BFS-Based Canonical Codes for Generating Graphs with Constraint Programming

Xiao Peng
LAAS-CNRS, Université de Toulouse, Toulouse, France

Christine Solnon
Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France

Abstract

We consider the problem of generating all graphs that satisfy some given additional constraints (on vertex degrees, or cycle lengths, for example). Most previous works have proposed to generate canonical codes associated with adjacency matrices. In this paper, we consider canonical codes based on Breadth First Search (BFS), and we show how to generate them with Constraint Programming (CP): we introduce a set of basic constraints that must be satisfied by all canonical codes, thus breaking many symmetries, and we introduce a global constraint to break other symmetries. We illustrate the interest of our approach on connected claw-free cubic graphs, and show that it outperforms state-of-the-art CP and SAT Modulo Theory (SMT) approaches.

2012 ACM Subject Classification Mathematics of computing → Graph enumeration; Computing methodologies → Artificial intelligence

Keywords and phrases Graph Generation, Automorphisms, Symmetry Breaking

Digital Object Identifier 10.4230/LIPIcs.CP.2025.32

1 Introduction

In combinatorics and graph theory, graph enumeration involves systematically searching for all graphs that satisfy some given properties (such as a specific number of edges, vertex degree constraints, or structural features like girth). This has numerous applications in fields like chemical informatics and drug discovery, where molecules are modeled as graphs [5, 19]. To prevent us from generating isomorphic graphs, *i.e.*, graphs that are equivalent up to a renaming of their vertices, we may use canonical codes, *i.e.*, words associated with graphs such that two graphs are isomorphic if and only if they have the same canonical code: instead of enumerating graphs, we enumerate canonical codes, thus ensuring that all generated graphs are non-isomorphic.

Canonical codes may be based on adjacency matrices, *i.e.*, 2-dimensional arrays M such that $M_{ij} = 1$ if (i, j) is an edge, and 0 otherwise. A code may be obtained from M by concatenating its rows (or columns), thus obtaining a binary word. Since different vertex permutations may produce different adjacency matrices and, therefore, different codes, the lexicographically smallest one is selected as the canonical code, thus uniquely identifying the graph. To avoid exhaustively considering all vertex permutations, symmetry-breaking constraints may be used. In [4], Codish et al. introduce symmetry-breaking constraints that enforce a non-descending row order in the adjacency matrix and ensure minimality under vertex permutations within specific partitioned sets.

The Nauty algorithm [13] generates a canonical labeling through an iterative partition refinement process, where vertices are colored based on their connectivity. Once a discrete partition is reached, it serves as the canonical label. Although there is no direct work that enumerates all canonical codes using Nauty's labeling as constraints, as far as we know, the concept of symmetry breaking via structural graph information has been applied by Codish et al. [3]. They encode partition refinement into constraints, and introduce additional constraints

31st International Conference on Principles and Practice of Constraint Programming (CP 2025).
Editor: Maria Garcia de la Banda; Article No. 32; pp. 32:1–32:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to maintain minimality in the adjacency matrix while preserving partition structure. This approach provides insights into constructing compact and complete symmetry-breaking constraints, effectively reducing the need to evaluate all vertex permutations. Further studies on complete symmetry breaking are explored in [8, 9].

In addition to the adjacency matrix, Codish et al. demonstrate in [10] how various higher-dimensional graph invariants, such as vertex degrees and the cardinality of common neighbors, can be leveraged to define symmetry-breaking constraints. They compare different combinations of these invariants to evaluate their effectiveness in reducing the search space.

Beyond *static symmetry breaking*, which relies on a predefined set of permutations, Kirchweger and Szeider introduce in [11] a novel SMT-based approach for graph generation using *dynamic symmetry breaking*. The idea is to detect symmetries in the partially generated graph to enforce adjacency matrix minimality during the solving process, thus significantly improving efficiency compared to static symmetry-breaking methods.

Another class of canonical codes relies on graph traversals, where codes are sequences of traversed edges. Graph mining algorithms, such as Gaston [15] and gSpan [20], are usually based on Depth-First-Search (DFS). In [18] and [16], canonical codes based on Breadth-First-Search (BFS) are used for particular classes of graphs (Deterministic Finite Automata in [18] and hexagonal graphs in [16]) for which graph isomorphism may be solved in polynomial-time. In [14], some BFS-based symmetry-breaking predicates are introduced for generating connected graphs.

In this paper, we extend the work of [14] and introduce new symmetry breaking constraints for BFS-based canonical codes. Indeed, BFSs naturally construct spanning trees, and the orbits formed by the vertices within a spanning tree can be exploited to derive compact symmetry breaking constraints, that are efficiently handled by CP solvers. Also, every prefix of our canonical code is a canonical code, thus enabling the construction of dynamic symmetry-breaking constraints, similar to the approach in [11].

The paper is organized as follows. In Section 2, we introduce notations. In Section 3, we introduce BFS-based codes, and define a canonical BFS-based code as the smallest possible code. In Section 4, we describe a CP model for generating BFS-based codes. In Sections 5 and 6, we study properties of canonical codes. In Section 7, we introduce a global constraint that exploits these properties to ensure canonicity. In Section 8, we evaluate our method for generating connected claw-free cubic graphs, and compare it with the approach of [10], which relies on an adjacency matrix representation, as well as with the SMT-based approach of [11] and with Nauty [13]. We also introduce a more compact graph representation for this benchmark, thus enabling us to solve larger instances.

2 Notations and definitions

We note $[i, j]$ the set of all integers ranging from i to j, $\#S$ the cardinality of a set S, and $\preceq_{lex}$ the lexicographic order for comparing sequences of integer values.

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijection $f : V_1 \to V_2$ that preserves edges, *i.e.*, $\forall u, v \in V_1, (u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$. This problem belongs to $\mathcal{NP}$, but it is not known to be in $\mathcal{P}$ nor to be $\mathcal{NP}$-complete, and it is conjectured to be $\mathcal{NP}$-intermediate.

Throughout this paper, we consider a connected graph $G = (V, E)$ such that $\#V = n + 1$ and $\#E = n + m$, so that $0 \leq m \leq \frac{(n-2)(n-1)}{2}$. We note d_i the degree of a vertex $i \in V$.

Algorithm 1 BFS_G.

Input: A graph $G = (V, E)$

Output: A code associated with a BFS traversal of G

```
1  for each vertex v ∈ V do initialise num[v] to -1;
2  choose a first vertex v_0 ∈ V, set num[v_0] to 0 and add v_0 to an empty FIFO queue q;
3  initialize i to 0, and c_f and c_b to empty sequences;
4  while q is not empty do
5      pop u from q;
6      for each vertex v adjacent to u do
7          if num[v] < 0 then // Forward edge
8              add num[u] at the end of c_f, push v in q, set num[v] to i, and increment i;
9          else if num[u] < num[v] then // Backward edge
10             add num[u] and num[v] at the end of c_b;
11         end
12     end
13 end
14 return the concatenation of c_f and c_b
```

3 Canonical BFS-based code

A code is a sequence of $n + 2m$ integer values, generated through a BFS of G as described in Algorithm 1. We use a FIFO queue q to store the vertices that have been discovered but not yet treated, and we build two sequences c_f and c_b of n and $2m$ integer values, respectively. When a vertex v is pushed in q, we store in $num[v]$ its q-number, which ranges from 0 for the first pushed vertex to n for the last pushed one. At each iteration of the while loop (Lines 4-13), a vertex u is popped from q, and we consider each vertex v adjacent to u.

