obtained in that way are called the *nested convex hulls* of the set S (notice that $S = H_0 \sqcup H_1 \sqcup \ldots \sqcup H_m$). If we denote by h_i the number of points in the hull H_i, then the list $[h_0, h_1, \ldots, h_m]$ is called the *hull structure* of the set S. Notice that $h_i \geq 3$ for $i < m$, while h_m may be any number greater than zero.

Assume now a finite CC system $\mathbb{P} = (P_n, ccw)$ of size n. Let $H_0, \ldots, H_m$ be the nested convex hulls of the set $P_n = \{0, 1, \ldots, n-1\}$ (that is, the nested convex hulls of the entire CC system). Let $[h_0, \ldots, h_m]$ be its hull structure. The CC system $\mathbb{P}$ is *regular* if H_0 contains the points $0, 1, \ldots, h_0 - 1$ (in any order), H_1 contains the points $h_0, \ldots, h_0 + h_1 - 1$ (in any order), H_2 contains the points $h_0 + h_1, \ldots, h_0 + h_1 + h_2 - 1$, etc. For instance, if the hull structure of a system of 10 points is $[4, 3, 3]$, then the system is regular if its convex hull H_0 contains the points $0, 1, 2, 3$ (in any order), the next hull H_1 contains points $4, 5, 6$ (in any order), and the remaining points $7, 8, 9$ make the innermost hull H_2 (again, in any order). A configuration c is *regular* if it represents a regular CC system.

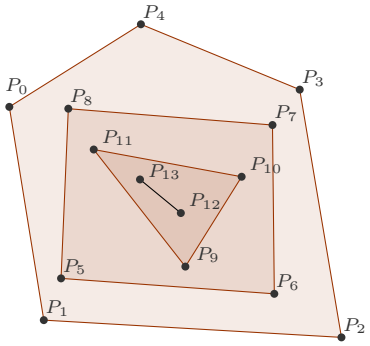

Figure 1: Nested convex hulls with the structure $[5, 4, 3, 2]$.

The important fact about regular CC systems is given in the following lemma.

Lemma 3.4. *Every finite CC system is isomorphic to some regular CC system of the same size.*

Proof. Let $\mathbb{P} = (P_n, ccw)$ is an arbitrary CC system of size n, and let $H_0, H_1, \ldots, H_m$ be the list of its nested convex hulls. As before, we denote its hull structure as $[h_0, \ldots, h_m]$. We can construct the permutation π such that points of H_0 are mapped to the points $0, 1, \ldots, h_0 - 1$, the points of H_1 are mapped to the points $h_0, h_0 + 1, \ldots, h_0 + h_1 - 1$, etc. The obtained CC system will be a regular CC system. □

A consequence of Theorem 3.4 is that we may consider only regular configurations, since each equivalence class of S_n has at least one regular member. Unfortunately, the lexicographically smallest configuration of a class does not have to be regular. For this reason, we change the definition of the canonical representative: the configuration is *canonical* if it is the lexicographically smallest *regular* configuration of its class.

A permutation is *regular* with respect to a regular configuration c, if it transforms c to another regular configuration c'. The next lemma claims that regular permutations just permute the points within each of the nested hulls of the CC system represented by c.

Lemma 3.5. *A permutation π is a regular permutation with respect to a regular configuration c of the hull structure $[h_0, h_1, \ldots, h_m]$ if and only if π is of the form $\pi = \pi_0\pi_1 \ldots \pi_m$, where π_i is a permutation of the points of the nested hull H_i.*

Proof. If π is of the form $\pi_0\pi_1 \ldots \pi_m$, then for each hull H_i, its points stay in that hull with only their order being permuted, so the configuration $c\pi$ is regular. Conversely, if π is a regular permutation with respect to c, then $c' = c\pi$ is a regular configuration. It is easy to see that isomorphisms preserve the hull structure of a configuration. This means that c' will have the same hull structure $[h_0, h_1, \ldots, h_m]$. Let $H'_0, H'_1 \ldots, H'_m$ be the nested hulls of c'. Since c' is regular, for each i, H_i and H'_i will be composed of the same points, possibly in different order. Therefore, the permutation π is a composition of permutations of the individual hulls. □

Therefore, in order to check whether a regular configuration c is canonical, we should lexicographically compare it to all configurations of the form $c\pi$, where π is a regular permutation with respect to c. If c has the hull structure $[h_0, h_1, \ldots, h_m]$, then the number of regular permutations is $h_0! \cdot h_1! \cdot \ldots \cdot h_m!$. Compared to $n! = (h_0 + h_1 + \ldots + h_m)!$, it is usually much smaller number of permutations to check.

We shall apply the Faradžev-Read algorithm so that for each n we maintain the list L_n of canonical (regular) configurations. In order to guarantee the algorithm correctness, we must ensure Property 1, i.e., that each canonical regular configuration of size n can be obtained by augmenting some canonical regular configuration of size $n-1$. First we prove that the parent of a regular configuration is also regular, as the following lemma claims.

Lemma 3.6. *Let c be a regular configuration of size n with the hull structure $[h_0, h_1, \ldots, h_m]$. Then its parent c' is a regular configuration of size $n-1$ with the hull structure $[h_0, h_1, \ldots, h_m - 1]$ if $h_m > 1$, or $[h_0, h_1, \ldots, h_{m-1}]$, if $h_m = 1$.*

Proof. Recall that the parent configuration c' correspond to the CC system with the point $n-1$ removed. Since c is regular, the point $n-1$ belongs to the innermost hull, so its removal only changes that hull, while other hulls stay the same, and the system stays regular. □

We will refer to the hull structure of the parent configuration as the *parent hull structure*. On the other hand, if we have a regular configuration c' of size $n-1$, its child configuration c of size n will be regular if and only if (1) the point $n-1$ is added to the innermost hull of c', or (2) is added as a new, singleton hull inside the innermost hull of c' (provided that the innermost hull of c' has at least three points). Therefore, if the hull structure of c' is $[h_0, h_1, \ldots, h_m]$, then the hull structure of child configurations must be either $[h_0, h_1, \ldots, h_m + 1]$ or $[h_0, h_1, \ldots, h_m, 1]$ (the second one is permitted only if $h_m \geq 3$). We refer to these hull structures as the *child hull structures* of the hull structure $[h_0, h_1, \ldots, h_m]$.

