Using Canonical Codes to Efficiently Solve the Benzenoid Generation Problem with Constraint Programming

Xiao Peng, Christine Solnon

To cite this version:

Xiao Peng, Christine Solnon. Using Canonical Codes to Efficiently Solve the Benzenoid Generation Problem with Constraint Programming. CP 2023 - 29th International Conference on Principles and Practice of Constraint Programming, Aug 2023, Toronto (CA), Canada. 10.4230/LIPIcs.CP.2023.17 . hal-04156847

HAL Id: hal-04156847
https://hal.science/hal-04156847v1

Submitted on 9 Jul 2023

Using Canonical Codes to Efficiently Solve the Benzenoid Generation Problem with Constraint Programming

Xiao Peng
Univ Lyon, INSA Lyon, Inria, CITI, EA3720, 69621 Villeurbanne, France

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

Abstract

The Benzenoid Generation Problem (BGP) aims at generating all benzenoid molecules that satisfy some given properties. This problem has important applications in chemistry, and Carissan et al (2021) have shown us that Constraint Programming (CP) is well suited for modelling this problem because properties defined by chemists are easy to express by means of constraints. Benzenoids are described by hexagon graphs and a key point for an efficient enumeration of these graphs is to be invariant to rotations and symmetries. In this paper, we introduce canonical codes that uniquely characterise hexagon graphs while being invariant to rotations and symmetries. We show that these codes may be defined by means of constraints. We also introduce a global constraint for ensuring that codes are canonical, and a global constraint for ensuring that a pattern is included in a code. We experimentally compare our new CP model with the CP-based approach of Carissan et al (2021), and we show that it has better scale-up properties.

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

Keywords and phrases Benzenoid Generation Problem, Canonical Code, Hexagon Graph

Digital Object Identifier 10.4230/LIPIcs.CP.2023.17

Acknowledgements We want to thank authors of [3] who helped us reproduce their results.

1 Introduction

Benzenoids are hydrocarbon molecules whose carbon atoms are forming cycles of size 6, *i.e.*, hexagons. An important chemistry problem concerns the generation of all benzenoids that satisfy some given properties [5]. This problem is called the Benzenoid Generation Problem (BGP) in [2]. As benzenoids are regular tillings with hexagonal faces, they may be represented by hexagon graphs such that a vertex is associated with every hexagonal face and an edge with every pair of vertices corresponding to adjacent faces [1, 2]. For example, we display in Figure 1 a benzenoid composed of five hexagonal faces and its associated hexagon graph. These hexagon graphs are always connected.

Different benzenoids may be represented with isomorphic hexagon graphs. For example, let us consider the hexagon graph displayed in Figure 1. The subgraph induced by c, d, and e is isomorphic to the subgraph induced by a, c, and d whereas their associated molecules are different because hexagons c, d, and e are not aligned whereas hexagons a, c, and d are aligned. To overcome this problem, we take into account edge directions, considering the six directions defined in Figure 1. In this case, the subgraph induced by c, d, and e is no longer isomorphic to the subgraph induced by a, c, and d because edges (c, d) and (d, e) have different directions (0 and 1), whereas edges (a, c) and (c, d) have the same direction (0).

To solve the BGP, we have to enumerate hexagon graphs that satisfy some given properties and these properties are usually easily expressed by means of constraints (such as, for example,

29th International Conference on Principles and Practice of Constraint Programming (CP 2023).
Editor: Roland H. C. Yap; Article No. 17; pp. 17:1–17:17
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

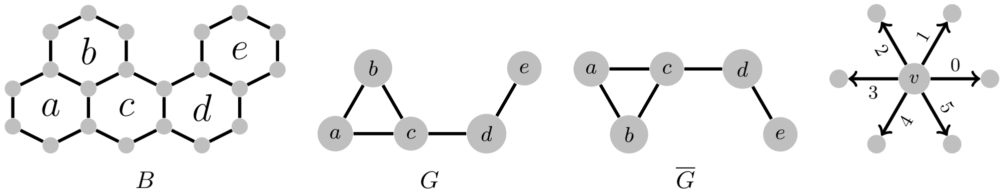


Figure 1 Example of benzenoid B and its associated hexagon graph G, and symmetrical graph $\overline{G}$. Right: Edge directions from a vertex v to each of its 6 possible neighbours.

the absence of cliques of order three, or the inclusion of some patterns). Two main approaches may be used:

- The dedicated approach of [1] uses canonical codes to represent hexagon graphs. These codes are invariant to rotations and symmetries and they are used to decide whether two graphs are isomorphic or not in linear time. However, these codes cannot represent benzenoids with holes (called coronoids). Furthermore, this approach is not declarative and it does not allow one to easily add constraints on the graphs to be enumerated.
- The Constraint Programming (CP)-based approach of [2, 3] basically searches for all connected subgraphs within an initial hexagon graph, and it is implemented in Choco [11] using graph variables. Using CP allows one to easily add constraints and this approach is able to generate all kinds of benzenoids, including coronoids. However, it is less efficient than the dedicated approach.

In this paper, we introduce an approach which is both efficient and declarative: like [1], it is based on canonical codes but these canonical codes can represent all benzenoids (including coronoids); like [2, 3], it is based on CP so that one may easily add constraints to specify structural properties. In Section 2, we introduce our new canonical code and study some of its properties that are used to efficiently generate canonical codes with CP. In Section 3, we extend canonical codes to the case where benzenoids are constrained to contain some given patterns. In Section 4, we introduce CP models for solving BGPs. In Section 5, we report experimental results and compare our approach with the CP-based approach of [2, 3].

Notations

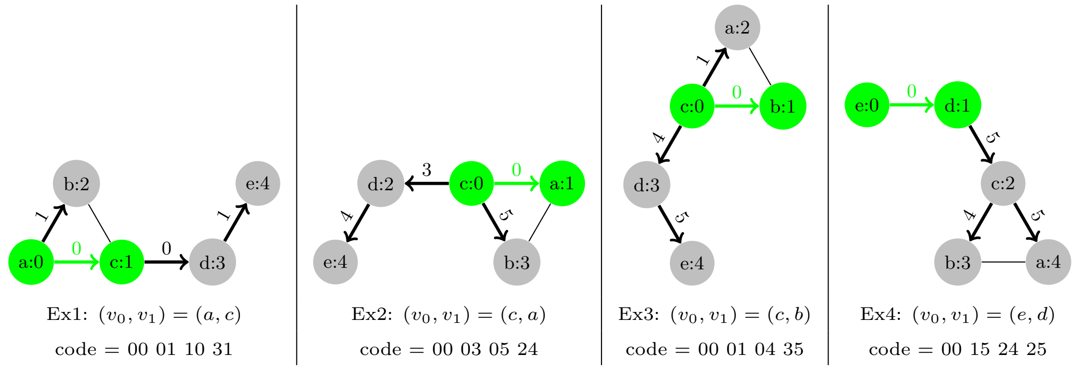
Given two integer values i and j, we note $[i, j]$ the set of all integer values ranging from i to j. Given a hexagon graph G, we note $\overline{G}$ the symmetrical graph obtained by mirroring G with respect to the x-axis, as displayed in Figure 1. Given a hexagon graph $G = (V, E)$ and a subset of vertices $S \subseteq V$, we note $G_{\downarrow S}$ the subgraph of G induced by S, *i.e.*, $G_{\downarrow S} = (S, E \cap S \times S)$.

2 Representation of Hexagon Graphs with Canonical Codes

A key point for an efficient enumeration of hexagon graphs is to be invariant to rotations and symmetries. For example, the graph displayed in Figure 1 is isomorphic to any other graph obtained by rotating it of $k * 60°$ with $k \in \mathbb{N}$ and/or mirroring it with respect to the x-axis. In [3], constraints are added to break these symmetries and avoid generating several times the same graph. In this paper, we propose to use another approach based on canonical codes, *i.e.*, integer sequences that uniquely characterise graphs and that are invariant to rotations and symmetries. We introduce our canonical code in Section 2.1. We study its properties in Section 2.2. We compare it with related work in Section 2.3.