- If v is reached for the first time (Lines 7-8), $(num[u], num[v])$ is said to be a *forward edge*. $num[u]$ is the parent of $num[v]$, and it is added at the end of c_f. At the end of Algorithm 1, there are n forward edges which form a spanning tree of G.
- If v has already been reached (Lines 9-10), $(num[u], num[v])$ is said to be a *Backward edge*, and $num[u]$ and $num[v]$ are added at the end of c_b. To avoid treating twice a same edge, we add the condition that $num[u]$ must be smaller than $num[v]$.

The final code is obtained by concatenating c_f and c_b (Line 14). Hence, a code is a sequence of $n + 2m$ integer values $p_1, p_2, \ldots, p_n, x_1, y_1, x_2, y_2, \ldots x_m, y_m$ such that for each $i \in [1, n]$, (p_i, i) is a forward edge and for each $i \in [1, m]$, (x_i, y_i) is a backward edge. The first part of a code, from p_1 to p_n is called *forward code*, whereas the second part of the code, from x_1 to y_m is called *backward code*. When there is no ambiguity, commas between vertices in codes may be removed.

Different codes may be built for a same graph, depending on (i) which vertex is chosen first (Line 2) and (ii) the order considered to visit successors of u (Line 6). In order to make the algorithm deterministic, we add a parameter π which is a permutation of V used to break ties: Line 2, the chosen vertex v_0 is the first vertex of π; Line 6, successors of u are visited in the order defined by π. The code obtained when using π to break ties is denoted $\text{BFS}_G(\pi)$.

▶ **Example 1.** Let us consider the graph G displayed in Fig. 1 when the permutation π is *eabcd*, as displayed in Fig. 1(i). The first vertex pushed in q is e, and we have $num[e] = 0$. When e is popped from q, we iterate on its adjacent vertices in the order defined by π:

- when $v = a$, we set $num[a]$ to 1 and add $num[e] = 0$ to c_f, *i.e.*, $(0, 1)$ is a forward edge;
- when $v = b$, we set $num[b]$ to 2 and add $num[e] = 0$ to c_f, *i.e.*, $(0, 2)$ is a forward edge;

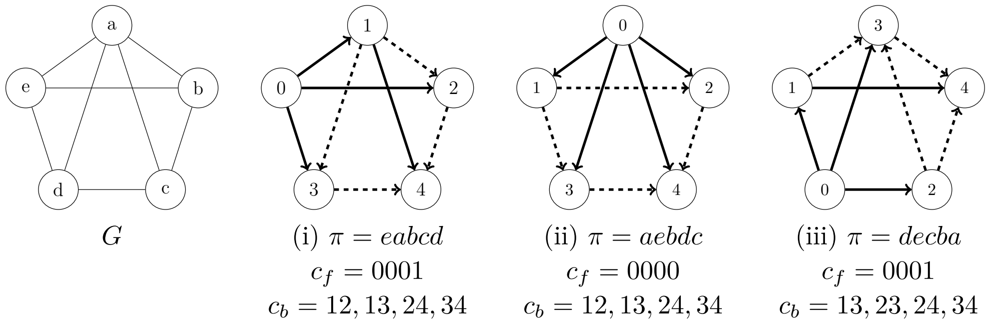


Figure 1 A graph G (on the left), and three BFS-based codes corresponding to three different permutations π. Vertices of the three graphs on the right are labeled with their q-numbers; forward-edges are displayed in solid lines; backward-edges are displayed in dotted lines.

- when $v = d$, we set $num[d]$ to 3 and add $num[e] = 0$ to c_f, *i.e.*, $(0, 3)$ is a forward edge.

When a is popped from q, we iterate on its adjacent vertices in the order defined by π:
- when $v = e$, we do nothing because edge (e, a) has already been treated;
- when $v = b$, we add $num[a] = 1$ and $num[b] = 2$ in c_b, *i.e.*, $(1, 2)$ is a backward edge;
- when $v = c$, we set $num[c]$ to 4 and add $num[a] = 1$ to c_f, *i.e.*, $(1, 4)$ is a forward edge;
- when $v = d$, we add $num[a] = 1$ and $num[d] = 3$ in c_b, *i.e.*, $(1, 3)$ is a backward edge.

When b is popped from q, we iterate on its adjacent vertices in the order defined by π:
- when $v = e$ or a, we do nothing because edges (e, b) and (a, b) have already been treated;
- when $v = c$, we add $num[b] = 2$ and $num[c] = 4$ in c_b, *i.e.*, $(2, 4)$ is a backward edge.

When d is popped from q, we iterate on its adjacent vertices in the order defined by π:
- when $v = e$ or a, we do nothing because edges (e, d) and (a, d) have already been treated;
- when $v = c$, we add $num[d] = 3$ and $num[c] = 4$ in c_b, *i.e.*, $(3, 4)$ is a backward edge.

When c is popped from q, we iterate on its adjacent vertices in the order defined by π, *i.e.*, a, b, and d, but we do nothing as edges (a, c), (b, c), and (d, c) have already been treated. Hence $\text{BFS}_G(eabcd) = 0001, 12, 13, 24, 34$.

Given a code, we can build the corresponding graph as the code allows us to reconstitute all its edges. For example, given $\text{BFS}_G(eabcd) = 0001, 12, 13, 24, 34$, we reconstitute the set of edges $\{(0, 1), (0, 2), (0, 3), (1, 4), (1, 2), (1, 3), (2, 4), (3, 4)\}$.

As shown in Example 1, there exist different possible codes for G, depending on the considered permutation π. We define a total order on the set of all possible codes that may be associated with G by considering a lexicographic order. Among all the possible codes for G, the smallest one according to this order is called the *canonical code* of G and it is unique.

▶ **Definition 2** (Canonical code $cc(G)$). *The canonical code of a graph $G = (V, E)$ is defined as $cc(G) = min\{BFS_G(\pi) : \pi \text{ is a permutation of } V\}$ when considering the lexicographic order $\preceq_{lex}$ to compare codes.*

In the example of Figure 1, the canonical code is $0000, 12, 13, 24, 34$.

An important property to allow an efficient enumeration of canonical codes is that any prefix of a canonical code is a canonical code: this property allows us to stop completing a code whenever its prefix is not canonical.

▶ **Theorem 3.** *Let $c = p_1p_2 \dots p_n, x_1y_1, \dots, x_my_m$ be a canonical code.*
- *For each $k \in [1, n]$, the prefix $c' = p_1 \dots p_k$ is a canonical code.*
- *For each $k \in [1, m]$, the prefix $c' = p_1 \dots p_n, x_1y_1, \dots, x_ky_k$ is a canonical code.*

Proof. Let G' be the subgraph of G that contains all edges defined by c'. Assume, for contradiction, that c' is not canonical, *i.e.*, there exists another lexicographically smaller code c'' that represents the same graph G'. We consider two cases.

Recall that our canonical code, derived from a BFS traversal, first lists the forward edges followed by the backward edges. In the first case, c' consists only of forward edges, implying that G' forms a tree. Since c'' is a valid traversal of G', it can be extended into a full traversal of G by discovering the remaining vertices, and results in a code smaller than c, which violates the canonicity of c. In the second case, c' includes backward edges. If the forward edge code $p_1 p_2 \dots p_n$ is already canonical, then the backward edges must follow a unique order, as dictated by lines 6-12 of Algorithm 1. A smaller c' would be extended to a full code smaller than c by enumerating the remaining backward edges.