Now we prove that the parent of canonical configuration must also be canonical (ensuring the Property 1).

Lemma 3.7. *If a configuration c is a canonical configuration of size n, then its parent configuration c' is a canonical configuration of size $n-1$.*

Proof. Since c is canonical, and thus regular by definition, according to Theorem 3.6, the parent configuration c' is also a regular configuration. Therefore, it suffices to prove that c' is the lexicographically smallest regular configuration in its class.

Assume the opposite, that there exists a regular permutation π' of the set $\{0, 1, \ldots, n-2\}$ (with respect to c') such that $c'\pi'$ is lexicographically smaller than c'. Let π be the permutation of the set $\{0, 1, \ldots, n-1\}$ such that $\pi(p) = \pi'(p)$ for each $p < n-1$, and $\pi(n-1) = n-1$. Then π will be a regular permutation with respect to c and $c'\pi'$ will be the parent configuration of the configuration $c\pi$. Therefore, $c\pi$ will be a regular configuration lexicographically smaller than c, which contradicts the fact that c is a canonical configuration. □

3.2.2. Optimizing the canonicity test

So far we have established that to test the canonicity of a regular configuration c, we must compare c to all the configurations of the form $c\pi$, where

π is a regular permutation with respect to c. Assume that the structure of c is $[h_0, h_1, \ldots, h_m]$, and that its nested hulls are $H_0, H_1, \ldots, H_m$. Each regular permutation permutes only the points within the nested hulls, so it can be decomposed as $\pi = \pi_0\pi_1 \ldots \pi_m$, where π_i is a permutation of the points within the hull H_i. At the first glance, it seems that all $h_0! \cdot h_1! \cdot \ldots \cdot h_m!$ combinations of these hull permutations should be checked, seeking a permutation that yields a lexicographically smaller configuration and disproving the canonicity of c, but it turns out that this could be further optimized.

Checking regular permutations incrementally. The key idea is to analyze all regular permutations incrementally, using a branch-and-bound algorithm for early pruning of the permutations that cannot produce a smaller configuration and disprove canonicity of c. Namely, since c is regular, due to the ordering of configuration triples, the prefix of c of length $\binom{h_0}{3}$ is only affected by permutations of the outermost hull H_0. For each permutation π_0 of H_0 we compare the prefix $c\pi_0$ of length $\binom{h_0}{3}$ to the prefix of c of the same length.

- If the prefix $c\pi_0$ is lexicographically smaller, we already know that c is not canonical (as the permutation π_0 could trivially be extended to a full regular permutation π that would give a lexicographically smaller regular configuration).
- If the prefix of $c\pi_0$ is lexicographically greater, then π_0 cannot be extended to a full regular permutation π such that $c\pi$ is lexicographically smaller than c, so all regular permutations that begin with π_0 can be skipped when trying to disprove the canonicity of c.
- Only if the two prefixes are equal, then we proceed by extending π_0 by checking permutations π_1 of H_1. Such permutations π_0 of H_0 that fix the prefix of c of the length $\binom{h_0}{3}$ are called 0-*automorphisms*.

We can go further with this approach. In general, an *i-automorphism* of the configuration c is a permutation $\pi_0\pi_1 \ldots \pi_i$ such that $\pi_0\pi_1 \ldots \pi_{i-1}$ is an $(i-1)$-automorphism of c, and π_i is a permutation of H_i such that the prefix of $c\pi_0\pi_1 \ldots \pi_i$ of length $\binom{h_0+h_1+\ldots+h_i}{3}$ is identical to the prefix of c of the same length. Notice that for a fixed $(i-1)$-automorphism $\pi_0\pi_1 \ldots \pi_{i-1}$, there may be no i-automorphisms that extend it.

When checking canonicity of c, for each $(i-1)$-automorphism $\pi_0\pi_1 \ldots \pi_{i-1}$ of c all permutations π_i of the hull H_i are analyzed by comparing prefixes of length $\binom{h_0+\ldots+h_i}{3}$ of $c\pi_0 \ldots \pi_i$ and c.

- If the permutation $\pi_0 \ldots \pi_i$ gives lexicographically smaller prefix, then the configuration c is not canonical.
- If it gives a lexicographically larger prefix, then its extensions cannot disprove canonicity of c and need not be analyzed.
- Otherwise $\pi_0 \ldots \pi_i$ is an i-automorphism and it is further examined by considering the permutations of H_{i+1}.

At the final stage, *m-automorphisms* of c correspond to regular permutations of the form $\pi = \pi_0\pi_1 \ldots \pi_m$ such that $c\pi = c$. We simply call such regular permutations *automorphisms* of c. The set of automorphisms of a configuration c

is always non-empty, since the identity permutation is always an automorphism. Automorphisms do not disprove the canonicity of c.

Cyclic permutations of the outermost hull. The outermost hull has some special properties and it turns out that not all its $h_0!$ permutations need to be explicitly analyzed. Namely, the points of the outermost hull H_0 can always be permuted such that the prefix of the obtained configuration of the length $\binom{h_0}{3}$ contains all zero bits, which is the lexicographically smallest prefix of that length. For instance, such case is when $H_0 = [h_0-1, h_0-2, \ldots, 1, 0]$. Therefore, all canonical configurations have prefixes containing of $\binom{h_0}{3}$ zeroes, and the 0-automorphisms are exactly the cyclic permutations of H_0. Other permutations of H_0 need not be considered, since they make this prefix greater.

Child hull structure. Further optimization comes from the fact that during Faradžev-Read enumeration we do not check canonicity of arbitrary configurations, but only those obtained by adding the innermost point to a parent configuration, that has already been checked and shown to be canonical. Checking canonicity of such child configurations with m nested hulls is faster if we know all its $(m-1)$-automorphisms in advance, and it turns out that we can easily learn those during canonicity check of the parent. Namely, child configurations have similar hull structures as their parents (with all nested hulls identical, except the innermost), so all i-automorphisms, for $0 \leq i < m$, of a child configuration c are the same as the i-automorphisms of its parent configuration c'. Since parents are canonical, only permutations that could possibly disprove the canonicity of a child configuration c are extensions of their $(m-1)$-automorphisms by some permutation π_m of H_m. We can optimize the canonicity testing if for each canonical parent configuration c' with the structure $[h_0, h_1, \ldots, h_{m'}]$, we learn and store all its m'-automorphisms and $(m'-1)$-automorphisms.