Algorithm 1 $\text{BFS}_G(v_0, v_1)$

```
   Input: A hexagon graph $G$ and an initial edge $(v_0, v_1)$ of $G$
   Output: The code associated with a BFS of $G$ started from $(v_0, v_1)$
1  rotate $G$ so that the direction of edge $(v_0, v_1)$ is equal to 0
2  for each vertex $v_i$ of $G$ do initialise $num[v_i]$ to $-1$;
3  set $num[v_0]$ to 0 and initialise a counter $c$ to 1
4  let $q$ be an empty FIFO queue; add $v_0$ in $q$
5  initialise code to an empty sequence
6  while $q$ is not empty do
7      remove from $q$ its oldest vertex $v_i$
8      for each edge $(v_i, v_j)$ of $G$ taken by order of increasing direction do
9          if $num[v_j] < 0$ then
10             add $v_j$ in $q$
11             set $num[v_j]$ to $c$ and increment $c$
12         add $num[v_i]$ and the direction of edge $(v_i, v_j)$ at the end of code
13 return code
```



Ex1: $(v_0, v_1) = (a, c)$

code = 00 01 10 31

Ex2: $(v_0, v_1) = (c, a)$

code = 00 03 05 24

Ex3: $(v_0, v_1) = (c, b)$

code = 00 01 04 35

Ex4: $(v_0, v_1) = (e, d)$

code = 00 15 24 25

Figure 2 Examples of runs of $\text{BFS}_G(v_0, v_1)$ when G is the graph of Figure 1 and (v_0, v_1) is equal to (a, c) for Ex1, (c, a) for Ex2, (c, b) for Ex3, and (e, d) for Ex4. For each vertex $v_i \in \{a, b, c, d, e\}$, we display $v_i : num[v_i]$ in circles. Each edge (v_i, v_j) used lines 10-12 is displayed in bold and its direction is displayed on top of it (the initial edge (v_0, v_1) is displayed in green).

2.1 Definition of Canonical Codes

Given a hexagon graph G with $n + 1$ vertices, a code is a sequence of n couples of integer values which is associated with a Breadth First Search (BFS) of G starting from a given initial edge (v_0, v_1), as described in Algorithm 1. The initial edge is used to define the orientation of the graph (line 1): we always consider the orientation such that v_1 is on the right of v_0. This allows us to be invariant to rotations.

▶ **Example 1.** We display in Figure 2 different runs of BFS on the hexagon graph of Figure 1, starting from different edges. In Ex1, we start from edge (a, c) which already has direction 0 so that we do not have to rotate the graph. When starting from edge (c, a) (resp. (c, b) and (e, d)), we have to rotate the graph of 180° (resp. 240° and 60°).

BFS assigns a different number $num[v_i]$ to each vertex v_i corresponding to the order vertices are discovered: v_0 is numbered with 0 (line 3) and when a vertex v_j is reached for

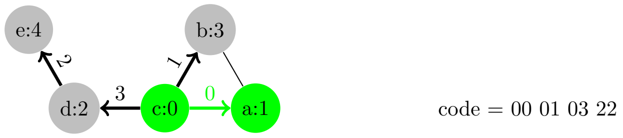


Figure 3 Run of $\text{BFS}_{\overline{G}}(c, a)$ where $\overline{G}$ is the graph symmetrical to the graph G of Figure 1.

the first time it is numbered with the next non-assigned value c (line 11). Each time the search discovers a new vertex v_j using an edge (v_i, v_j), $num[v_i]$ and $d(v_i, v_j)$ are added at the end of the code, where $d(v_i, v_j)$ is the direction of edge (v_i, v_j) (line 12).

▶ **Example 2.** In Figure 2 (Ex1), we illustrate a run of $\text{BFS}_G(a, c)$ when G is the graph of Figure 1. In this case, the code is 00 01 10 31: the first couple is 00 because vertex 1 (c) has been reached from vertex 0 (a) and $d(a, c) = 0$; the second couple is 01 because vertex 2 (b) has been reached from vertex 0 (a) and $d(a, b) = 1$; the third couple is 10 because vertex 3 (d) has been reached from vertex 1 (c) and $d(c, d) = 1$; the fourth couple is 31 because vertex 4 (e) has been reached from vertex 3 (d) and $d(d, e) = 1$.

Note that the loop lines 8-12 considers edges outgoing from v_i by order of increasing directions. This ensures that we always compute the same code provided that the direction of the first edge (v_0, v_1) is fixed to 0.

Given a code $p_1d_1\ p_2d_2 \dots p_nd_n$ computed from a graph with $n + 1$ vertices, we can draw this graph, starting from vertex number 0: for each $i \in [1, n]$, p_i gives the number of the predecessor of vertex number i and d_i gives the direction of the edge (p_i, i). Once all vertices have been drawn, missing edges can be added as every vertex is connected to all its neighbours. For example, in Ex1, we add an edge between vertices 1 and 2 because they have neighbour positions.

There exist different possible codes for a given graph. Each code corresponds to a different BFS starting from a different initial edge (v_0, v_1). We define a total order on the set of all possible codes that may be associated with a given graph by considering a lexicographic order. Among all the possible codes for a graph, the smallest one according to this order is called the canonical code of this graph and it is unique.

So far, our canonical code is invariant to rotations but not to symmetries. To become invariant to symmetries, we must also compute all codes starting from all possible edges while considering the symmetrical graph $\overline{G}$. This leads us to the following definition.

▶ **Definition 3** (Canonical code $cc(G)$)**.** *The canonical code of a hexagon graph $G = (V, E)$ is: $cc(G) = \min\{BFS_G(v_0, v_1), BFS_{\overline{G}}(v_0, v_1) : (v_0, v_1) \in E\}$ when considering a lexicographic order to compare codes.*

▶ **Example 4.** The smallest code that may be computed for the graph G of Figure 1 is $\text{BFS}_G(c, b)$, displayed in Figure 2. However, if we consider the symmetrical graph $\overline{G}$, $\text{BFS}_{\overline{G}}(c, a)$ is smaller than $\text{BFS}_G(c, b)$ (see Figure 3). This code is the smallest one for both G and $\overline{G}$, *i.e.*, $cc(G) = 00\ 01\ 03\ 22$.

2.2 Properties of Canonical Codes

The next two theorems are used to efficiently enumerate canonical codes with CP in Section 4.

▶ **Theorem 5.** *Given a code $p_1d_1\ p_2d_2 \dots p_nd_n$, we have:*
Property a: $\forall i \in [1, n-1], p_i \leq p_{i+1}$*;*

Property b: $\forall i \in [1, n-1], (p_i = p_{i+1}) \Rightarrow (d_i < d_{i+1})$.

Proof. Property a is a consequence of the fact that (i) q is a FIFO queue, (ii) vertices are added in q by order of increasing number, and (iii) when a vertex v_i is removed from q we add to *code* all couples $p_i d_i$ such that $p_i = num[v_i]$ and the vertex at direction d_i from v_i is not yet numbered. Hence, all couples added to *code* during one iteration of the loop lines 6-12 have the same value for p_i. Property b is a straightforward consequence of the fact that the loop lines 8-12 considers edges by order of increasing directions. ◀

▶ **Theorem 6.** *Let $c = p_1 d_1\ p_2 d_2 \ldots p_n d_n$ be a canonical code. For each $i \in [1, n-1]$, the prefix $c_i = p_1 d_1\ p_2 d_2 \ldots p_i d_i$ of c is also a canonical code.*

Proof. Let G and G_i be the graphs associated with c and c_i, where each vertex is numbered with $num[v_i]$. As c_i is a prefix of c, G_i is the subgraph of G induced by the vertices numbered from 0 to i. Let us assume that c_i is not canonical, *i.e.*, there exists another code $c'_i = p'_1 d'_1\ p'_2 d'_2 \ldots p'_i d'_i$ that is smaller than c_i while representing the same subgraph G_i. Let (v_0, v_1) be the edge of G_i such that $c'_i = \text{BFS}_{G_i}(v_0, v_1)$ (resp. $c'_i = \text{BFS}_{\overline{G_i}}(v_0, v_1)$ if c'_i is computed in the symmetrical graph). Let $c' = \text{BFS}_G(v_0, v_1)$ (resp. $c' = \text{BFS}_{\overline{G}}(v_0, v_1)$) be the code obtained by performing a search of G (resp. $\overline{G}$) that starts from the same initial edge as the one used to obtain c'_i. Let j be the smallest vertex number for which c' and c'_i have different values, *i.e.*, $\forall k < j$, vertex number k has the same edges in both G and G_i, whereas vertex number j has more edges in G than in G_i. More precisely, let S (resp. S_i) be the set of edges outgoing from j in G (resp. G_i). We have $S_i \subset S$ as G_i is a subgraph of G. We have to distinguish two different cases.