Thus, in both cases, we reach a contradiction, proving that every prefix of a canonical code is also canonical. ◀

4 Basic CP model

We propose to generate codes with CP, thus allowing one to easily add other application-dependent constraints. Our CP model has the following integer variables:

- for each $i \in [1, n]$, p_i corresponds to the parent of vertex i in the spanning tree (in other words, (p_i, i) is a forward edge), and its domain is $[0, i-1]$ because the parent of i must have been discovered before i;
- for each $j \in [1, m]$, x_j and y_j correspond to the two endpoints of the jth backward edge and their domain is $[1, n]$ because an edge incident to 0 cannot be backward;
- for each $i \in [0, n]$, d_i corresponds to the degree of vertex i, and its domain is $[1, n-1]$.

The constraints are listed in Fig. 2. Constraints C1 to C7 are satisfied by any BFS-based code, even if it is not canonical:

- C1 is a consequence of the fact that the vertex whose q-number is 1 has been pushed in q just after 0 and, therefore, its parent can only be 0.
- C2 and C4 are consequences of the fact that, at each iteration of the while loop (Lines 4-13), the vertex u popped from q has a q-number increased by one (as vertices are pushed in q by increasing q-number).
- C3 is a consequence of the condition Line 9.
- C5 expresses the fact that, for each backward edge (x_i, y_i), the parent of y_i has been discovered before x_i. Indeed, let us suppose that this is not the case, *i.e.*, $p_{y_i} > x_i$. In this case, the parent of y_i would necessarily be x_i because y_i would not yet have been discovered when exploring the vertices adjacent to x_i (Lines 7-8).
- C6 and C7 relate degrees with edge variables.

C1: $p_1 = 0$
C2: $\forall i \in [2, n], p_{i-1} \leq p_i$
C3: $\forall i \in [1, m], x_i < y_i$
C4: $\forall i \in [1, m-1], x_i \leq x_{i+1}$
C5: $\forall i \in [1, m], p_{y_i} < x_i$
C6: $d_0 = \#\{i \in [1, n] | p_i = 0\}$
C7: $\forall i \in [1, n], d_i = 1 + \#\{j \in [1, n] | p_j = i\} + \#\{j \in [1, n] | x_j = i\} + \#\{j \in [1, n] | y_j = i\}$
C8: $\forall i \in [1, n], d_0 \geq d_i$
C9: $\forall i \in [1, m-1], x_i < x_{i+1} \vee (x_i = x_{i+1} \wedge y_i < y_{i+1})$
C10: $\forall i \in [1, n-1], p_i = p_{i+1} \Rightarrow \#\{j \in [1, n] | p_j = i\} \geq \#\{j \in [1, n] | p_j = i+1\}$

Figure 2 Basic CP model for generating BFS-based codes.

Constraints C8 to C10 prevent us from generating some non-canonical codes, *i.e.*, codes that are not the smallest possible ones:

- C8 comes from the fact that c starts with d_0 occurrences of 0: if there exists a vertex i such that $d_i > d_0$, then a smaller code is obtained by starting BFS from the vertex.
- If C9 is not satisfied, then a smaller code is obtained by exchanging $x_i y_i$ and $x_{i+1} y_{i+1}$.
- if C10 is not satisfied, then a smaller code is obtained by visiting $i+1$ before i.

We have experimentally compared this first CP model with the model introduced in [14] on a toy problem that aims at enumerating all graphs with k vertices and $2k-2$ edges. Both models have been implemented in Choco, and compute the same sets of solutions. However, our model is more efficient: when $k = 5$ (resp. 6, 7, and 8), it needs 0.02 (resp. 0.1, 0.4, and 4.3) seconds whereas the model of [14] needs 0.04 (resp. 0.2, 1.4, and 28.4) seconds. This comes from the fact that (i) we order sibling vertices with respect to the number of children (thanks to constraint C10) instead of sub-tree weights, and (ii) we explicitly model backward edges with x_j and y_j variables instead of using an adjacency boolean matrix.

Constraints C1 to C10 allow us to enumerate all canonical BFS-based codes, but some non-canonical codes may also be enumerated. Hence, in the next two sections we introduce two additional properties that must be satisfied by canonical codes, and that are used to propagate a global constraint that ensures the canonicity of BFS-based codes, as described in Section 7.

5 Breaking Symmetries of Forward Codes with Vertex Labels

In this section, we introduce a property which is used in Section 7 to ensure the canonicity of prefixes of forward codes. Forward code prefixes correspond to trees, and the tree isomorphism problem may be solved in polynomial time by computing vertex labels [1]. We exploit similar labels but, unlike the original algorithm of [1], we do not rename labels with integer values at each level of the tree, but define the label of a vertex i as a sequence that contains an integer value for each vertex in the subtree rooted in i. This allows us to compute labels in a more incremental way during the search.

▶ **Definition 4** (Vertex label). *Let $c_f = p_1 p_2 \dots p_k$ with $k \leq n$ be the prefix of a forward code. Let T be the tree associated with this prefix. For every vertex $i \in [0, k]$, let T_i be the subtree of T rooted at i, and Γ_i be the sequence of all vertices in T_i ordered by increasing value. The label of i, denoted l_i, is the sequence obtained by replacing every vertex j in Γ_i by $\#Ch_{c_f}(j)$ where $\#Ch_{c_f}(j)$ is the number of children of j in the tree defined by the forward code c_f.*

For example, let us consider the tree T displayed on the left of Fig. 3. The forward code associated with this tree is $c_f = 000113$. When $i = 1$, we have $\Gamma_1 = 145$ and $l_1 = 200$ because $\#Ch_{c_f}(1) = 2$ and $\#Ch_{c_f}(4) = \#Ch_{c_f}(5) = 0$. When $i = 0$, we have $\Gamma_0 = 0123456$ and $l_0 = 3201000$ because $\#Ch_{c_f}(0) = 3$, $\#Ch_{c_f}(1) = 2$, $\#Ch_{c_f}(3) = 1$, and $\#Ch_{c_f}(2) = \#Ch_{c_f}(4) = \#Ch_{c_f}(5) = \#Ch_{c_f}(6) = 0$.

Given the label l_0 of the root of a tree, we can reconstitute this tree as each value at position i in l_0 gives the number of children of vertex i. Hence, there is a bijection between each forward code and the label of the root of its associated tree. For example, we display in Fig. 3 two isomorphic trees T and T'. T corresponds to the forward code 000113 and the label of its root is 3201000, whereas T' corresponds to the forward code 000112 and the label of its root is 3210000.

An interesting property of vertex labels is that the children of a vertex in the tree associated with a canonical forward code prefix have non-decreasing labels, as stated below.

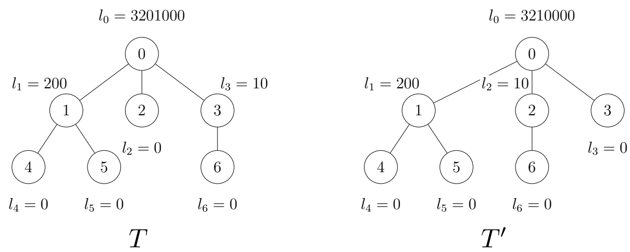


Figure 3 Vertex labels of two isomorphic trees T and T'.