- If its child c has the structure $[h_0, h_1, \ldots, h_m] = [h_0, h_1, \ldots, h_{m'}+1]$, then its $(m-1)$-automorphisms will be the same as the $(m'-1)$-automorphisms of c'. We only have to check their extension by permutations of the innermost hull $H_{m'} \cup \{n-1\}$.
- If the child c has the structure $[h_0, h_1, \ldots, h_m] = [h_0, h_1, \ldots, h'_m, 1]$, then $(m-1)$-automorphisms of c will be the same as the m'-automorphisms of c'. For each of them, there is only one regular permutation π of c that extends it, since its H_m hull consists only of the point $n-1$, so it must be $\pi(n-1) = n-1$.

In both cases, if c turns out to be canonical, we store its $(m-1)$-automorphisms and m-automorphisms to facilitate checking canonicity of its further extensions.

Since the number of i-automorphisms of configurations with several nested hulls rapidly drops (often to one), this technique very quickly reduces the number of permutations that are checked to just a handful.

Permuting individual hulls. The final optimization considers enumeration of the permutations of the individual hulls. Assume we have a fixed $(i-1)$-automorphism $\pi_0\pi_1 \ldots \pi_{i-1}$ and we want to check the permutations of the hull H_i. There are $h_i!$ such permutations, but we do not have to examine all of

them. Instead, we can again construct permutations incrementally. The hull H_i consists of the points $s, s+1, \ldots, s+h_i-1$, where $s = h_0 + h_1 + \ldots + h_{i-1}$, and the configuration $c\pi_0 \ldots \pi_{i-1}$ is already defined on all normalized triples pqr, such that $p < q < r < s$, since the $(i-1)$-automorphism $\pi_0\pi_1 \ldots \pi_{i-1}$ is fixed. We incrementally construct π_i by first fixing $\pi_i^{-1}(s)$, then $\pi_i^{-1}(s+1)$, and so on (we choose each value of H_i in turn). At each step the partially defined permutation π_i of the hull H_i extends the fixed prefix of the resulting configuration $c\pi_0\pi_1 \ldots \pi_i$. For instance, fixing the value $\pi_i^{-1}(s)$ defines the configuration $c\pi_0\pi_1 \ldots \pi_i$ on all normalized triples pqs.

- If the fixed prefix becomes lexicographically greater than the prefix of c of the same length, we can backtrack and try another value for the current point.
- On the other hand, if the fixed prefix becomes lexicographically smaller, then we already know that c is not canonical, and we do not have to further extend π_i.
- Only if the prefix stays equal to the corresponding prefix of c, we further extend the permutation π_i.

A complete permutation π_i will be reached only in the case of an i-automorphism.

The overall algorithm is shown in Algorithm 4. The case when the structure of c is $[h_0]$ is considered first (in that case only the all-zero configuration is canonical, and its automorphisms are cyclic permutations). Otherwise, we use $(m-1)$-automorphisms inherited from the parent configuration. For each $(m-1)$-automorphism π, we check whether it can be extended by a permutation π_m of H_m so that $c\pi\pi_m$ is lexicographically smaller than c. This is done by the auxiliary procedure `search_smaller_configuration`(). If it turns out that there is no such permutation, this auxiliary procedure returns *false*, and we proceed with the next $(m-1)$-automorphism. In that case, the auxiliary procedure also records m-automorphisms of c that extends π, if any.

The procedure `search_smaller_configuration`() is presented in more details in Algorithm 5. It is a recursive procedure that incrementally builds the permutation π_m of H_m, as previously explained. Its first argument π is an $(m-1)$-automorphism of the configuration c. The recursion goes on s, which is the next point for which the value $\pi^{-1}(s)$ should be fixed. The S is the set of the remaining values that can be assigned to $\pi^{-1}(s)$ (initially H_m). The procedure first looks for the values $r \in S$ that, when assigned to $\pi^{-1}(s)$, makes the prefix of $c\pi$ lexicographically smaller (which means that c is not canonical). In the same loop, the procedure records the values $r \in S$ that keep the prefix equal (the *equals* list in Algorithm 5). If we did not find an extended permutation π' that makes the prefix smaller, we try to further extend the permutations from *equals* in a recursive fashion. When a complete permutation π' is reached (i.e. when $s = n-1$), if it is in *equals*, it is stored as an m-automorphism of c.

3.3. Augmenting configurations using SAT solvers

Let L_{n-1} be the list of all canonical configurations of size $n-1$. The Faradžev-Read algorithm augments the elements of L_{n-1} in turn, and then

Require: c is a regular configuration of size n with hull structure $[h_0, \ldots, h_m]$
Ensure: the procedure returns *true* iff c is canonical
  begin
  if $m = 0$ **then**
    {c consists of H_0 only}
    if c is all-zero configuration **then**
      $m\_automorhpisms(c) = cyclic\_permutations(n)$
      return *true*
    else
      return *false*
  {$(m-1)$-automorphisms of c are inherited from the parent configuration c'}
  for all $(m-1)$-automorphism π of c **do**
    $s = h_0 + \ldots + h_{m-1}$
    if `search_smaller_configuration`(π, s, H_m, c) **then**
      return *false*
  return *true*
  end

Algorithm 4: `is_canonical`(c)

the augmented objects are checked for canonicity. In our case, the augmenting operation will be based on the parent-child relationship between the configurations – each canonical configuration from L_{n-1} is augmented to the list of all its regular children. Thanks to Theorem 3.7, each canonical configuration is a child of a canonical parent, so Property 1 needed in a general Faradžev-Read algorithm scheme is satisfied. Moreover, each configuration has exactly one parent, which allows us to use the simplified version of Faradžev-Read scheme (Algorithm 2). This means that we do not have to define the order on configurations, nor we have to prove other two properties mentioned in Section 2.1.

Configurations are essentially Boolean lists, constrained by CC system axioms that are Boolean constraints. Therefore, finding all augmenting configurations is essentially a Boolean constraint satisfaction problem. The augmenting operation will be implemented by utilizing an All-SAT solver. Namely, if we assign a propositional atom A_{pqr} to each normalized triple pqr, then each configuration c can be identified with the valuation v_c such that A_{pqr} is true in v_c if and only if $c(pqr) = 1$. Now it is sufficient to encode the CC axioms and other necessary constraints as propositional clauses, and the satisfying valuations of such CNF formula will correspond to the augmented configurations.