Case 1: the largest edge direction in S_i is equal to the largest edge direction in S. In this case, c' is smaller than c'_i because the sequence of couples associated with the successors of j in c' is necessarily smaller than the sequence of couples associated with the successors of j in c'_i (*e.g.*, if edge directions in S_i are $\{1, 5\}$ whereas edge directions in S are $\{1, 2, 5\}$, then $j1\ j2\ j5 < j1\ j5$).

Case 2: the largest edge direction in S_i is smaller than the largest edge direction in S. In this case, either c'_i is a prefix of c' (if j is the greatest vertex with an outgoing edge), or c' is smaller than c'_i (*e.g.*, if edge directions in S_i are $\{1, 2\}$ whereas edge directions in S are $\{1, 2, 5\}$, $j1\ j2\ j5 < j1\ j2\ k$ because $k > j$, as stated in Property a).

In both cases, this implies that $c' < c$ which contradicts the fact that c is canonical. ◀

Computational complexity. The canonical code of a graph with n vertices is computed in $\mathcal{O}(n^2)$: BFS is run twice for each edge and the number of edges is in $\mathcal{O}(n)$ as a vertex cannot have more than six neighbours; the time complexity of BFS is $\mathcal{O}(n)$ as the loop lines 6-12 is iterated n times and the loop lines 8-12 is iterated at most six times.

We may speed-up the computation by avoiding some calls to BFS. To this aim, we display in Figure 4 the 13 different possible patterns for the neighbourhood of a vertex v (up to rotations). For each pattern P_i such that v has k_i successors, every code computed by BFS with $v_0 = v$ starts with a prefix $0d_1\ 0d_2\ \ldots\ 0d_{k_i}$, and we give below the smallest possible prefix composed of $2k_i$ integers, denoted π_i, the set S of edges (v, v_k) of G such that $\text{BFS}_G(v, v_k)$ starts with π_i, and the set $\overline{S}$ of edges (v, v_k) of G such that $\text{BFS}_{\overline{G}}(v, v_k)$ starts with π_i (these edges are highlighted in Figure 4).

$\pi_6 = 00\ 01\ 02\ 03\ 04\ 05$	$S = \{(v, v_i) : i \in [1, 6]\}$	$\overline{S} = \{(v, v_i) : i \in [1, 6]\}$
$\pi_5 = 00\ 01\ 02\ 03\ 04$	$S = \{(v, v_1)\}$	$\overline{S} = \{(v, v_5)\}$
$\pi_4 = 00\ 01\ 02\ 03$	$S = \{(v, v_1)\}$	$\overline{S} = \{(v, v_4)\}$
$\pi_{3+1} = 00\ 01\ 02\ 04$	$S = \{(v, v_1)\}$	$\overline{S} = \{(v, v_3)\}$
$\pi_{2+2} = 00\ 01\ 03\ 04$	$S = \{(v, v_1), (v, v_3)\}$	$\overline{S} = \{(v, v_2), (v, v_4)\}$

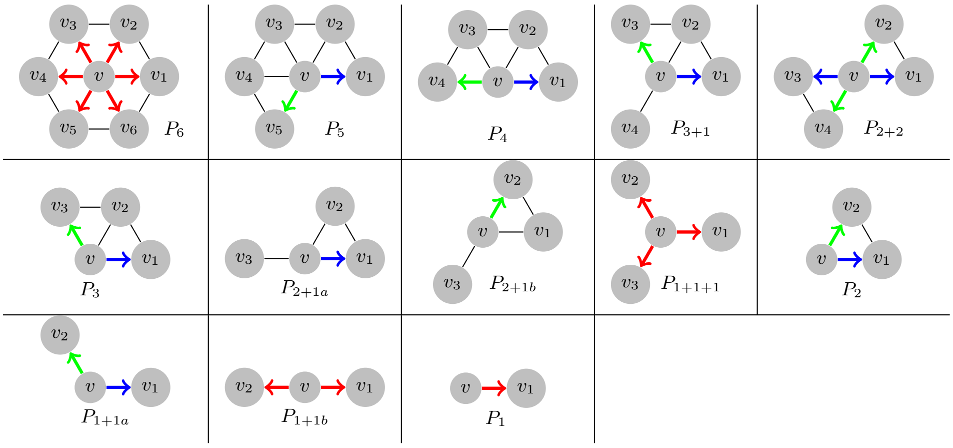


Figure 4 Different neighbourhood patterns for a vertex v. We highlight in blue (resp. green and red) every edge (v, v_k) such that $\mathrm{BFS}_G(v, v_k)$ (resp. $\mathrm{BFS}_{\overline{G}}(v, v_k)$ and both $\mathrm{BFS}_G(v, v_k)$ and $\mathrm{BFS}_{\overline{G}}(v, v_k)$) starts with the smallest possible prefix.

$\pi_3 = 00\ 01\ 02$	$S = \{(v, v_1)\}$	$\overline{S} = \{(v, v_3)\}$
$\pi_{2+1a} = 00\ 01\ 03$	$S = \{(v, v_1)\}$	$\overline{S} = \emptyset$
$\pi_{2+1b} = 00\ 01\ 03$	$S = \emptyset$	$\overline{S} = \{(v, v_2)\}$
$\pi_{1+1+1} = 00\ 03\ 05$	$S = \{(v, v_1), (v, v_2), (v, v_3)\}$	$\overline{S} = \{(v, v_1), (v, v_2), (v, v_3)\}$
$\pi_2 = 00\ 01$	$S = \{(v, v_1)\}$	$\overline{S} = \{(v, v_2)\}$
$\pi_{1+1a} = 00\ 02$	$S = \{(v, v_1)\}$	$\overline{S} = \{(v, v_2)\}$
$\pi_{1+1b} = 00\ 03$	$S = \{(v, v_1), (v, v_2)\}$	$\overline{S} = \{(v, v_1), (v, v_2)\}$
$\pi_1 = 00$	$S = \{(v, v_1)\}$	$\overline{S} = \{(v, v_1)\}$

▶ **Example 7.** Let us consider the graph G of Figure 1. The pattern of vertex a in G is P_2 (with $a = v$, $c = v_1$, and $b = v_2$). Therefore, the smallest possible code computed with $v_0 = a$ starts with the prefix $\pi_2 = 00\ 01$, and the two codes that start with this prefix are $\mathrm{BFS}_G(a, c)$ and $\mathrm{BFS}_{\overline{G}}(a, b)$. Also, the pattern of vertex c in G is P_{2+1b} (with $c = v$, $a = v_2$, $b = v_1$, and $d = v_3$). Therefore, the smallest possible code computed with $v_0 = c$ starts with the prefix $\pi_{2+1a} = 00\ 01\ 03$, and the only code that starts with this prefix is $\mathrm{BFS}_{\overline{G}}(c, a)$.

To further reduce the number of searches, we define a total order among patterns.

▶ **Definition 8.** *[Order between patterns] Let P_i and P_j be two patterns (as listed in Figure 4), and let $\pi_i 1$ and $\pi_j 1$ be the sequences obtained by adding "1" at the end of π_i and π_j, respectively. We define $P_i < P_j$ (resp. $P_i = P_j$) if $\pi_i 1$ is lexicographically smaller than (resp. equal to) $\pi_j 1$. We obtain the following order:*

$$P_6 < P_5 < P_4 < P_{3+1} < P_3 < P_{2+2} < P_{2+1a} = P_{2+1b} < P_2 < P_{1+1+1} < P_{1+1a} < P_{1+1b} < P_1$$

Given two vertices v_i and v_j, if the pattern of v_i is smaller than the pattern of v_j, then we know that every code computed from v_i is smaller than every code computed from v_j. Hence, to compute the canonical code of a hexagon graph G, we first search for the smallest pattern P_k in G. Then, for each vertex v_i such that the pattern of v_i in G is equal to P_k, we compute all codes from the starting edges highlighted in Figure 4. Finally, we return the smallest computed code.

▶ **Example 9.** Let us consider the graph G of Figure 1. The pattern of vertex a (resp. b, c, d, and e) is P_2 (resp. P_2, P_{2+1b}, P_{1+1a}, and P_1). The smallest pattern is P_{2+1b}. Therefore, the canonical code is obtained by running $\mathrm{BFS}_{\overline{G}}(c, a)$, and it is useless to run BFS_G or $\mathrm{BFS}_{\overline{G}}$ from any other edge.