▶ **Property 1.** *Let $c_f = p_1 p_2 \dots p_k$ with $k \leq n$ be the prefix of a canonical forward code. We have: $\forall i \in [1, k-1], p_i = p_{i+1} \Rightarrow l_i \succeq_{lex} l_{i+1}$.*

Proof. Let T_i and T_{i+1} be the subtrees rooted at i and $i+1$, respectively. The level of a vertex in T_i (resp. T_{i+1}) is its distance to the root i (resp. $i+1$). Let $\Gamma_i = q_1 q_2 \dots q_{\#T_i}$ be the ordered sequence of vertices in T_i, and $\Gamma_{i+1} = q'_1 q'_2 \dots q'_{\#T_{i+1}}$ be the ordered sequence of vertices in T_{i+1}. Since $p_i = p_{i+1}$, i and $i+1$ are sibling. Therefore, vertices at level k in T_i are discovered earlier than those at level k in T_{i+1}. Suppose $l_i \prec_{lex} l_{i+1}$ by contradiction, *i.e.*, $\#Ch_{c_f}(q_1) \dots \#Ch_{c_f}(q_{\#T_i}) \prec_{lex} \#Ch_{c_f}(q'_1) \dots \#Ch_{c_f}(q'_{\#T_{i+1}})$. This can occur either when (i) l_i is a prefix of l_{i+1} or (ii) it exists $c \in [1, \#T_i]$ such that $\forall j \in [1, c-1], \#Ch_{c_f}(q_j) = \#Ch_{c_f}(q'_j) \wedge \#Ch_{c_f}(q_c) < \#Ch_{c_f}(q'_c)$. In both cases, if we swap the order of i and $i+1$, then all vertices in Γ_{i+1} will be explored before those in Γ_i at the same level, resulting in a larger label l_0, which contradicts the canonicity of c_f. ◀

6 Breaking Symmetries of Backward Edges

Given a forward code prefix, we propose to exploit automorphisms in the tree associated with this prefix in order to break symmetries on backward edges. An automorphism of a tree $T = (V_T, E_T)$ is a permutation σ of V_T which is an isomorphism from T to T. Given a forward code prefix c_f, we only consider automorphisms that are still automorphisms on all possible extensions of c_f to full canonical forward codes, as defined below.

▶ **Definition 5** ($Aut(c_f)$)**.** *Let $c_f = p_1 \dots p_k$ with $k \leq n$ be a prefix of a canonical forward code. $Aut(c_f)$ is the set of all automorphisms $\sigma : [1, k] \rightarrow [1, k]$ that are still valid on all canonical extensions of c_f,* i.e., *for any sequence $p_{k+1} \dots p_n$ such that $p_1 \dots p_n$ is canonical, the automorphism σ' such that $\forall i \in [1, k], \sigma'(i) = \sigma(i)$ and $\forall i \in [k+1, n], \sigma'(i) = i$ is an automorphism of the tree associated with $p_1 \dots p_n$,* i.e., *$\sigma' \in Aut(p_1 \dots p_n)$.*

For example, let us consider the tree T' displayed in Fig. 3. The canonical code of T' is 000112. Let us assume that $n = 9$, *i.e.*, we still have to add three forward edges to the tree. Let σ_{45} be the automorphism that only exchanges vertices 4 and 5 and leaves unchanged all other vertices, *i.e.*, $\sigma_{45}(4) = 5, \sigma_{45}(5) = 4$, and $\forall i \notin \{4, 5\}, \sigma_{45}(i) = i$. σ_{45} does not belong to $Aut(000112)$. Indeed, there exist extensions of 000112 for which σ_{45} is not an automorphism. For example, if we add the forward edges $(3, 7)$ and $(4, 8)$ to T', leading to the forward code 00011234, then σ_{45} is no longer an automorphism because the subtree rooted in 4 is no longer isomorphic to the subtree rooted in 5.

Now, let us consider the tree obtained from T' by adding the forward edge $(6, 7)$. The associated forward code is 0001126. In this case, σ_{45} belongs to $Aut(0001126)$ because it is still an automorphism on all canonical extensions of 0001126. Indeed, 4 and 5 cannot be parent (because 6, which is at the same level as 4 and 5 but with a larger q-number, already has a child).

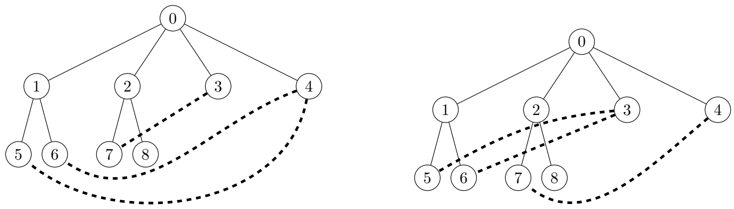


Figure 4 Example of two isomorphic graphs (backward edges are displayed with dotted lines). The code of the left-hand graph is $00001122, 37, 45, 46$ and it is not canonical because we obtain a smaller code (*i.e.*, $00001122, 35, 36, 47$) when exchanging 3 with 4, as displayed on the right.

The following property allows us to exploit automorphisms to break symmetries on backward edges.

▶ **Property 2.** *Let $c = p_1 \dots p_n, x_1y_1, \dots, x_my_m$ be a canonical code of G. The following property holds for any $k \in [1, n]$ and any $l \in [1, m]$ such that $\forall i \in [1, l], x_i, y_i \in [1, k]$:*
$\forall \sigma \in Aut(p_1 \dots p_k), x_1y_1 \dots x_ly_l \preceq_{lex} \sigma(x_1y_1 \dots x_ly_l)$
where $\sigma(x_1y_1 \dots x_ly_l)$ is obtained from $x_1y_1 \dots x_ly_l$ in two steps: (i) we compute the set of edges $S = \{(min\{\sigma(x_i), \sigma(y_i)\}, max\{\sigma(x_i), \sigma(y_i)\}) | i \in [1, l]\}$, and (ii) we sort the set S of edges in lexicographically ascending order and concatenate them into a sequence of $2l$ values.

Proof. By definition, when applying σ to the tree T associated with $p_1 \dots p_n$, we obtain a tree which is isomorphic to T and, therefore, has the same canonical code as T. For contradiction, let us assume that there exists $l \in [1, m]$ such that $x_1y_1 \dots x_ly_l \succ_{lex} \sigma(x_1y_1 \dots x_ly_l)$. In this case, the code obtained from c by replacing $x_1y_1 \dots x_ly_l$ with $\sigma(x_1y_1 \dots x_ly_l)$ is lexicographically smaller than c, which is in contradiction with the fact that c is canonical. ◀

For example, let us consider the graph G displayed in the left of Fig. 4. The permutation σ_{34} that exchanges 4 with 3 and leaves unchanged all other vertices belongs to $Aut(00001122)$. The code of G is $00001122, 37, 45, 46$. In Step 1, we compute the set S of edges obtained when applying the permutation σ_{34} to $\{(3, 7), (4, 5), (4, 6)\}$, *i.e.*, $S = \{(4, 7), (3, 5), (3, 6)\}$. Then, we sort edges of S in lexicographic order to obtain the sequence $35, 36, 47$ which is lexicographically smaller than $37, 45, 46$. Hence, $00001122, 37, 45, 46$ is not canonical.

7 Global *CanonicalCode* Constraint

Properties 1 and 2 are used to propagate the global constraint defined below.