Assume that c is a canonical configuration of size $n-1$ with the hull structure $[h_0, h_1, \ldots, h_m]$ that should be augmented. As said before, its regular children may have one of the following two hull structures: $[h_0, h_1, \ldots, h_m + 1]$ (the point $n-1$ is added to the innermost hull) and $[h_0, h_1, \ldots, h_m, 1]$ (the point is added as a new singleton hull inside the innermost hull). The second case is permitted only if $h_m \geq 3$. Each of these two cases is considered in turn, and the union of the obtained sets of children is return as the result. In both cases, the CNF formula must encode the CC axioms (since the resulting configurations must be feasible), the constraints that fix c as the parent configuration, and the constraints that fix the structure of the child configurations.

Require: π is a partial permutation of the set $P_n = \{0, 1, \ldots, n-1\}$ such that $\pi(p)$ is defined for all $p \in H_k$, $k < m$, and $\pi^{-1}(q)$ is defined for all $q < s$
Require: s is the next point for which $\pi^{-1}(s)$ should be fixed
Require: S is the set of available values from H_m that may be assigned to $\pi^{-1}(s)$
Require: c is a regular configuration
Ensure: the procedure returns *true* iff the fixed prefix of $c\pi$ becomes lexicographically smaller than the prefix of c of the same length
begin
{The list *equals* will contain the extended permutations π' that keep the fixed prefix of $c\pi'$ equal to the corresponding prefix of c}
$equals = [\,]$
for all $r \in S$ **do**
{Extending π by fixing the value that maps to s}
$\pi' = \pi$
$\pi'^{-1}(s) = r$
{$<_{lex}$ compares only the relevant prefixes of $c\pi'$ and c}
if $c\pi' <_{lex} c$ **then**
return *true* {found smaller configuration}
else if $c <_{lex} c\pi'$ **then**
{do nothing; permutations π' that make the prefix lexicographically greater are skipped}
else
$equals.push((\pi', r))$ {Store permutations π' that keep the prefix equal}
if $s + 1 = n$ **then**
{Recursion exit: the complete permutation π' is reached}
if *equals* not empty **then**
{if *equals* is not empty, it contains π' – an m-automorphism of c}
$m\_automorhpisms(c).append(\pi')$
return *false*
{If the permutation is not complete, we try to extend it recursively}
for all $(\pi', r) \in equals$ **do**
if `search_smaller_configuration`$(\pi', s+1, S \setminus \{r\}, c)$ **then**
return *true*
return *false*
end

Algorithm 5: `search_smaller_configuration`(π, s, S, c)

Parent configuration encoding. In order to fix c as the parent configuration, for each $pqr \in \mathcal{N}_{n-1}$, we have the unit clause $\{A_{pqr}\}$ if $c(pqr) = 1$, and the unit clause $\{\neg A_{pqr}\}$, otherwise. We denote this set of clauses as F_{parent}.

Axiom Ax4 encoding. Only the axioms Ax4 and Ax5 should be encoded, since the axioms Ax0-Ax3 are implicitly integrated into the definition of the configuration. When the axiom Ax4 is concerned, recall that this axiom claims that for each quadruple of distinct points p, q, r, t it must hold that $ccw(pqt) \wedge ccw(qrt) \wedge ccw(rpt) \Rightarrow ccw(pqr)$. Notice that for each set of four distinct points we actually have 4! quadruples, depending on the permutation of points, and as many axiom instances. Fortunately, we do not have to en-

code all of them. Namely, according to Knuth [4], assuming $p < q < r < t$, it is sufficient to consider only two cases: $ccw(pqt) \wedge ccw(qrt) \wedge ccw(rpt) \Rightarrow ccw(pqr)$ and $ccw(prt) \wedge ccw(rqt) \wedge ccw(qpt) \Rightarrow ccw(prq)$. Translated to clauses (and normalized triples), we obtain $\{\neg A_{pqt}, \neg A_{qrt}, A_{prt}, A_{pqr}\}$ and $\{\neg A_{prt}, A_{qrt}, A_{pqt}, \neg A_{pqr}\}$. It is also sufficient to consider only quadruples that contain the point $n-1$, since all quadruples not containing $n-1$ are part of the parent configuration c and, therefore, already satisfy the axiom Ax4. Since we assumed $p < q < r < t$, this means that we can fix $t = n-1$. We denote this set of clauses as F_{ax4}.

Axiom Ax5 encoding. Recall that the axiom Ax5 claims that for each quintuple of distinct points p, q, r, t, s it must hold that $ccw(tsp) \wedge ccw(tsq) \wedge ccw(tsr) \wedge ccw(tpq) \wedge ccw(tqr) \Rightarrow ccw(tpr)$. Again, according to Knuth [4], instead of considering all 5! permutations for each set of five points, it is sufficient to consider only quintuples such that $p < q < r$, and for each such quintuple it is sufficient to consider only two axiom instances: $ccw(tsp) \wedge ccw(tsq) \wedge ccw(tsr) \wedge ccw(tpq) \wedge ccw(tqr) \Rightarrow ccw(tpr)$, and $ccw(tsp) \wedge ccw(tsq) \wedge ccw(tsr) \wedge ccw(tpq) \wedge ccw(trq) \Rightarrow ccw(trp)$. The clauses obtained depend on the normalization of triples. For instance, if $p < q < r < t < s$, we will have the clauses: $\{\neg A_{pts}, \neg A_{qts}, \neg A_{rts}, \neg A_{pqt}, \neg A_{qrt}, A_{prt}\}$, and $\{\neg A_{pts}, \neg A_{qts}, \neg A_{rts}, \neg A_{pqt}, A_{qrt}, \neg A_{prt}\}$. Other cases are similar. As before, it is sufficient to consider only quintuples that contain the point $n-1$. Since $p < q < r$, it must hold $r = n-1$, or $s = n-1$, or $t = n-1$. We denote this set of clauses as F_{ax5}.