Note that this does not change the time complexity as in the worst case we may have $\mathcal{O}(n)$ vertices with a smallest pattern. For example, when we have an horizontal chain of vertices, all vertices have pattern P_{1+1b}, except the two endpoints which have pattern P_1. In this case, we must run BFS_G and $\mathrm{BFS}_{\overline{G}}$ for every edge outgoing from vertices with pattern P_{1+1b} (*i.e.*, all vertices but the two endpoints).

2.3 Comparison with other canonical codes

In the general case, building the canonical code of a graph is a problem which is not known to be in $\mathcal{P}$ nor to be $\mathcal{NP}$-complete (it is isomorphic-complete). There exist rather efficient algorithms such as Nauty [9], but these algorithms have an exponential time complexity in the worst case. Canonical codes are widely used in graph mining tools such as gSpan [12] or Gaston [10]. When graphs are embedded in a 2D space, canonical codes may be computed in polynomial time [8]. These canonical codes may be simplified when considering grid graphs such that all faces are squares [6]. Canonical codes defined in [12, 10, 6] share some similarities with our canonical codes as they are computed by performing graph traversals and assigning numbers to vertices according to the order of discovery. However, in [12, 10, 6], traversals are Depth First Searches (DFSs). Performing BFSs instead of DFSs allows us to avoid some calls to BFS (as explained in Section 2.2). It also allows us to exploit properties a and b to define codes by means of constraints (see Section 4).

In [1], canonical codes are introduced for benzenoid structures. These codes only describe boundary cycles and assume that there is no hole within these cycles, thus preventing the representation of coronoids. Our codes fully describe hexagon graphs, including vertices inside the boundary cycle and, therefore, we can represent coronoids.

3 Canonical Codes in Case of Required Patterns

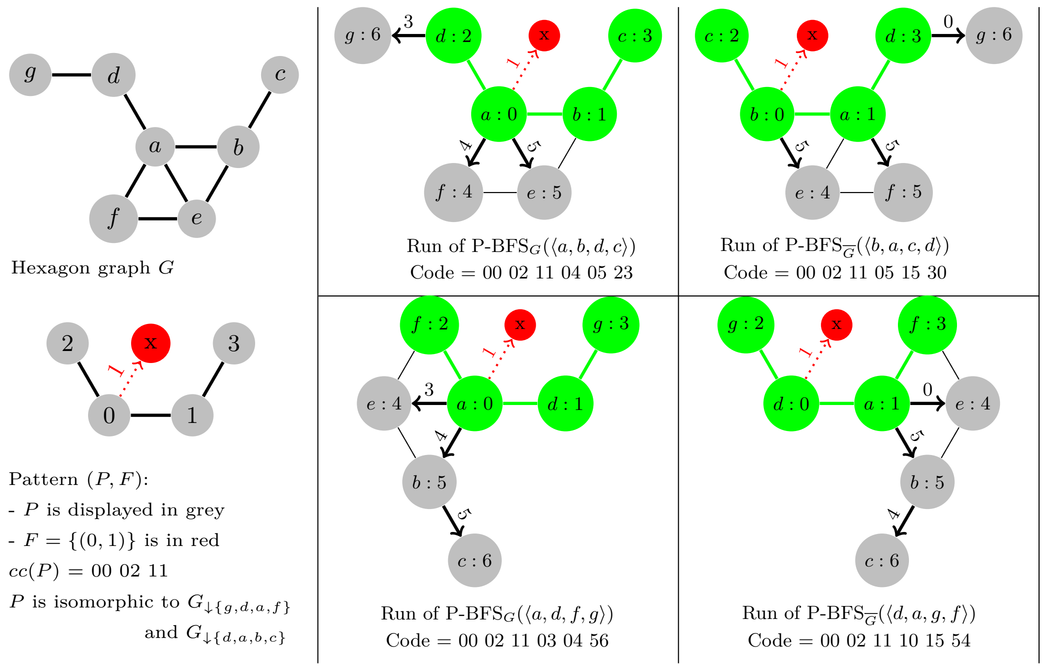
Many BGPs involve enumerating hexagon graphs that contain some given patterns, where patterns both specify mandatory and forbidden vertices [3]. More precisely, a pattern is defined by a couple (P, F) such that P is a hexagon graph that specifies mandatory vertices, and F is a set of couples that specify forbidden vertex positions, *i.e.*, for each couple $(p_i, d_i) \in F$, vertex p_i must not have an outgoing edge in direction d_i.

▶ **Example 10.** The pattern (P, F) displayed in Figure 5 has four mandatory vertices in grey and one forbidden vertex in red. (P, F) occurs twice in G as P is isomorphic to $G_{\downarrow\{d,a,b,c\}}$ (resp. $G_{\downarrow\{g,d,a,f\}}$) and a has no outgoing edge in direction 1 (resp. a has no outgoing edge in direction 1 when rotating G so that $G_{\downarrow\{g,d,a,f\}}$ has the same orientation as P).

To efficiently generate graphs that contain a pattern, the idea is to constrain canonical codes to start with this pattern so that all generated graphs contain it by construction. We introduce this new canonical code in Section 3.1, and we study its properties in Section 3.2.

3.1 Definition of P-canonical codes

To ensure that codes start with the canonical code of a given pattern (P, F), we perform BFSs of G and $\overline{G}$ from a given P-sequence that corresponds to an occurrence of (P, F) in G or $\overline{G}$. More precisely, this P-sequence is defined as follows.



Hexagon graph G

Pattern (P, F):

- P is displayed in grey
- $F = \{(0, 1)\}$ is in red

$cc(P)$ = 00 02 11

P is isomorphic to $G_{\downarrow\{g,d,a,f\}}$ and $G_{\downarrow\{d,a,b,c\}}$

Run of P-BFS$_G(\langle a, b, d, c\rangle)$
Code = 00 02 11 04 05 23

Run of P-BFS$_{\overline{G}}(\langle b, a, c, d\rangle)$
Code = 00 02 11 05 15 30

Run of P-BFS$_G(\langle a, d, f, g\rangle)$
Code = 00 02 11 03 04 56

Run of P-BFS$_{\overline{G}}(\langle d, a, g, f\rangle)$
Code = 00 02 11 10 15 54

Figure 5 Left: Example of hexagon graph G and pattern (P, F). Right: To compute $cc_{P,F}(G)$, we run P-BFS$_G$ (resp. P-BFS$_{\overline{G}}$) with the P-sequences $\langle a, b, d, c\rangle$ and $\langle a, d, f, g\rangle$ (resp. $\langle b, a, c, d\rangle$ and $\langle d, a, g, f\rangle$). The P-canonical code of G is $cc_{P,F}(G)$ = 00 02 11 03 04 56.

▶ **Definition 11** (P-sequence of (G, P, F)). *Let G be a hexagon graph with n vertices, and (P, F) be a pattern with $1 < k < n$ mandatory vertices. A P-sequence of (G, P, F) is a sequence $s = \langle v_0, v_1, \ldots, v_{k-1}\rangle$ of k vertices of G such that*

(i) *the subgraph of G induced by $\{v_0, \ldots, v_{k-1}\}$,* i.e., *$G_{\downarrow s}$, is isomorphic to P;*

(ii) *$\forall (p_i, d_i) \in F$, $G_{\downarrow s}$ has no outgoing edge from the vertex corresponding to p_i in direction d_i when rotating G so that $G_{\downarrow s}$ and P have the same orientation;*

(iii) *the code returned by $BFS_{G_{\downarrow s}}(v_0, v_1)$ is canonical;*

(iv) *for each $i \in [0, k-1]$ the number assigned by $BFS_{G_{\downarrow s}}(v_0, v_1)$ to v_i is equal to i,* i.e., *$num[v_i] = i$ at line 13 of Algorithm 1.*

Each P-sequence corresponds to an occurrence of (P, F) in G, and (P, F) may occur more than once in G. To compute all P-sequences of (G, P, F), we iterate on each edge of G and try to build a P-sequence that starts with this edge using Algorithm 2: given the canonical code c of P, an edge (v_0, v_1) of G, and a set F of forbidden positions, seq(G, v_0, v_1, c, F) returns the P-sequence of (G, P, F) that starts with $\langle v_0, v_1\rangle$ if it exists, and it returns *null* otherwise. To build the P-sequence, it uses the canonical code c to associate vertices of G to vertex numbers used in the canonical code: for each vertex number $i \in [0, k-1]$, $vertex[i]$ is the vertex of G that corresponds to i, and for each couple $p_i d_i$ in the canonical code, we ensure that vertex i in P corresponds to the vertex of G that is at direction d_i from $vertex[p_i]$, if it exists (line 5); if it does not exist, then there is no P-sequence that starts from (v_0, v_1) and we return *null* (line 4). When a P-sequence is completed, we check that there are no vertices at forbidden positions (line 6).