▶ **Definition 6.** *Let $n \geq 1$ and $m \geq 0$ be integer values, and $\forall i \in [1, n], \forall j \in [1, m], p_i, x_j$, and y_j be integer variables. The global constraint $cc(p_1, \dots, p_n, x_1, y_1, \dots, x_m, y_m)$ is satisfied iff $p_1 \dots p_n x_1 y_1 \dots x_m y_m$ is a BFS-based canonical code.*

Propagation of Property 1

We exploit Property 1 to ensure that the forward code is canonical: When the domain of a variable p_k with $k \leq n$ is reduced to a singleton, if the domain of p_j is also reduced to a singleton for each $j \in [1, k-1]$, and if there exists $i \in [1, k-1]$ such that $p_i = p_{i+1}$ and $l_i \prec_{lex} l_{i+1}$, then a failure is raised. Note that this ensures that the forward code is the smallest possible one when starting the search from vertex 0. However, if there exists a vertex $i \in [1, k]$ such that $1 + \#Ch_{p_1 \dots p_k}(i) = \#Ch_{p_1 \dots p_k}(0)$, then it may be possible that a smaller forward code exists. Hence, to fully ensure that $p_1 \dots p_k$ is canonical, we must build every tree starting from a vertex $i \in [1,k]$ such that $1 + \#Ch_{p_1 \dots p_k}(i) = \#Ch_{p_1 \dots p_k}(0)$, build the associated smallest forward code, and check that it is not smaller than $p_1 \dots p_k$.

For an efficient propagation of Property 1, we maintain a 2 dimensional array t such that, $\forall i, j \in [1,k], t[i][j] \in \{-1, 0, 1\}$ depending on whether $l_i \prec_{lex} l_j$, $l_i = l_j$, or $l_i \succ_{lex} l_j$.

Propagation of Property 2

We exploit Property 2 to detect some cases where the backward code is not canonical: When the domain of a variable y_l is reduced to a singleton, if the domains of x_j and y_j are also reduced to singletons for each $j \in [1,l]$, a failure is raised if there exists $\sigma \in Aut(p_1 \dots p_k)$ such that $x_1 y_1 \dots x_l y_l \succ_{lex} \sigma(x_1 y_1 \dots x_l y_l)$ where $k \in [1,n]$ is the largest value such that the domain of p_i is reduced to a singleton for each $i \in [1,k]$. We use vertex labels to compute a partition of the vertices in orbits (vertices i and j are in a same orbit if $p_i = p_j$ and $l_i = l_j$), and we use this partition to compute automorphisms. However, to compute $Aut(p_1 \dots p_k)$ when $k < n$, we need to discard automorphisms that may not be valid after the addition of $n-k$ new forward edges (as stated in Def. 5). More precisely, if two vertices u and v belong to a same orbit in $p_1 p_2 \dots p_k$, and the largest vertex in their respective subtrees T_u and T_v is smaller than p_{k+1}, then this orbit is considered to compute $Aut(p_1 \dots p_k)$ because it is preserved in $p_1 \dots p_n$, as T_u and T_v cannot be further extended with forward edges.

Canonicity Check

Properties 1 and 2 are necessary conditions for canonicity, but they are not sufficient. For example, let us consider the tree T' displayed in Fig. 3, and let us assume we have added edges $(2,4), (3,5)$, and $(4,5)$ to T'. In this case, the code $000112, 24, 35, 45$ is not canonical, though its forward code is canonical and it satisfies Properties 1 and 2. Indeed, the canonical code is 000112,23,34,46, and it is obtained by starting the BFS from vertex 4 of T'.

Hence, we need to check if there exists another BFS that leads to a smaller code (in which case we raise a failure). More precisely, let $k \in [1,n]$ be the largest value such that the domain of p_i is reduced to a singleton for each $i \in [1,k]$, let $l \in [1,m]$ be the largest value such that the domains of both x_i and y_i are reduced to singletons for each $i \in [1,l]$, and let $G = (V,E)$ be the corresponding graph, *i.e.*, $V = [0,k]$ and $E = \{(p_i, i) | i \in [1,k]\} \cup \{(x_i, y_i) | i \in [1,l]\}$. If there exists a permutation π of $[0,k]$ such that $\text{BFS}_G(\pi) < p_1 \dots p_k x_1 y_1 \dots x_l y_l$, then we raise a failure. We exploit Constraint C8 to limit the set of permutations π to those that start with a vertex that has the same degree as 0 in G. We also exploit Constraints C9 and C10 as well as Properties 1 and 2 to break ties when choosing the next vertex v to visit (Line 6 of Algo 1). Finally, we exploit the property LexBFS introduced in [6] to avoid some BFSs (that cannot lead to canonical codes) by breaking ties when choosing the next neighbor v of u to visit (Line 6). More precisely, for each neighbor v of u, let $N_v = \{num[w] | (v,w) \in E \land num[w] < num[u]\}$ be the set of q-numbers of neighbors of v that have already been numbered, and let S_v be the sequence obtained by sorting elements of N_v by increasing value. At each iteration of the loop Lines 6-11, we choose the vertex v which has the smallest sequence S_v, where a sequence S_v is smaller than another sequence $S_{v'}$ if S_v is a prefix of $S_{v'}$ or if $S_v \prec_{lex} S_{v'}$ (see [6] for more details).

Table 1 Results for Problems P_1 to P_3, when considering only Constraints $C1$ to $C10$, or when combining Constraints $C1$ to $C10$ with the global constraint cc. k is the number of vertices, nb is the number of solutions, and t is the CPU time (in seconds) to enumerate these solutions. When time exceeds 200s, $t =' -'$, and nb gives the number of codes enumerated in 200s.

k	P_1				P_2				P_3			
	$C1$-$C10$		$C1$-$C10$+ cc		$C1$-$C10$		$C1$-$C10$ + cc		$C1$-$C10$		$C1$-$C10$ + cc	
	nb	t	nb	t	nb	t	nb	t	nb	t	nb	t
5	15	0.02	2	0.01	3	0.00	1	0.00	0	0.00	0	0.00
7	8,959	0.29	126	0.23	70	0.00	2	0.00	0	0.01	0	0.01
9	9,736,406	177.2	26,631	9.77	3,507	0.11	3	0.01	756,497	20.83	5,804	1.88
11	>9,740,145	-	>318,456	-	286,884	9.17	5	0.07	>8,593,667	-	>835,095	-

First Experiments on Toy Problems

We have implemented our global constraint and our CP model in Choco [17] in a straightforward way[1], using a global *count* constraint to implement Constraints C6, C7, and C10.

To evaluate the interest of our global constraint, we consider three toy problems, denoted P_1, P_2 and P_3. P_1 aims at enumerating all graphs with k vertices and $2k-2$ edges. We denote $G(k, 2k-2)$ this set of graphs. P_2 aims at enumerating all graphs of $G(k, 2k-2)$ which have one vertex of degree $k-1$ and $k-1$ vertices of degree three. P_3 aims at enumerating all graphs of $G(k, 2k-2)$ which have exactly two vertices of degree $\lceil k/2 \rceil$ while all other vertices have degrees strictly lower than $\lceil k/2 \rceil$. Constraints on vertex degrees are defined in a very straightforward way as our CP model already has variables associated with degrees.

In Table 1, we give experimental results obtained on an Intel Xeon E5-2623v3 of 3.0GHz with 32GB of RAM. For the three problems, the number of solutions when considering only Constraints C1 to C10 is much larger than the actual number of different graphs (given by column nb of $C1$-$C10$+cc). For example, when $k = 5$, $G(k, 2k-2)$ only contains 2 different graphs but we generate 15 different codes. The addition of the global constraint cc allows us to compute only canonical codes, and this strongly reduces both the number of generated solutions and the CPU time. When adding constraints on vertex degrees, the number of canonical codes and, therefore, the CPU time are strongly decreased, especially for P_2.