Child hull structure encoding. In hulls of the child have the structure $[h_0, h_1, \ldots, h_m + 1]$, we must first encode that the point $n-1$ is inside the hulls $H_0, H_1, \ldots, H_{m-1}$. For each hull H_i $(i < m)$, and each two consecutive points $p, q \in H_i$, we have the unit clause $\{A_{pq(n-1)}\}$, if $p < q$, or $\{\neg A_{qp(n-1)}\}$, if $q < p$. Then, we must encode that the point $n-1$ is added to the innermost hull H_m of the parent configuration. That is, there must be exactly one pair of consecutive points $p, q \in H_m$ such that $\neg ccw(pq(n-1))$. If $H_m = [p_0, p_1, \ldots, p_k]$, this fact can be encoded with the clause $\{\neg ccw(p_0p_1(n-1)), \neg ccw(p_1p_2(n-1)), \ldots, \neg ccw(p_kp_0(n-1))\}$ (at least one negative literal is true), and with the set of binary clauses of form $\{ccw(p_ip_{i+1}(n-1)), ccw(p_jp_{j+1}(n-1))\}$ (at most one of the negative literals is true). The exact encoding of the clauses depends on the normalization of triples. For instance, if $p_0 < p_1 < \ldots < p_k$, then we have the clause $\{\neg A_{p_0p_1(n-1)}, \neg A_{p_1p_2(n-1)}, \ldots, \neg A_{p_{k-1}p_k(n-1)}, A_{p_0p_k(n-1)}\}$, the set of clauses $\{A_{p_ip_{i+1}(n-1)}, A_{p_jp_{j+1}(n-1)}\}$, where $0 \leq i < j < k$, and the set of clauses $\{\neg A_{p_0p_k(n-1)}, A_{p_jp_{j+1}(n-1)}\}$, where $0 \leq j < k$. Other cases are considered similarly. We denote the described set of clauses as F_{inner}.

In case of the structure $[h_0, h_1, \ldots, h_m, 1]$, we must encode that the point $n-1$ is inside the hulls $H_0, H_1, \ldots, H_m$. This is done by unit clauses, similarly as before. We denote the described set of clauses as F_{inside}.

The augmenting operation is given in Algorithm 6. It is assumed that we are equipped with an All-SAT solver that has the method $solve(F)$ which returns the set of all valuations satisfying the set of clauses F. The function $decode(V)$ construct the list of child configurations that correspond to the obtained satisfying valuations V. This list is returned as the list of augmented configurations.

Require: c is a canonical configuration of size $n-1$ of the structure $[h_0, h_1, \ldots, h_m]$
Ensure: the returned list L contains all regular children of c
begin
{ First we enumerate the children of the structure $[h_0, h_1, \ldots, h_m + 1]$ }
$F = F_{parent} \cup F_{ax4} \cup F_{ax5} \cup F_{inner}$
$V = solver.solve(F)$ { V is the set of satisfying valuations of F }
$L = decode(V)$ { function $decode()$ maps the valuations from V to the corresponding configurations }
{ Then, we enumerate the children of the structure $[h_0, h_1, \ldots, h_m, 1]$, if $h_m \geq 3$ }
if $h_m < 3$ **then**
return L
$F = F_{parent} \cup F_{ax4} \cup F_{ax5} \cup F_{inside}$
$V = solver.solve(F)$
$L = L \cup decode(V)$
return L
end

Algorithm 6: `augment`(c)

4. Implementation and results

The algorithm described in previous section is implemented in `C++` programming language.[4] With the default settings, the implementation just prints the number of non-isomorphic configurations of a given size, but the appropriate compilation options may be given in order to print all canonical configurations (as strings) to the standard output. In this section we provide details about the implementation and the obtained results.

4.1. SAT solvers used

By default, the implementation uses a simple, DPLL-based SAT solver implemented from scratch as a part of this project. Just like modern CDCL SAT solvers [20], out solver is implemented in an iterative fashion, using the stack of literals that supports backtracking (unlike the classical DPLL [21] which is based on recursion). The solver also implements two-watched-literals scheme [22]. All-SAT capability is implemented by applying backtrack explicitly whenever a complete satisfying valuation is constructed (except when the current decision level is zero, which means that we just have enumerated the last satisfying valuation). The solver also implements the clause database simplification, which is crucial for efficiency, since the problems we are solving include a great number of unit clauses.

Unlike state-of-the-art solvers, our solver does not include non-chronological backtracking, conflict analysis, clause learning, restarting, etc. We believe that such algorithmic improvements would not be much useful in our setting, since the SAT problems we are solving are relatively easy, compared to the hard industrial SAT instances, where such techniques are indispensable. Such observations

[4]The implementation is available at: `https://github.com/milanbankovic/convex`

are based on the relatively small number of conflicts and decisions needed to complete the solving (e.g. for $n = 10$, the average number of conflicts and the average number of decisions per invocation of the solver were both 1.01). On the other hand, since the solver is invoked once for each canonical configuration that should be augmented, we actually have to solve a quite large number of relatively easy All-SAT problems. For this reason, the most important property is that the solver must be *lightweight*, meaning that its data structures can be efficiently created and reinitialized when needed. Such reinitialization must take into account the fact that the problems we are solving share a great portion of the clause database (such as Ax4 and Ax5 clauses), which should not be initialized from scratch each time the solver is invoked. Instead, we initialize such clauses only once, when the solver is first created. During the simplification, the clauses that become true at the zero decision level due to the unit clauses are *deactivated* (i.e. transferred from the watch lists to the *inactive lists*), but not destroyed. When the solver is invoked again for the next problem, we just reactivate those clauses that need to be reactivated (based on the new set of unit clauses that are present in the next CNF formula).

We also experimented with the state-of-the-art CDCL-based SAT solver `picosat` [24]. It is an incremental solver, meaning that clause sets can be inserted and removed efficiently. It also supports All-SAT, by addition of the blocking clause. These two properties were promising enough to try it. Unfortunately, it turned out that its usage slowed down the implementation significantly. We believe that the reason is the ,,heaviness" of a CDCL-based solver such as `picosat`: it contains very complex algorithms and data structures that are well suited in solving individual, very hard SAT problems, but it is not meant to be invoked million of times to solve relatively easy SAT problems.

Another third-party solver we tried is the BDD-based All-SAT solver described in [23]. This solver was also promising due to its advanced All-SAT capabilities, but the results were similarly disappointing as with `picosat`. The reasons seem to be the same.