To be invariant to symmetries, we must also compute all P-sequences of $(\overline{G}, P, F)$ by running seq$(\overline{G}, v_0, v_1, c, F)$ for each edge (v_0, v_1) of $\overline{G}$.

Algorithm 2 seq(G, v_0, v_1, c, F)

```
Input: A hexagon graph $G$, an edge $(v_0, v_1)$ of $G$, the canonical code
       $c = p_1 d_1 \ \dots \ p_{k-1} d_{k-1}$ of a pattern $P$, and the set $F = \{(p'_1, d'_1), \dots, (p'_f, d'_f)\}$
       of forbidden vertices
Output: A P-sequence of $(G, P, F)$ starting with $\langle v_0, v_1 \rangle$, or $null$ if such a P-sequence
        does not exist
rotate $G$ so that the direction of edge $(v_0, v_1)$ is equal to 0
$vertex[0] \leftarrow v_0$; $vertex[1] \leftarrow v_1$
for $i \in [2, k-1]$ do
    if $vertex[p_i]$ does not have an outgoing edge in direction $d_i$ then return $null$;
    $vertex[i] \leftarrow$ vertex at direction $d_i$ from $vertex[p_i]$
if $\exists (p'_i, d'_i) \in F$ s.t. $vertex[p'_i]$ has an outgoing edge in direction $d'_i$ then return $null$;
return $\langle vertex[0], vertex[1], \dots, vertex[k-1] \rangle$
```

▶ **Example 12.** Let us consider the graph G and the pattern (P, F) of Figure 5. The canonical code of P is 00 02 11, and it may be computed either by $\text{BFS}_P(0,1)$ or by $\text{BFS}_{\overline{P}}(1,0)$ because P is symmetrical. Also, this pattern is isomorphic to two subgraphs of G. Hence, the four runs of seq that return a P-sequence corresponding to an occurrence of (P, F) are:

- $\text{seq}(G, a, b, 00\ 02\ 11, \{(0,1)\}) = \langle a, b, d, c \rangle$ corresponding to $0 = a, 1 = b, 2 = d, 3 = c$;
- $\text{seq}(\overline{G}, b, a, 00\ 02\ 11, \{(0,1)\}) = \langle b, a, c, d \rangle$ corresponding to $0 = b, 1 = a, 2 = c, 3 = c$;
- $\text{seq}(G, a, d, 00\ 02\ 11, \{(0,1)\}) = \langle a, d, f, g \rangle$ corresponding to $0 = a, 1 = d, 2 = f, 3 = g$;
- $\text{seq}(\overline{G}, d, a, 00\ 02\ 11, \{(0,1)\}) = \langle d, a, g, f \rangle$ corresponding to $0 = d, 1 = a, 2 = g, 3 = f$.

All other runs of seq return *null*.

In Figure 6, the graph G has only one P-sequence for pattern P (*i.e.*, $\text{seq}(G, 0, 1, cc(P), F) = \langle 0, 1, 2, 3 \rangle$). $\text{seq}(G, 2, 4, cc(P), F) = null$ because G already has a vertex at direction 4 from 2 when rotating G so that the direction of edge $(2, 4)$ is equal to 0.

Given a hexagon graph G and a P-sequence $s = \langle v_0, v_1, \dots, v_{k-1} \rangle$, we define an algorithm, called $\text{P-BFS}_G(s)$, which performs a BFS of G that starts from s, and which is obtained from Algorithm 1 by replacing lines 3 and 4 by the following lines:

```
display the canonical code of $G_{\downarrow s}$
let $q$ be an empty FIFO queue
for $i \in [0, k-1]$ do  set $num[v_i]$ to $i$, and add $v_i$ in $q$
initialise the counter $c$ to $k$
```

These lines ensure that the code returned by P-BFS starts with the canonical code of the pattern associated with s so that this pattern is included in the returned code by construction. The numbers assigned to the vertices of s correspond to those assigned by BFS when computing the canonical code of the pattern and we add the vertices of s in q to ensure that these vertices are treated first.

The P-canonical code of (G, P, F) is the smallest code obtained from any P-sequence.

▶ **Definition 13** (P-canonical code $(cc_{P,F}(G))$. *The P-canonical code of $G = (V, E)$ with respect to a pattern (P, F) is $cc_{P,F}(G) = \min\ \{P\text{-}BFS_G(s) : s \in S\} \cup \{P\text{-}BFS_{\overline{G}}(s) : s \in \overline{S}\}$ where*

- $S = \{seq(G, v_0, v_1, cc(P), F) : (v_0, v_1) \in E \wedge seq(G, v_0, v_1, c, F) \neq null\}$,
- $\overline{S} = \{seq(\overline{G}, v_0, v_1, cc(P), F) : (v_0, v_1) \in E \wedge seq(\overline{G}, v_0, v_1, c, F) \neq null\}$.

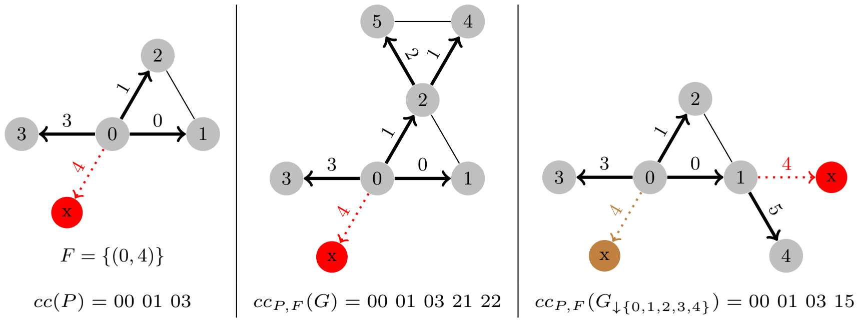

$F = \{(0,4)\}$
$cc(P) = 00\ 01\ 03$
$cc_{P,F}(G) = 00\ 01\ 03\ 21\ 22$
$cc_{P,F}(G_{\downarrow\{0,1,2,3,4\}}) = 00\ 01\ 03\ 15$
0
1
2
3
4
5
x

Figure 6 Example of P-canonical code with a non P-canonical prefix. Left: Pattern (P, F). Middle: Graph G. Right: $G_{\downarrow\{0,1,2,3,4\}}$. The prefix 00 01 03 21 of $cc_{P,F}(G)$ corresponds to $G_{\downarrow\{0,1,2,3,4\}}$ and it is not P-canonical as $cc_{P,F}(G_{\downarrow\{0,1,2,3,4\}})$ is smaller.

▶ **Example 14.** Let us consider the graph G and the pattern (P, F) of Figure 5. As seen in Example 12, there are four different candidate P-sequences s from which we may compute a code. We display in Figure 5 the runs of P-BFS from these four P-sequences.

3.2 Properties of P-canonical codes

The time complexity for computing $cc_{P,F}(G)$ when G has n vertices, P has k vertices (with $k < n$), and F has f couples is $\mathcal{O}(n(n + f))$:

- the canonical code of P is computed in $\mathcal{O}(k^2)$;
- Algorithm 2 is in $\mathcal{O}(k + f)$, and it is called $\mathcal{O}(n)$ times to build all P-sequences;
- P-BFS is in $\mathcal{O}(n)$ and it is called once per P-sequence, *i.e.*, $\mathcal{O}(n)$ times.

Properties of Theorem 5 are no longer valid as illustrated in Figure 5: the code returned by P-BFS$_G(\langle a, d, f, g\rangle)$ does not satisfy Property a as $p_3 = 1$ and $p_4 = 0$; the code returned by P-BFS$_{\overline{G}}(\langle d, a, g, f\rangle)$ does not satisfy Property b as $p_3 = p_4 = 1$ whereas $d_3 = 1$ and $d_4 = 0$. This comes from the fact that the canonical code of P is inserted at the beginning of the sequence, before starting the search. If a vertex of G corresponding to a pattern vertex has some extra edges outgoing from it in G but not in P, then the integer couple associated with this edge is added after $cc(P)$ and it may violate Properties a or b. However, we can easily show that these properties are satisfied in the second part of the code, corresponding to the vertices that do not correspond to pattern vertices, *i.e.*, given a code $p_1d_1 \ \ldots \ p_nd_n$ returned by P-BFS$_G(\langle v_0, \ldots, v_{k-1}\rangle)$, we have:
Property a': $\forall i \in [k, n-1]$, $p_i \leq p_{i+1}$;
Property b': $\forall i \in [k, n-1]$, $(p_i = p_{i+1}) \Rightarrow (d_i < d_{i+1})$.