8 Application to the Generation of Connected Claw-Free Cubic Graphs

Cubic graphs are graphs in which each vertex has degree 3. A cubic graph is claw-free if it does not contain $K_{1,3}$ as an induced subgraph. This is equivalent to requiring that every vertex must participate in at least one triangle.

To enumerate all claw-free cubic graphs, we define a CP model composed of Constraint C1 to C10 combined with our global cc constraint. To enforce the cubic degree condition, we set the domain of each degree variable d_i to $\{3\}$, for each $i \in [0, n]$.

To ensure claw-freeness, we introduce a global constraint that ensures that each vertex is involved in at least one triangle. The propagator of this constraint maintains two sets of edges as proposed in [2]: *mandatory edges*, which must be in the solution, and *possible edges*, which may be included. We use sparse sets to efficiently maintain these sets [12].

[1] Our code is available at `https://github.com/godotshaw/bfscanonicalcode.git`

A dynamic variable selection strategy is employed to ensure that $dep_p - dep_x > 1$ is maintained throughout the search, where dep_p and dep_x are the depth levels of the last instantiated forward and backward edge variables, respectively. By prioritizing the instantiation of p-variables while interleaving backward edge variables, we avoid the enumeration of useless spanning trees. Additionally, for every $i \in [1, m]$, y_i is instantiated just after x_i.

As the procedure for checking the canonicity is rather expensive, we introduce a parameter f which allows us to control the frequency of canonicity checking: When $f = 1$, canonicity is checked after each edge assignment; when $f > 1$, it is checked every f edge assignment. Of course, when all variables are assigned, the canonicity check is performed, whatever the value of f is, in order to ensure that the global constraint is satisfied. A similar parameter is used in the SMT-based approach of [11].

Table 2 shows the results for $n \in [20, 44]$ by steps of 2, when our parameter f belongs to $\{1, 10, 20, 30\}$. When $f = 1$, run times are very often longer than when $f \geq 10$. A good tradeoff is reached when $f \in \{10, 20\}$.

In Table 2, we also display the results reported by Codish et al. in [10] (the code of this approach is not available). This approach does not break all symmetries and, therefore, it may compute redundant graphs that are isomorphic to previously enumerated graphs. For example, when $n = 32$, there are 731 different graphs whereas the approach of [10] computes 29,069 solutions. As a consequence, this approach does not scale well and cannot be used to solve larger instances within a reasonable amount of time. As a comparison, our method which only computes non-isomorphic graphs achieves a speed-up of more than 60 for $n \geq 30$ (note however that the two approaches have been run on different computers).

We have adapted the SMT-based approach described in [11] to generate claw-free cubic graphs, and we display the results obtained with this approach, on the same computer as the one used in our experiments. To handle the claw-free constraint, we directly utilize the solver's built-in feature, **"–forbidden-induced-subgraphs"**, to prohibit the predefined induced subgraph $K_{1,3}$. In [11], a parameter similar to f is used to control the frequency of **"minimality check"**, a non-polynomial operation that verifies whether the partially defined graph is canonical in their approach. We report results obtained with $f \in \{1, 10, 20, 30\}$ (the default value for this parameter is 20). The best results of SMT are obtained when $f = 10$, and our method consistently outperforms SMT across all instances.

Finally, we display results obtained with Nauty [13] (using `geng -F` to ensure claw-freeness). Our approach is always more efficient than Nauty, but it needs more memory (*e.g.*, 444584Kb instead of 22760Kb when $n = 30$).

Graph contraction

To show the versatility of our approach, we show how to exploit a property introduced in [7] to speed-up the generation of claw-free cubic graphs.

▶ **Proposition 7** (Claim A in [7]). *The vertex set V of a claw-free cubic graph G can be uniquely partitioned into k sets V_1, ..., V_k such that $\forall i \in [1, k]$, the subgraph of G induced by V_i is either a triangle, as displayed in Fig. 5(a) or a diamond as displayed in Fig. 5 (b).*

Hence, a claw-free cubic graph can be contracted by replacing specific patterns with labeled meta-vertices. Let us first define the basic contraction operation.

▶ **Definition 8** (Contraction). *Given a graph $G = (V, E)$, a pattern graph $P = (V_P, E_P)$, and a set $S \subseteq V$ such that the subgraph of G induced by S is isomorphic to P, the contraction of G with respect to P and S is the multigraph $G' = (V', E')$ such that the occurrence of P in G is replaced with a single meta-vertex u_S, i.e., $V' = V \setminus S \cup \{u_S\}$ and $E' = \{(u, v) \in E | \{u, v\} \subseteq V \setminus S\} \cup \{(u_S, v) | \exists u \in S, (u, v) \in E\}$. The meta-vertex u_S is called a P-meta-vertex.*

Table 2 Results for generating connected claw-free cubic graphs. Each line successively gives the number n of vertices, the number of different graphs, the results of [10] (number of solutions and time), and the time of our CP model and of the SMT approach of [11] (when the frequency of canonicity checking is in $\{1/1, 1/10, 1/20, 1/30\}$), as well as Nauty [13]. All times are in seconds and best times are highlighted in gray. We display '-' when time exceeds 3600 seconds.

n	graphs	Approach of [10]		Our CP model $C1$-$C10$+cc				SMT approach of [11]				Nauty
		sols	time	1/1	1/10	1/20	1/30	1/1	1/10	1/20	1/30	
20	15	132	2.1s	0.5	0.4	0.5	0.5	1.6	0.8	1.5	1.3	0.5
22	27	307	3.9s	1.0	0.7	0.8	0.9	3.3	1.9	2.5	3.5	2.3
24	54	660	11.0s	1.2	1.2	1.3	1.6	9.4	3.6	4.2	5.3	10.5
26	94	1,835	45.4s	2.2	1.6	2.0	2.5	19.9	5.3	6.5	7.7	48.6
28	181	4,372	2.57m	3.7	2.5	2.7	4.2	33.5	10.6	15.1	18.5	218.2
30	369	10,567	6.60m	6.4	4.9	4.3	8.0	63.2	20.8	21.5	29.0	3293.7
32	731	29,069	24.28m	11.0	8.9	7.5	14.4	138.4	33.2	46.4	44.7	-
34	1,502	-	-	25.0	18.5	16.0	20.9	313.6	70.2	87.6	82.4	-
36	3,187	-	-	59.7	32.9	44.4	37.3	512.9	135.5	177.7	190.6	-
38	6,914	-	-	144.7	92.8	128.1	77.6	935.6	249.6	328.7	348.9	-
40	15,025	-	-	421.9	232.2	295.7	206.8	1936	548.8	734.0	775.0	-
42	33,687	-	-	1195	456.2	589.9	594.0	4872	1420	1526	1511	-
44	77,450	-	-	3094	1133	1142	1461	-	2979	-	3584	-

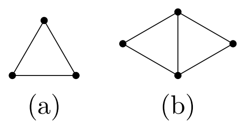


Figure 5 (a): Triangle. (b): Diamond.

E' is a multiset (and therefore G' is a multigraph) because there may exist a vertex $v \in V \setminus S$ such that several vertices of S are adjacent to v.

We consider a contracted graph obtained by contracting triangles and diamonds. In a cubic graph, two diamonds cannot share a same vertex. Also, two triangles that are not in a diamond cannot share a same vertex. To ensure a deterministic process such that the contracted graph is unique, we contract diamonds before triangles.