4.2. The easy cases ($n \leq 10$)

For $n \leq 10$, we confirmed the previously published results ([4, 7]) quite easily with our implementation, and the running times are shown in Table 1. The results are obtained on the computer equipped with 48 instances of AMD Opteron 6168 processor (1.9GHz), and 96Gb of RAM, but running in a single processor, sequential mode. The results obtained by using our implementation of DPLL based All-SAT solver are compared to those obtained by using `picosat` and BDD-based All-SAT solver.

4.3. The case $n = 11$

The number of non-isomorphic CC systems of size 11 obtained by our algorithm is 4 686 329 954, and the running time was 153 990 seconds (or almost 42 hours), using our implementation of DPLL-based All-SAT solver. We did not try the `picosat` and BDD-based All-SAT variants, due to their proven inferior performance for smaller n.

n	Count	Time (DPLL)	Time (picosat)	Time (BDD)
3	1	0.006	0.006	0.006
4	2	0.006	0.006	0.016
5	3	0.006	0.006	0.018
6	20	0.007	0.009	0.023
7	242	0.019	0.066	0.071
8	6 405	0.337	1.033	0.639
9	316 835	7.203	61.247	20.480
10	28 627 261	707	11 247	1 859

Table 1: The results for $n \leq 10$. Times are given in seconds

4.4. Parallelization of the implementation and the case $n = 12$

Attacking the case $n = 12$ was a quite ambitious task, because of the great number of parent configurations of size 11 that should be augmented (more than $4.6 \cdot 10^9$). If BFS version was used, in case of $n = 12$, we would have to store more than 4.6 billion canonical configurations of size 11 in the memory (together with the associated hull structures and automorphisms), before we start enumeration of canonical configurations of size 12, which would certainly consume the most of the available memory (even on the machine with 96Gb of RAM). Therefore, we used the DFS version, whose memory consumption was negligible. The Algorithm 3 is initially called with x being the only canonical configuration of size 3, $m = 3$ and n is the required size of the configurations that are being enumerated. With the DFS based approach, the memory complexity problem is resolved, but the time complexity still makes the $n = 12$ case practically intractable (the expected running time for a sequential version of the algorithm, on a single processor is estimated to be more than a year). Fortunately, the algorithm may be efficiently parallelized, since different configurations of the same size may be processed independently. That is, in Algorithm 3, each recursive call of the function `faradzev_read_dfs`() may be invoked as an independent task and scheduled for the execution on one of the available threads from the thread pool. The main challenge here is an efficient load-balancing. In our implementation, we rely on the Intel's *Thread-Building-Blocks (TBB)*[5], which is a multi-threaded library that permits different ways of parallelization within the shared memory model. The **forall** loop from Algorithm 3 is implemented using the `parallel_reduce` TBB construct, which partitions the list $[y_1, \ldots, y_k]$ (taking into account the load-balancing requirements and the number of available working threads), processes each partition in parallel by different threads, and then assembles the final list L_n (the *reduction*, in TBB's terminology).

To test the parallel version of the algorithm, we first invoked it for $n = 11$, using all 48 available processors. The required time was around 78 minutes, which is more than 32 times faster than the sequential version. Notice that the speedup is not linear, which could be explained by the additional time needed to split the work, and to assemble the obtained partial results.

Finally, we invoked the parallel version of the algorithm for $n = 12$, again

[5] `http://www.threadingbuildingblocks.org`

using all the available processors. The required time was 19 days, 1 hour and 25 minutes. The obtained number of non-isomorphic CC systems of size 12 is 1 382 939 012 729. The results of parallel execution are summarized in Table 2.

n	Count	Time
11	4 686 329 954	78
12	1 382 939 012 729	27 445

Table 2: The results for parallel execution. Times are given in minutes

4.5. Enumeration of order types using our implementation

Our implementation can be easily adapted to enumerate abstract order types instead of CC systems. For this purpose, it is sufficient to additionally compare each augmented configuration c with the configurations obtained by permuting the corresponding *mirrored configuration* (i.e. the configuration obtained from c by inverting each of its bits). In order to employ the homomorphism principle in the same fashion as before, the concept of *mirrored automorphism* is introduced – a mirrored automorphism is a permutation that, when applied to the mirrored configuration of c, produces c itself. Such mirrored automorphisms are stored for each canonical configuration c, together with its own automorphisms, and then used for the canonicity testing.

Using our implementation (in sequential mode), the abstact order types of size 10 are enumerated in 380 seconds, which is about half the time needed for enumeration of CC systems of the same size. This was somewhat expected, since the number of order types is almost twice smaller (there are 14 320 182 order types of size 10). The same holds for $n = 11$, where the sequential implementation enumerated all 2 343 203 071 order types in 1310 minutes, or almost 22 hours. This is comparable to the result reported in [13], where the same enumeration is obtained in about 20 hours. On the other hand, the parallel implementation (running the same hardware as before) enumerated order types of size 11 in about 36 minutes, which is again about half the time needed for parallel enumeration of CC systems of the same size. This means that "order types / CC systems" time consumption ratio is about 0.5, for both sequential and parallel implementations. For this reason, we believe that the order types of size 12 could be obtained in about 10 days using our parallel implementation (and our hardware), although we did not performed such enumeration.

5. Comparison to other approaches

Our approach to enumeration of non-isomorphic CC systems is based on augmentation of CC systems of smaller size, and checking the obtained augmented CC systems for canonicity, in a Faradžev-Read's fashion. In its simplified form (Algorithm 2), Faradžev-Read's method is similar to other approaches used for exhaustive enumeration of combinatorial objects, and some of them have been already employed for enumeration of order types. In the following text, we present such existing approaches and compare them to our algorithm in more details.

Order type enumeration by Scheucher et al. The work of Scheucher et al. [13] is the most relevant work to compare with, since it has many similarities with our approach. For instance, enumeration of order types of size n is done by augmentation of already enumerated order types of $n-1$, just like in our approach. The augmentation is performed by adding a new extremal point to the order type being augmented in all possible ways. The obtained order type is first sorted around the newly added point, and as such is appended to the list if and only if its representation (in the form of *lambda matrix*) is lexicographically minimal among the equivalent representations (with respect to relabeling and mirroring). Notice that this approach, in some sense, also combines augmentation and checking for canonicity. This makes the approach quite similar to Faradžev-Read's method, although not explicitly mentioned by the authors.