When there are no forbidden vertices (*i.e.*, $F = \emptyset$), Theorem 6 is still valid, *i.e.*, every prefix of a canonical code is also canonical (the proof is similar to the proof of Theorem 6 and it is omitted due to space limits).

However, this theorem is no longer valid whenever $F \neq \emptyset$, *i.e.*, the prefix of a P-canonical code may not be P-canonical, as shown in the following example.

▶ **Example 15.** In Figure 6, the occurrence of (P, F) in G corresponds to $G_{\downarrow\{0,1,2,3\}}$. The prefix 00 01 03 21 of $cc_{P,F}(G)$ is not P-canonical: this prefix corresponds to $G_{\downarrow\{0,1,2,3,4\}}$ and (P, F) occurs twice in it (once in $G_{\downarrow\{0,1,2,3\}}$ with the forbidden vertex displayed in brown, and once in $G_{\downarrow\{0,1,2,4\}}$ with the forbidden vertex displayed in red). If we extend $G_{\downarrow\{0,1,2,3,4\}}$ by adding a vertex on the brown position, we obtain the graph G that contains only one occurrence of (P, F) and whose P-canonical code starts with a prefix greater than 00 01 03 15. If we extend $G_{\downarrow\{0,1,2,3,4\}}$ by adding a vertex on the red position, we obtain another graph (not isomorphic to G) that also contains only one occurrence of (P, F).

This example shows us that a non P-canonical code may be extended to a P-canonical code. This comes from the fact that a pattern may have more occurrences in the graph associated with a prefix of a code c than in the graph associated with c, due to forbidden vertices. However, when the pattern is symmetrical (including its forbidden vertices), we may discard some non P-canonical codes. More precisely, we define below the dominated codes that may be discarded because they cannot be extended to P-canonical codes.

▶ **Definition 16.** *(Dominated code) Let (P, F) be a symmetrical pattern, $G = (V, E)$ be a hexagon graph that contains at least one occurrence of (P, F), and $W \subset V$ be a subset of vertices such that $G_{\downarrow W}$ is isomorphic to P. Let C be the set of codes computed from a P-sequence $s = \langle v_0, v_1, \dots, v_k\rangle$ such that (v_0, v_1) is an edge of $G_{\downarrow W}$. A code $c \in C$ is dominated if there exists another code $c' \in C$ such that $c' < c$.*

▶ **Theorem 17.** *For each P-canonical code c and each prefix c' of c, c' is not dominated.*

Proof. This is a straightforward consequence of the fact that all P-sequences that start from an edge (v_0, v_1) of $G_{\downarrow W}$ correspond to different automorphisms of (P, F) and are interchangeable. ◀

▶ **Example 18.** In Figure 5, (P, F) is symmetrical. The dominated codes are P-BFS$_{\overline{G}}(\langle b, a, c, d\rangle)$ (when $W = \{a, b, c, d\}$) and P-BFS$_{\overline{G}}(\langle d, a, g, f\rangle)$ (when $W = \{a, d, f, g\}$. However, we cannot discard P-BFS$_G(\langle a, b, d, c\rangle)$ (though it is larger than P-BFS$_G(\langle a, d, f, g\rangle)$ as it may lead to a P-canonical code if a vertex is added on the forbidden position for $W = \{a, d, f, g\}$.

4 CP Models for Enumerating Hexagon Graphs

As hexagon graphs may be uniquely represented with canonical codes, we solve BGPs by enumerating canonical codes, and we propose to use CP to achieve this enumeration as this allows us to easily add constraints on the structures. In Section 4.1, we introduce a first CP model for enumerating consistent codes. In Section 4.2, we introduce a global constraint for ensuring that codes are canonical. In Section 4.3, we introduce a CP model for enumerating all structures that contain some given patterns.

4.1 CP Model for Enumerating Consistent Codes

We say that a code is consistent if there exist a hexagon graph G and an edge (v_0, v_1) of G such that BFS$_G(v_0, v_1)$ or BFS$_{\overline{G}}(v_0, v_1)$ return this code. Obviously, the code must satisfy Properties a and b listed in Theorem 5. However, this is not enough because we must also ensure that all vertices have different positions in the plane. To this aim, we associate 2D coordinates with vertices. We assume that all edges have a length equal to two and that vertex 0 has coordinates $(0, 0)$. For every other vertex i, its coordinates depend on the coordinates of its predecessor p_i and the direction d_i of edge (p_i, i): if the coordinates of p_i are (x, y) and $d_i = 0$ (resp. 1, 2, 3, 4, and 5), then vertex i is at coordinates $(x + 2, y)$ (resp. $(x + 1, y + \sqrt{3})$, $(x - 1, y + \sqrt{3})$, $(x - 2, y)$, $(x - 1, y - \sqrt{3})$, and $(x + 1, y - \sqrt{3})$).

▶ **Example 19.** Let us consider the code 00 02 24. Vertex 1 is at coordinates $(2, 0)$ (because the edge that reaches 1 from 0 has direction 0), vertex 2 is at coordinates $(1, \sqrt{3})$ (because the edge that reaches 2 from 0 has direction 1), and vertex 3 is at coordinates $(0,0)$ (because the edge that reaches 3 from 2 has direction 4). As vertices 0 and 3 have the same coordinates, this code is not consistent.

For each vertex $i \in [1,n]$, our CP model uses four integer variables x_i, y_i, p_i, d_i:

$\boldsymbol{x_i}$ is the abscissa of i, and its domain is $D(x_i) = [-2n, 2n]$;

$\boldsymbol{y_i}$ is the ordinate of i divided by $\sqrt{3}$, and its domain is $D(y_i) = [-n, n]$;

$\boldsymbol{p_i}$ is the vertex which has discovered i, and its domain is $D(p_i) = [0, i-1]$ ($p_i < i$ because i cannot be discovered by a vertex j which has not yet been discovered);

$\boldsymbol{d_i}$ is the direction of edge (p_i, i), and its domain is $D(d_i) = [0, 5]$.

For vertex 0, we define $x_0 = 0$, $y_0 = 0$, $p_0 = -1$, and $d_0 = -1$. The code associated with these variables is $p_1 d_1 \ \ldots \ p_n d_n$

To ensure the consistency of the generated codes, we post the following constraints:

$\boldsymbol{C_1}$: all vertices have different coordinates, *i.e.*, $\forall i, j \in [0,n], i \neq j \Rightarrow x_i \neq x_j \vee y_i \neq y_j$;

$\boldsymbol{C_2}$: Properties a and b (defined in Theorem 5) are satisfied, *i.e.*, $\forall i \in [1, n-1], p_i \leq p_{i+1} \wedge p_i = p_{i+1} \Rightarrow d_i < d_{i+1}$;

$\boldsymbol{C_3}$: For each edge (p_i, i), coordinates of i and p_i are consistent with direction d_i, *i.e.*, $\forall i \in [1,n], (d_i, x_i, x_{p_i}, y_i, y_{p_i}) \in T$ where T is the table defined as follows:

$$\begin{aligned}
T \;=\; & \{(0, x+2, x, y, y) : x \in [-2n, 2n-2], y \in [-n, n]\} \\
\cup \;& \{(1, x+1, x, y+1, y) : x \in [-2n, 2n-1], y \in [-n, n-1]\} \\
\cup \;& \{(2, x-1, x, y+1, y) : x \in [-2n+1, 2n], y \in [-n, n-1]\} \\
\cup \;& \{(3, x-2, x, y, y) : x \in [-2n+1, 2n], y \in [-n, n]\} \\
\cup \;& \{(4, x-1, x, y-1, y) : x \in [-2n+1, 2n], y \in [-n+1, n]\} \\
\cup \;& \{(5, x+1, x, y-1, y) : x \in [-2n, 2n-1], y \in [-n+1, n]\}.
\end{aligned}$$

4.2 Global Constraint for Ensuring Canonicity

We must ensure that codes are canonical to become invariant to rotations and symmetries. To this aim, we introduce a new global constraint defined below.