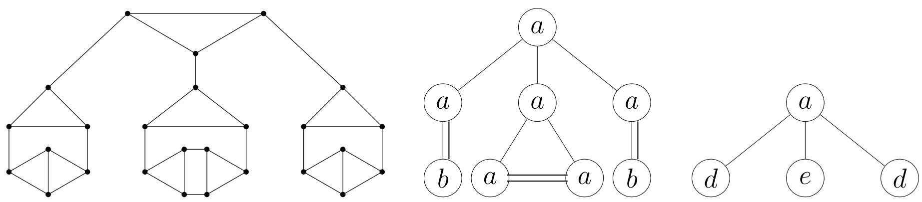
▶ **Definition 9** (Contracted graph G_{ab}). *Given a claw-free cubic graph G, the contracted graph G_{ab} is the graph obtained by (i) contracting every occurrence of the diamond pattern (b) in G into a b-meta-vertex, and (ii) contracting every occurrence of the triangle pattern (a) into an a-meta-vertex.*

We display in Fig. 6 an example of contracted graph G_{ab}.

Proposition 7 ensures us that G_{ab} only contains meta-vertices. As G is a cubic graph and the triangle has 3 vertices of degree 2, the degree of every a-meta-vertex is 3. Similarly, the degree of every b-meta-vertex in G_{ab} is 2 because the diamond has 2 vertices of degree 3.

▶ **Proposition 10.** *There are exactly 3 patterns that may create multi-edges in G_{ab}. These patterns, named (c), (d), and (e), are displayed in Fig. 7.*

Proof. A bridge in a connected graph is an edge such that the graph without this edge is no longer connected. There are only two possible claw-free cubic graphs that contain no bridge. These two graphs are displayed in Fig. 8 (first two graphs on the left) and they are treated as special cases. All other claw-free cubic graphs contain at least one bridge. Let G be one of these graphs and G' be the graph obtained from G by removing all bridges. G' is composed of k connected components such that each connected component is a claw-free graph that



(i) G (ii) G_{ab} (iii) G_c

Figure 6 Example of graph contraction. From the claw-free cubic graph G we obtain the multigraph G_{ab} by contracting the two diamond occurrences and then contracting the 6 triangle occurrences. From G_{ab}, we obtain G_C by contracting the occurrence of pattern (e), and the 2 occurrences of pattern (d).

does not contain bridges and that contains at least one vertex of degree 2. The multi-edges in G_{ab} are also multi-edges in G'_{ab}. Each connected component of G' is necessarily one of the three graphs displayed in Fig. 7. ◀

Hence, to remove all multi-edges, we contract G_{ab} as defined below, starting with occurrences of (e) because (c) is a subgraph of (e).

▶ **Definition 11** (Contracted graph G_C). *Given a contracted graph G_{ab}, the contracted graph G_C is obtained from G_{ab} by (i) contracting every occurrence of (e) into an e-meta-vertex, then (ii) contracting every occurrence of (c) or (d) into a c-meta-vertex or a d-meta-vertex.*

We display in Fig. 6 an example of contracted graph G_C. We can show that for any initial claw-free cubic graph G, the contracted graph G_c is either one of the 5 graphs displayed in Fig. 8, which are treated as special cases, or it is a simple connected graph that does not contain multi-edges and that contains 5 different kinds of meta-vertices: the degree of meta-vertices of type a (resp. b, c, d, and e) is 3 (resp. 2, 1, 1, and 2). The meta-vertices of G_C define a partition of the vertices of G: each meta-vertex of type a (resp. b, c, d, and e) corresponds to a set of 3 (resp. 4, 6, 7, and 9) vertices of V.

▶ **Proposition 12.** *Let G and G' be two claw-free cubic graphs, and let G_C and G'_C be their corresponding contracted graphs. Then G_C and G'_C are isomorphic if and only if G and G' are isomorphic.*

Proof. To prove the proposition, we establish that the contraction process from a claw-free cubic graph G to its contracted graph G_C is bijective. First, the contraction process following Definition 9 and Definition 11 is deterministic. Additionally, we label each vertex in G_C according to the subgraph pattern from which it was contracted, ensuring that G_C retains information about the structure of G. Therefore, the contracted graph G_C is uniquely determined by G. Conversely, given G_C, we can reconstruct G uniquely by replacing each meta-vertex in G_C with its corresponding subgraph pattern. Since each subgraph pattern is symmetric with respect to the vertices of degree 2, there is no ambiguity in reconnecting the subgraphs. The connections between these subgraphs follow directly from the adjacency structure of G_C, ensuring a unique reconstruction of G.

This uniqueness in both directions (from G to G_C and from G_C to G) implies that G and G_C are isomorphically equivalent. ◀

The enumeration of claw-free cubic graphs can now be simplified to the enumeration of contracted graphs G_C (except when $n \in \{4, 6, 8, 10, 12\}$ in which case we must add the special cases displayed in Fig. 8). These contracted graphs have labels associated with vertices. Hence, we introduce a new canonical code for labeled graphs.

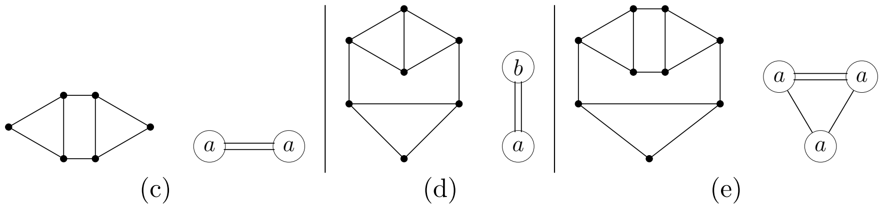


Figure 7 Patterns (c),(d),(e) and their corresponding contracted graph G_{ab}.

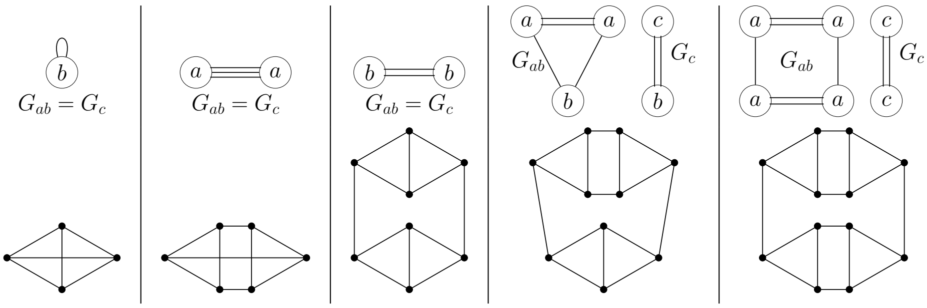


Figure 8 Claw-free cubic graphs for which G_C contains multi-edges or loops: for each graph G on the bottom row, we display its associated contracted graphs G_{ab} and G_C on the top row.