The main difference between the work given in [13] and our approach is in the representation of order types (CC systems, respectively). In [13], the authors use standard representation of order types in the form of lambda matrix. The value of $\lambda(i,j)$ represents the number of points $k \notin \{i,j\}$ such that $ccw(ijk)$ holds. It is well-known fact [25] that the orientation of all triples can be reconstructed from the information given in the lambda matrix, which makes lambda matrices a compact way to represent order types (or CC systems). Since the lambda matrix of an order type depends on the labeling, the canonical representation is taken to be the lexicographically smallest lambda matrix (compared row-by-row). This corresponds to one of the *natural labelings*, that is, labelings in which one of the points in the convex hull of the point set is labeled as 0th point, and other points are then sorted around that point, in clockwise order. This means that, in order to check for canonicity, it is sufficient to compare the given natural labeling to other natural labelings of the same set of points. In our approach, we have used binary string representations (which we called configurations) encoding the triplets orientations. In order to limit the number of permutations to be checked during the canonicity test, we analyzed the structures of the nested convex hulls of the point sets and applied the homomorphism principle to such structures.

Another important difference is in the axiomatization that is used during the augmentation phase. If we assume that a natural labeling is used, then it can be proven [14] that we can also assume that the points are also ordered by ascending x-coordinates. In that case, the *signotope axioms* can be used: for each four points $i < j < k < l$ it holds that the orientation of the triples in the sequence ijk, ijl, ikl, jkl is changed at most once. Now the abstract order types of given size correspond to orientation mappings that satisfy the signotope axioms. In [13], the signotope axioms are enforced to all order types obtained by augmentation. In our approach, we have used Knuth's axioms [4] for the same purpose.

The final difference is that our algorithm uses All-SAT solver for augmentation, while in [13], the authors use their own recursive procedure that incrementally defines the orientation of triples containing the newly added point in all possible ways, such that signotope axioms remain satisfied. It is worth mentioning that the use of SAT solver in the context of signotope axioms is also possible, and is used, for instance, in [14].

Enumeration by Aichholzer et al. In [5, 12], authors enumerate abstract order types using quite different approach based on *wiring diagrams* [11]. Namely,

abstract order types of size n correspond to topologically different *pseudoline arrangements* – that is, arrangements of n simple curves, where each two curves cross exactly once. The order of these crossings determines the corresponding order type. A *wiring diagram* is a realization of a pseudoline arrangement in an Euclidean plane, consisting of lines that are mostly horizontal, except when they cross with other lines. The enumeration of abstract order types is now reduced to the enumeration of different wiring diagrams, satisfying the suitable set of constraints. For this purpose, the method given in [25] is used.

The main part of work in [5, 12] considers the realizability of abstract order types (which we do not consider in our work). In that context, a similar concept of augmentation is used. Namely, if an order type of size n is realizable, then all its sub-order types of size $n-1$ are also realizable. That means that, in order to find a realization of a given order type of size n, we may look for a realization of some of its sub-order types of size $n-1$ (within the database of previously enumerated realizations of order types of size $n-1$) and extend it with a new point. This approach usually works, but not always. In the remaining cases, the alternative methods were used to find realizations, or to prove non-realizability (simulated annealing in [5], method of Bokowski and Richter [26] in [12]).

Another interesting part of work given in [12] concerns enumeration of order types satisfying a given *subset property*. A subset property is any property that, if holds for an order type of size n, it also holds for some of its sub-order types of size $n-1$. In that case, any order type of size n satisfying the property can be obtained by augmenting some order type of size $n-1$, satisfying the same property. This enables the usage of some of the algorithm schemes for exhaustive enumeration based on augmentation (such as Faradžev-Read's algorithm scheme). In [12], the *reversed search method* by Akis and Fukuda [27] is used. In the reversed search method, there are two important concepts that depend on a particular problem being tackled. The first is *adjacency oracle* which is responsible for enumeration of the objects adjacent to the given object in the search space graph. In the context of order types enumeration, this oracle would enumerate all order types constructed by augmentation of a given order type that satisfy the given subset property. The second concept is *parent mapping* which is a function that assigns to each object its unique parent (i.e. choose one of its adjacent objects in a deterministic fashion). In our setting, since each order type may be obtained by augmentation of more than one distinct order types of smaller size, we must determine the one that is its parent among them, according to some suitable criteria (for instance, the one with lexicographically smallest lambda matrix). The enumeration of objects starts from some initial objects called *sinks* (in our context, these might be order types of some small size that satisfy the given subset property). The tree (or forest, in case of multiple sinks) is constructed in depth-first fashion, by enumerating adjacent objects of the current object x and adding these objects to the tree as children of x if and only if x is recognized as their parent.

Compared to Faradžev-Read's method, the reversed search method is more general, and it can be applied to enumeration problems where the notion of isomorphism is not defined, or not important (for instance, for enumeration of spanning trees in a given graph). At the first glance, Faradžev-Read's algorithm (in its depth-first variant) might be considered as an instance of the reversed search method, where adjacency is defined with the augmentation operation, filtered by the canonicity test. On the other hand, the parent of each object x of

size n will be the smallest canonical object of size $n-1$ (with respect to the defined total ordering) from which the object x can be obtained by augmentation. However, it is important to notice that, while such instance of reversed search algorithm would indeed simulate Faradžev-Read's algorithm scheme, there is one important difference. Namely, in Faradžev-Read's algorithm, it is not required to be able to construct the parent object efficiently (that is, to find the smallest of all potential parents). The decision on whether the object x should be added to the tree is not based on whether the object from which it is constructed by augmentation is its parent, but is based on comparison with the greatest object of the same size added so far to the tree. In other words, the reversed search algorithm has the concept of *parent*, while the Faradžev-Read's algorithm has the concept of *ordering*, which makes the two methods incompatible in general. However, since the *simplified* version of Faradžev-Read's algorithm (that we used for enumeration of CC systems) does not use ordering, it may be considered as an instance of reversed search algorithm scheme.