▶ **Definition 20.** *Given an integer value $n \geq 1$ and a tuple of four integer variables (p_i, d_i, x_i, y_i) for each $i \in [1,n]$, the constraint canonical$(\{(p_i, d_i, x_i, y_i) : i \in [1,n]\})$ is satisfied if the code $p_1 d_1 \ \ldots \ p_n d_n$ is canonical.*

To propagate this constraint, we maintain a vector *pat* such that, for each vertex i, *pat*$[i]$ is equal to the current pattern of i (with respect to the edges that have been added so far). We also maintain a matrix M such that for each vertex i and each direction $d \in [0,5]$, $M[i][d]$ is either equal to the vertex at direction d from i, if such a vertex exists, or to -1 otherwise. *pat* and M are updated each time a new couple (p_i, d_i) is instantiated. This may be done in constant time by exploiting vertex coordinates.

We trigger a failure whenever there is a vertex $i > 0$ such that *pat*$[i] <$ *pat*$[0]$, as we have seen in Section 2.2 that the pattern of i cannot be smaller than the pattern of 0 (according to the order of Definition 8).

Also, we trigger a failure whenever assigned variables define a code prefix which is not canonical, because Theorem 6 tells us that a non-canonical prefix cannot be extended to a canonical code. More precisely, when all variables that occur in a code prefix $p_1 d_1 \ \ldots \ p_k d_k$ with $k \leq n$ are assigned, we check that this prefix is canonical and, if it is not the case, we trigger a failure. To check whether a prefix is canonical or not, we try to build smaller codes by running BFS from other edges than $(0, 1)$. We limit the number of BFSs to be performed by exploiting patterns, as explained in Section 2.2.

Finally, we ensure that the degree of every vertex is upper bounded by a value g which depends on $pat[0]$: $g = 5$ when $pat[0] = P_5$; $g = 4$ when $pat[0] \in \{P_4, P_{3+1}, P_3, P_{2+2}\}$; $g = 3$ when $pat[0] \in \{P_{2+1a}, P_{2+1b}, P_2, P_{1+1+1}\}$; $g = 2$ when $pat[0] = P_{1+1}$; and $g = 1$ when $pat[0] = P_1$. When the degree of a vertex i reaches g, we remove i from the domain of every non-assigned variable p_k in order to prevent i from having more outgoing edges than g.

4.3 CP Model for Enumerating Graphs with a Given Pattern

Given a pattern (P, F) with m mandatory vertices, the CP model for enumerating all graphs with $n + 1 > m$ vertices that contain (P, F) uses four integer variables x_i, y_i, p_i, and d_i for each vertex $i \in [0, n]$, like in Section 4.1. However, for each pattern vertex $i \in [0, m - 1]$, we assign the variables x_i, y_i, p_i, and d_i according to the canonical code $cc(P)$.

▶ **Example 21.** For the pattern P displayed in Figure 6, we set $x_0 = y_0 = 0, p_1 = p_2 = p_3 = 0, d_1 = 0, d_2 = 1, d_3 = 3, x_1 = 2, y_1 = 0, x_2 = 1, y_2 = 1, x_3 = -2$, and $y_3 = 0$.

Like in Section 4.1, we post constraint C_1 to ensure that all vertices have different coordinates. We modify constraint C_2 as Properties of Theorem 5 are satisfied only for vertices that are not in the pattern, *i.e.*, $\forall i \in [m, n - 1], p_i \leq p_{i+1} \wedge p_i = p_{i+1} \Rightarrow d_i < d_{i+1}$. We post constraint C_3 but with a modified table T' to ensure that no vertex is at a forbidden position. More precisely, we compute the set X_F of coordinates of forbidden positions in F and define T' from table T as follows: $T' = \{(d, x, x', y, y') \in T : (x, y) \notin X_F \wedge (x', y') \notin X_F\}$.

Finally, we post a global constraint to ensure that the generated codes are P-canonical. This constraint is defined as follows.

▶ **Definition 22.** *Given the canonical code $cc(P)$ of a pattern P with m vertices, an integer value $n > m$ and a tuple of four integer variables (p_i, d_i, x_i, y_i) for each $i \in [1, n]$, the constraint P-canonical$_{cc(P)}(\{(p_i, d_i, x_i, y_i) : i \in [1, n]\})$ is satisfied if the code $p_1d_1 \ \ldots \ p_nd_n$ is P-canonical.*

To propagate this constraint, we cannot trigger a failure whenever assigned variables define a code prefix which is not P-canonical as we have seen in Section 3.2 that Theorem 6 is no longer valid. However, we exploit Theorem 17 to trigger a failure whenever the current prefix is dominated. Also when all p_i and d_i variables are assigned, we check that the code is P-canonical and trigger a failure whenever this is not the case.

Some BGPs involve enumerating graphs that contain two given patterns (P, F) and (P', F'). Without loss of generality, we assume that $\#P \geq \#P'$ where $\#G$ denotes the number of vertices of a graph G. In this case, we post the constraint *P-canonical*$_{cc(P)}$ to ensure that all generated codes contain pattern (P, F) by construction. To ensure that the graph G associated with the generated code also contains pattern (P', F'), we post a global constraint that ensures that $G_{\downarrow[k,\#G]}$ contains an occurrence of (P', F'), where $k = \#P$ whenever the two patterns must be disjoint, whereas $k = 0$ otherwise. This constraint is defined below.

▶ **Definition 23.** *Given a pattern (P, F), two integer values k and n such that $k+\#P \leq n$, and a tuple of two integer variables (p_i, d_i) for each $i \in [0, n]$, the constraint subgraph$_{P,k}(\{(p_i, d_i) : i \in [0, n]\})$ is satisfied if there exists $S \subseteq [k, n]$ such that $G_{\downarrow S}$ is isomorphic to P and there is no vertex at a forbidden position.*

We have implemented a very basic propagator for this constraint: it is not propagated during the search, and we only check that it is entailed when all variables have been assigned. This is done by searching for an edge (v_0, v_1) of $G_{\downarrow[k,n]}$ such that $\text{seq}(G_{\downarrow[k,n]}, v_0, v_1, cc(P'), F') \neq$ *null*. If we do not find such an edge, then the constraint is not satisfied; otherwise we have found an occurrence of (P', F') in $G_{\downarrow[k,n]}$.

5 Experimental Evaluation

In this section, we experimentally evaluate our approach on four different BGPs (see [2, 3, 4] for more details):

BGP1 enumerates hexagon graphs without single vertex holes (where a single vertex hole is a position without vertex which is surrounded by a cycle of 6 vertices);

BGP2 enumerates catacondensed graphs, *i.e.*, hexagon graphs without single vertex holes and without cliques of order 3;

BGP3 enumerates hexagon graphs without single vertex holes and that contain a given pattern (taken in the list of eight patterns used in [3] and recalled in Appendix A);

BGP4 enumerates hexagon graphs without single vertex holes and that contain two given patterns (taken in the list of eight patterns used in [3] and recalled in Appendix A) so that the two patterns are not overlapping. (In [3], a variant is considered where patterns may share vertices; we do not display results for this variant as the conclusions are very similar.)

For each problem, the number n of vertices is given, and we report results for $n \in [2, 10]$.

We consider the CP models described in Section 4, implemented in Choco [11], is available at `https://gitlab.inria.fr/xipeng/bgp`. To forbid single vertex holes, we exploit *pat* and M data structures. To forbid cliques of order 3, we ensure that, for each vertex i, $pat[i] \in \{P_{1+1+1}, P_{1+1a}, P_{1+1b}, P_1\}$ as all other patterns contain a clique of order 3.

Our approach, denoted CCode, is compared with the CP-based approach of [3], denoted BenzAI, and we used the Choco implementation available at `https://github.com/benzAI-team/BenzAI`. We do not compare our approach with the dedicated approach of [1] as it cannot enumerate hexagon graphs with holes (for BGP1, this approach finds less solutions than our approach), and it is not possible to add extra constraints in a declarative way so that it cannot be used to solve BGP2, BGP3, or BGP4 without modifying the code, or performing a post-processing to remove solutions that do not satisfy the extra constraints.

All experiments are carried out on an Intel Core Xeon E5-2623v3 of 3.0GHz×16 with 32GB of RAM.