▶ **Definition 13** (Canonical code for a labeled graph). *Let $G = (V, E)$ be a connected graph, L be a finite set of labels, and $l : V \to L$ be a vertex labeling function. A code is a sequence $c \cdot l_0 l_1 \dots l_n$, where $n = \#V - 1$, c is a BFS-based code of G and $\forall i \in [0, n], l_i \in L$ is the label of the vertex whose q-number is i. The canonical code, denoted $ccl(G)$, is the lexicographically smallest among all possible codes generated by different BFS traversals of G.*

Under this definition, the canonical code for the contracted graph in Fig. 6 is represented as $000, adde$. Given a connected claw-free cubic graph G of order n, the size of the corresponding contracted graph G_c is not fixed but instead depends on the types and frequencies of patterns present in G. Hence, we introduce a new integer variable N_x for each $x \in \{a, b, c, d, e\}$ that corresponds to the number of x-meta-vertices. We also introduce two new integer variables n_C and m_C that correspond to the number of variables and edges of the contracted graph G_C. Finally, for each $i \in [0, n_C - 1]$, we introduce a variable l_i which represents the label of the ith meta-vertex of G_C. The following constraints must be satisfied:

- $n_C = N_a + N_b + N_c + N_d + N_e$
- $m_C = \frac{3n}{2} - (3N_a + 5N_b + 8N_c + 10N_d + 13N_e)$
- $n = 3N_a + 4N_b + 6N_c + 7N_d + 9N_e$
- Connectedness: $n_C \leq m_C + 1$
- Occurrence of vertices with the same label:
  $\forall k \in \{a, b, c, d, e\}, N_k = \#\{i \mid l_i = k,\ i \in [0, n_C - 1]\}$
- Degree constraints:
  $\forall i \in [0, n_C - 1],\ d_i = 3 \Leftrightarrow l_i = a \wedge d_i = 2 \Leftrightarrow l_i \in \{b, c\} \wedge d_i = 1 \Leftrightarrow l_i \in \{d, e\}$

In this model, the 5 special graphs shown in Fig 8 are not addressed. However, they are accounted for when counting the number of solutions for $n \in \{4, 6, 8, 10, 12\}$.

We add to these constraints the constraints C1 to C10 of Fig. 2, while replacing n and m with n_C and m_C. We also extend our global canonicity constraint by ensuring that the label sequence is the smallest possible one under all possible BFS traversals.

Table 3 presents the solving time for enumerating all contracted graphs corresponding to all claw-free cubic graphs for $n \in \{20, \ldots, 60\}$ when the frequency parameter f is set to 1 (greater values for f do not improve results). Since the contracted graph is significantly smaller than the original claw-free cubic graph, we observe a notable reduction in solving times. For instance, when $n = 36$, generating 6914 solutions takes only 2.2 seconds, whereas enumerating the primary claw-free cubic graphs requires 32.9 seconds. With this approach, we can generate graphs up to $n = 60$ within 9 hours.

Table 3 Results for generating all claw-free cubic graphs when using contracted graphs.

n	graphs	time	n	graphs	time	n	graphs	time
20	15	0.1s	34	1,502	1.6s	48	418,112	131.2s
22	27	0.2s	36	3,187	2.2s	50	1,005,927	293.9s
24	54	0.3s	38	6,914	3.2s	52	2,412,987	756.0s
26	94	0.3s	40	15,025	5.9s	54	5,934,636	1,877.8s
28	181	0.4s	42	33,687	11.8s	56	14,823,532	4,526.9s
30	369	0.7s	44	77,450	23.6s	58	37,005,614	11,836.1s
32	731	0.8s	46	177,465	54.1s	60	94,412,125	30,948.9s

9 Conclusion

We introduce a new canonical graph encoding based on a BFS traversal, which is well-suited for constraint-based approaches. Using this encoding, we define a CP model that enumerates all non-isomorphic graphs satisfying specific constraints by generating their corresponding canonical codes. The key aspect of this encoding is identifying a spanning tree that yields the lexicographically smallest representation of the graph. We demonstrate how this encoding enables the formulation of fundamental static symmetry-breaking constraints. Additionally, we leverage the crucial property that every prefix of a canonical code must also be canonical, allowing us to define a global constraint that dynamically enforces symmetry-breaking, ensuring the canonicity of the generated codes.

We tested our approach on connected claw-free cubic graphs, which are regular and inherently challenging for BFS-based methods. However, our experimental results demonstrate that our approach outperforms existing state-of-the-art techniques based on adjacency matrix representations. Beyond this, we explored the properties of local structures within this class of graphs that allow us to contract graphs into more compact labeled graphs. Our BFS-based canonical code may be easily extended to labeled graphs, and this allows us to enumerate graphs of larger order, up to 60 vertices. As part of our future work, we aim to refine our encoding for such labeled graphs and develop dedicated propagation strategies.

Our approach is well-suited for enumerating graphs with a limited diameter, such as diameter-2-critical graphs and required-girth extremal graphs [11]: In these cases, the number of canonical spanning trees remains manageable, allowing them to be precomputed and used for static symmetry breaking. We plan to publish further results on these aspects in the near future, as they could not be included in this work due to space limitations.

References

1 Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

2 Nicolas Beldiceanu, Irit Katriel, and Xavier Lorca. Undirected forest constraints. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Re-*

search (OR) Techniques in Constraint Programming, pages 29–43. Springer, 2006. doi:10.1007/11757375_5.

3 Michael Codish, Graeme Gange, Avraham Itzhakov, and Peter J Stuckey. Breaking symmetries in graphs: the nauty way. In *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*, pages 157–172. Springer, 2016. doi:10.1007/978-3-319-44953-1_11.

4 Michael Codish, Alice Miller, Patrick Prosser, and A Stuckey. Breaking symmetries in graph representation, 2013.

5 Laurianne David, Amol Thakkar, Rocío Mercado, and Ola Engkvist. Molecular representations in ai-driven drug discovery: a review and practical guide. *Journal of cheminformatics*, 12(1):56, 2020. doi:10.1186/S13321-020-00460-5.

6 Michel Habib and Christophe Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 145(2):183–197, 2005. doi:10.1016/J.DAM.2004.01.011.

7 Michael A Henning and Christian Löwenstein. Locating-total domination in claw-free cubic graphs. *Discrete Mathematics*, 312(21):3107–3116, 2012. doi:10.1016/J.DISC.2012.06.024.

8 Marijn JH Heule. Optimal symmetry breaking for graph problems. *Mathematics in Computer Science*, 13:533–548, 2019. doi:10.1007/S11786-019-00397-5.

9 Avraham Itzhakov and Michael Codish. Breaking symmetries in graph search with canonizing sets. *Constraints*, 21:357–374, 2016. doi:10.1007/S10601-016-9244-Z.

10 Avraham Itzhakov and Michael Codish. Breaking symmetries with high dimensional graph invariants and their combination. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 133–149. Springer, 2023. doi:10.1007/978-3-031-33271-5_10.

11 Markus Kirchweger and Stefan Szeider. Sat modulo symmetries for graph generation and enumeration. *ACM Transactions on Computational Logic*, 2024.

12 Vianney Le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, Uppsala, Sweden, September 2013. URL: https://hal.science/hal-01339250.

13 Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of symbolic computation*, 60:94–112, 2014. doi:10.1016/J.JSC.2013.09.003.

14 Vyacheslav Moklev and Vladimir Ulyantsev. Bfs enumeration for breaking symmetries in graphs, 2018. arXiv:1804.02273.

15 Siegfried Nijssen and Joost N Kok. The gaston tool for frequent subgraph mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77–87, 2005. doi:10.1016/J.ENTCS.2004.12.039.

16 Xiao Peng and Christine Solnon. Using canonical codes to efficiently solve the benzenoid generation problem with constraint programming. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

17 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. doi:10.21105/joss.04708.

18 Vladimir Ulyantsev, Ilya Zakirzyanov, and Anatoly Shalyto. Bfs-based symmetry breaking predicates for DFA identification. In *Language and Automata Theory and Applications - 9th International Conference, LATA 2015*, volume 8977 of *Lecture Notes in Computer Science*, pages 611–622. Springer, 2015. doi:10.1007/978-3-319-15579-1_48.

19 David Weininger. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988. doi:10.1021/CI00057A005.

20 Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, 2003. doi:10.1145/956750.956784.