6. Conclusions and Further Work

We have presented an instance of Faradžev-Read's algorithm and used it to enumerate all non-isomorphic CC systems with up to 12 points (using a multiprocessor computer). Enumerating all non-isomorphic CC systems with 13 or more points currently seems out of reach with the hardware we have at our disposal.

There are two crucial components of every Faradžev-Read's algorithm instance: checking canonicity and augmenting objects.

By finding a suitable invariant – the nested convex hull to which a point belongs to, by analyzing only regular configurations and by incrementally building permutations we have managed to optimize the canonicity test so that the profiling for $n = 10$ shows that the time it takes is only around 14% of the overall running time. Since the number of automorphisms reduces as the number of points increases and hulls become more nested, we estimate that for $n = 11$, and $n = 12$ the time for checking canonicity is quite negligible.

Analyzing only regular configurations and fixing the structure of child convex hulls also significantly speeds up the augmentation procedure. Its efficiency turns up to be crucial, as our profiling shows that it consumes more than 75% of the overall runtime. Due to the fact that the orientation of triples is essentially a Boolean value, constrained by CC system axioms that are Boolean constraints, already in CNF form, augmentation is reduced to All-SAT problem. We have applied a custom-made, lightweight SAT solver that showed better performance compared to state-of-the-art All-SAT solvers (due to the fact that the current problem requires solving a very large number of similar, relatively easy SAT problems, and not just one very hard SAT problem). It remains to test if introducing non-chronological backtracking and lemma learning (but without compromising the solver's lightweightness) would speed up the enumeration. Mixing combinatorial enumeration algorithms based on augmentation (such as Faradžev-Read's scheme) with all-SAT solving seems to be a promissing direction, and we advocate that it might give good results in other domains.

Funding: This work was partially supported by the Serbian Ministry of Science [grant number 174021].

References

[1] P. Erdős, G. Szekeres, A combinatorial problem in geometry, Compositio Mathematica 2 (1935) 463–470.

[2] F. Marić, Fast Formal Proof of the Erdős–Szekeres Conjecture for Convex Polygons with at Most 6 points, Journal of Automated Reasoning 62 (3) (2019) 301–329.

[3] G. Szekeres, L. Peters, Computer solution to the 17-point Erdős-Szekeres problem, ANZIAM J. 48 (2) (2006) 151–164.

[4] D. E. Knuth, Axioms and hulls, Vol. 606, Springer, 1992.

[5] O. Aichholzer, F. Aurenhammer, H. Krasser, Enumerating order types for small point sets with applications, Order 19 (3) (2002) 265–281.

[6] J. Bokowski, J. Richter, B. Sturmfels, Nonrealizability proofs in computational geometry, Discrete & Computational Geometry 5 (4) (1990) 333–350.

[7] A. Beygelzimer, S. Radziszowski, On halving line arrangements, Discrete Mathematics 257 (2-3) (2002) 267–283.

[8] I. Faradzev, Constructive enumeration of combinatorial objects, Colloques Internat. CNRS (260) (1978) 131–135.

[9] R. C. Read, Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations, in: Annals of Discrete Mathematics, Vol. 2, Elsevier, 1978, pp. 107–120.

[10] G. Brinkmann, Isomorphism rejection in structure generation programs, in: Discrete Mathematical Chemistry, 1998.

[11] S. Felsner, J. E. Goodman, Pseudoline arrangements, Handbook of Discrete and Computational Geometry 3 (2017).

[12] O. Aichholzer, H. Krasser, Abstract order type extension and new results on the rectilinear crossing number, Computational geometry 36 (1) (2007) 2–15.

[13] M. Scheucher, H. Schrezenmaier, R. Steiner, A note on universal point sets for planar graphs, in: International Symposium on Graph Drawing and Network Visualization, Springer, 2019, pp. 350–362.

[14] M. Scheucher, Two disjoint 5-holes in point sets, Computational Geometry 91 (2020) 101670. doi:https://doi.org/10.1016/j.comgeo.2020.101670.
URL http://www.sciencedirect.com/science/article/pii/S092577212030064X

[15] M. Balko, P. Valtr, A sat attack on the erdős–szekeres conjecture, European Journal of Combinatorics 66 (2017) 13 – 23, selected papers of EuroComb15. doi:https://doi.org/10.1016/j.ejc.2017.06.010.
URL http://www.sciencedirect.com/science/article/pii/S0195669817300847

[16] F. Marić, Verifying Faradžev-Read Type Isomorph-Free Exhaustive Generation, in: N. Peltier, V. Sofronie-Stokkermans (Eds.), Automated Reasoning, Springer International Publishing, Cham, 2020, pp. 270–287.

[17] G. Brinkmann, R. Deklerck, Generation of union-closed sets and Moore families, Journal of Integer Sequences 21 (1) (2018) 9–18.

[18] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, Vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009.

[19] S. A. Cook, The complexity of theorem-proving procedures, in: Proceedings of the third annual ACM symposium on Theory of computing, ACM, 1971, pp. 151–158.

[20] J. Marques-Silva, I. Lynce, S. Malik, Conflict-Driven Clause Learning SAT Solvers, in: Handbook of Satisfiability, IOS Press, 2009, Ch. 4, pp. 131–155.

[21] M. Davis, G. Logemann, D. Loveland, A Machine Program for Theorem-Proving, Communications of the ACM 5 (7) (1962) 394–397. doi:10.1145/368273.368557.
URL http://doi.acm.org/10.1145/368273.368557

[22] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver, in: Annual ACM IEEE Design Automation Conference, ACM, 2001, pp. 530–535.

[23] T. Toda, T. Soh, Implementing efficient all solutions SAT solvers, Journal of Experimental Algorithmics (JEA) 21 (2016) 1–44.

[24] A. Biere, Picosat essentials, Journal on Satisfiability, Boolean Modeling and Computation 4 (2-4) (2008) 75–97.

[25] J. Goodman, R. Pollack, Multidimensional sorting, SIAM J. Comput. 12 (1983) 484–507.

[26] J. Bokowski, J. Richter, On the finding of final polynomials, European Journal of Combinatorics 11 (1) (1990) 21–34. doi:https://doi.org/10.1016/S0195-6698(13)80052-2.
URL https://www.sciencedirect.com/science/article/pii/S0195669813800522

[27] D. Avis, K. Fukuda, Reverse search for enumeration, Discrete applied mathematics 65 (1-3) (1996) 21–46.