In Table 1, we report CPU times of BenzAI and CCode on BGP1 and BGP2. For BGP1, BenzAI is faster when $n \leq 3$, but CCode is faster when $n \geq 4$ and when $n = 10$ it is 824 times as fast. For BGP2, BenzAI is faster when $n \leq 7$, but CCode is faster when $n \geq 8$ and when $n = 10$ it is 93 times as fast. In Table 2, we report CPU times of BenzAI and CCode on BGP3 and BGP4. CCode is always faster. When $n = 9$, it is 78 (resp. 136) times as fast as BenzAI for BGP3 (resp. BGP4). When $n = 10$, BenzAI is not able to solve any of the 8 (resp. 36) instances of BGP3 (resp. BGP4) within one hour whereas CCode never exceeds 71s.

6 Conclusion

We have introduced a new canonical code for representing hexagon graphs while being invariant to rotations and symmetries. This canonical code is the smallest code associated

Table 1 Results of BenzAI and CCode for BGP1 and BGP2. For each $n \in [2, 10]$ and each problem, we report the number of solutions (#sol) and the time in seconds of BenzAi and CCode.

	BGP1			BGP2		
	#sol	BenzAI	CCode	#sol	BenzAI	CCode
$n = 2$	1	0.00	0.01	1	0.01	0.01
$n = 3$	3	0.00	0.01	2	0.01	0.01
$n = 4$	7	0.05	0.03	5	0.01	0.02
$n = 5$	22	0.05	0.11	12	0.01	0.07
$n = 6$	81	0.14	0.22	36	0.07	0.15
$n = 7$	331	0.67	0.30	118	0.20	0.28
$n = 8$	1436	19.58	1.08	412	3.88	0.40
$n = 9$	6510	392.24	4.66	1492	19.78	2.21
$n = 10$	30129	22874.00	25.16	5587	732.75	7.90

Table 2 Results of BenzAi and CCode for BGP3 and BGP4. For each $n \in [2, 10]$ and each problem, we report the number of instances (#i) and the average, maximum and minimum time in seconds over these instances for BenzAI and CCode. We report '-' when all runs exceed 3600s.

n	BGP3							BGP4						
	#i	BenzAI			CCode			#i	BenzAI			CCode		
		avg	max	min	avg	max	min		avg	max	min	avg	max	min
2	2	0.02	0.02	0.02	0.01	0.01	0.00	0	—	—	—	—	—	—
3	6	0.02	0.03	0.01	0.00	0.01	0.00	0	—	—	—	—	—	—
4	7	0.03	0.04	0.02	0.00	0.02	0.00	3	0.07	0.16	0.02	0.01	0.03	0.00
5	8	0.05	0.07	0.01	0.01	0.07	0.00	11	0.06	0.20	0.04	0.01	0.06	0.00
6	8	0.38	0.43	0.23	0.03	0.15	0.00	23	1.41	2.51	0.93	0.02	0.16	0.01
7	8	0.95	1.06	0.79	0.13	0.35	0.01	29	3.78	5.42	2.30	0.10	0.50	0.01
8	8	20.33	23.84	13.94	0.73	1.61	0.04	34	65.06	83.21	50.28	0.49	1.69	0.03
9	8	306.30	399.25	75.05	3.94	10.52	0.23	35	400.51	574.15	226.87	2.94	10.85	0.20
10	8	-	-	-	34.60	69.75	2.20	36	-	-	-	24.19	71.03	1.61

with a BFS of the graph, and we have shown how to define codes by means of constraints, thus allowing us to generate all consistent codes with CP. We have shown that every prefix of a canonical code is canonical and this property allows us to trigger a failure whenever the current code prefix is not canonical.

We have also shown how to efficiently enumerate all codes that start with a given prefix, thus ensuring that a given pattern is included in the generated graphs. However, in this case it may happen that the prefix of a canonical code is no longer canonical, due to the fact that patterns may specify forbidden vertices. In this case, we exhibit a weaker property which allows us to filter some codes that cannot be extended to a canonical code.

We have experimentally compared our approach with the CP-based approach of [3], and we have shown that it scales much better, especially when there are a lot of solutions to enumerate.

As our approach is based on CP, new properties that must be satisfied by the generated graphs are defined in a declarative way by means of constraints. To this aim, the user has access to four variables for each vertex i, *i.e.*, p_i, d_i, x_i, and y_i. In some cases, it may be convenient to be able to add constraints on vertex patterns. This is the case, for example, for forbidding cliques of order 3 (in BGP2). In our first implementation, which is a proof of concept, the pattern of a vertex is maintained with a backtrackable data structure (using *IStateInt* Choco objects). As future work, we plan to introduce integer variables that represent patterns in order to allow the user to easily define new constraints on patterns.

We also plan to improve the propagation of the global constraint *subgraph* in order to detect some inconsistencies earlier: we could adapt Algorithm 2 (seq) in order to search for the largest sequence in polynomial time, and trigger a failure whenever the size of this sequence plus the number of non assigned vertices is smaller than n.

Finally, we would like to extend our approach to other kinds of graphs such as grid graphs (where vertices are associated with square faces instead of hexagonal faces), for example. We hope this will pave the way for new applications of CP such as, for example, applications that search for patterns in cellular automata as in [7].

References

1 Gunnar Brinkmann, Gilles Caporossi, and Pierre Hansen. A constructive enumeration of fusenes and benzenoids. *J. Algorithms*, 45(2):155–166, 2002.

2 Yannick Carissan, Denis Hagebaum-Reignier, Nicolas Prcovic, Cyril Terrioux, and Adrien Varet. Using constraint programming to generate benzenoid structures in theoretical chemistry. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming - 26th International Conference, CP*, volume 12333 of *Lecture Notes in Computer Science*, pages 690–706. Springer, 2020.

3 Yannick Carissan, Denis Hagebaum-Reignier, Nicolas Prcovic, Cyril Terrioux, and Adrien Varet. Exhaustive generation of benzenoid structures sharing common patterns. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP*, volume 210 of *LIPIcs*, pages 19:1–19:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

4 Yannick Carissan, Denis Hagebaum-Reignier, Nicolas Prcovic, Cyril Terrioux, and Adrien Varet. How constraint programming can help chemists to generate benzenoid structures and assess the local aromaticity of benzenoids. *Constraints An Int. J.*, 27(3):192–248, 2022.

5 S. J. Cyvin, J. Brunvoll, and B. N. Cyvin. Search for concealed non-kekulean benzenoids and coronoids. *Journal of Chemical Information and Computer Sciences*, 29(4):236–244, 1989.

6 Romain Deville, Élisa Fromont, Baptiste Jeudy, and Christine Solnon. Grima: A grid mining algorithm for bag-of-grid-based classification. In Antonio Robles-Kelly, Marco Loog,

Battista Biggio, Francisco Escolano, and Richard C. Wilson, editors, *Structural, Syntactic, and Statistical Pattern Recognition - Joint IAPR International Workshop, S+SSPR, Proceedings*, volume 10029 of *Lecture Notes in Computer Science*, pages 132–142, 2016.

7 Romain Deville, Élisa Fromont, Baptiste Jeudy, and Christine Solnon. Mining frequent patterns in 2d+t grid graphs for cellular automata analysis. In Pasquale Foggia, Cheng-Lin Liu, and Mario Vento, editors, *Graph-Based Representations in Pattern Recognition - 11th IAPR-TC-15 International Workshop, GbRPR 2017, Anacapri, Italy, May 16-18, 2017, Proceedings*, volume 10310 of *Lecture Notes in Computer Science*, pages 177–186, 2017.

8 Stéphane Gosselin, Guillaume Damiand, and Christine Solnon. Efficient search of combinatorial maps using signatures. *Theor. Comput. Sci.*, 412(15):1392–1405, 2011.

9 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014.

10 Siegfried Nijssen and Joost N. Kok. The gaston tool for frequent subgraph mining. In *Proceedings of the 2nd International Workshop on Graph-Based Tools, GraBaTs 2004, Rome, Italy, October 2, 2004*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 77–87. Elsevier, 2004. doi:10.1016/j.entcs.2004.12.039.

11 C. Prud'homme, J.-G. Fages, and X. Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. URL: http://www.choco-solver.org.

12 Xifeng Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724, 2002. doi:10.1109/ICDM.2002.1184038.

A Patterns used in BGP3 and BGP4

The eight patterns used in BGP3 and BGP4 are displayed below (mandatory and forbidden vertices are displayed in black and red, respectively).