3coSoKu and its Logic Programming Modeling [⋆]

Nicola Rizzo and Agostino Dovier

DMIF, University of Udine, Italy

Abstract. In this paper we analyze the physical puzzle IcoSoKu and we propose its generalization called 3coSoKu. We prove the NP-completeness of the latter. Then we encode it both in the constraint programming language MiniZinc and in the logic programming paradigm of Answer Set Programming. We use our encodings to experimentally prove the conjecture that every initial state for IcoSoKu admits a solution.

Keywords: IcoSoKu · 3coSoKu · NP-completeness · MiniZinc · ASP.

1 Introduction

IcoSoKu, shown in Figure 1, is a mechanical puzzle created by Andrea Mainini in 2009. The IcoSoku problem is presented in detail in Section 2. Since it admits

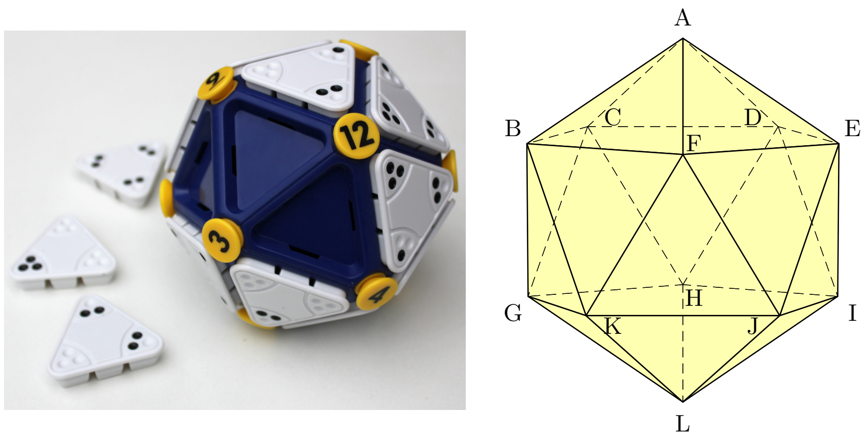

Fig. 1. IcoSoKu: a plastic blue icosahedron, 12 yellow numbered pegs, and 20 triangular tiles (left), and a possible naming of icosahedron vertices (right).

only a finite number of different initial states, from the point of view of decidability/complexity it is a finite game (and, hence, trivial). In this paper we present a

[⋆] Copyright © 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

generalization of the problem that admits an infinite number of different initial states, called `3coSoKu`, and we prove its NP-completeness in Section 3.

We then model the `3coSoKu` problem (and, as a side effect, IcoSoKu) using the constraint modeling language MiniZinc [13] (Section 4) and Answer Set Programming [3] (Section 5). The two chosen paradigms are the standard *de facto* for constraint programming (thanks to the MiniZinc challenge organized since 2008 [12]) and for solving combinatorial problems in logic programming (thanks to the ASP competition organized since 2007 [4]). In Section 6 we compare experimentally the two declarative approaches, along the lines of the empirical study conducted in [1]. Moreover, we use them in solving all possible instances of IcoSoKu, proving the conjecture that claims it to be possible. Some conclusions are drawn in Section 7. All the code and scripts used and described in the paper are available from http://clp.dimi.uniud.it/sw/.

2 The Problem

The IcoSoKu puzzle consists of (*i*) a plastic blue icosahedron, (*ii*) 12 yellow pegs with the numbers from 1 to 12 written on their tips, and (*iii*) 20 equilateral triangular tiles with from 0 up to 3 black dots near each vertex. The game is set up by placing all the pegs on the vertices of the icosahedron in an arbitrary way; the goal is to place all the tiles on the faces of the solid in a way such that the number of black dots surrounding each vertex is equal to the number of its peg. All the pieces of the puzzle must be used. Globally, there are $\sum_{i=1}^{12} i = 78$ black dots (see Figure 2). Each triangle can be rotated (for the sake of completeness we add that it cannot be flipped: the other face is not colored by dots). In [6,10] it is claimed that every setup of the pegs can be solved, but a proof of this conjecture is not given.

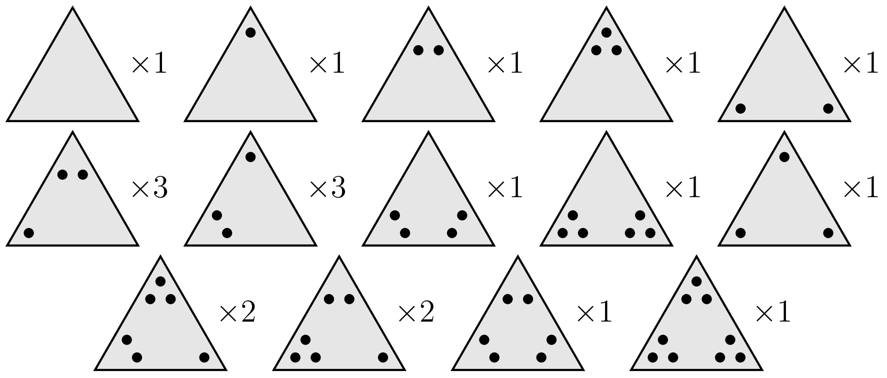

Fig. 2. The set of twenty tiles for IcoSoKu.

We can represent each triangle of Figure 2 with a triple of numbers, read in clockwise order written in non-decreasing lexicographical ordering:

$$(0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,1,1),\dots,(1,2,3),(1,3,2),(2,2,2),(3,3,3).$$

In the process of modeling this problem, each tile is assigned to an integer number in $\{1,\dots,20\}$. The possible rotations are encoded as 0 (0°), 1 (120°), and 2 (240°). Let us observe that the set does not include all the triples in $\{0,\dots,3\}^3$ (e.g., $(1,3,3),(2,3,3),\dots$). Moreover, we assign the letters from A to L to the vertices of the icosahedron, as seen in Figure 1 (right), and we refer to the faces using the three vertices that they involve in clockwise order. Thus the 20 faces are ABC, ACD, ADE, AEF, …, HLI, ILJ, JLK.

Given a face $f = v_0v_1v_2$, we use $\pi_i(f)$ to indicate its $(i+1)$-th element, with $i = 0,1,2$, that is $\pi_i(f) = v_i$. This is sufficient to describe the placement of a tile. For instance, assigning tile $(0,1,2)$ to face ABC with rotation 1 means to rotate the tile by 120°, obtaining tile $(2,0,1)$, and to place 2 black dots on the corner corresponding to vertex A, 0 dots on vertex B and 1 dot on vertex C.

2.1 Related work

An interesting benchmark for estimating the performance in solving instances of IcoSoKu is De Biasi's JavaScript online solver [5], implementing a backtracking algorithm that fills the faces of the icosahedron with the tiles, following a heuristic and deterministic search of the solution that prioritizes the most filled vertices. An experimental analysis of running times is given in Section 6.

IcoSoKu as a constraint programming problem has been recently studied by Liu et al. [9]: they observe that there are 24 different triangular tiles obtainable using $i \in \{0,1,2,3\}$ black dots, after accounting for rotations, whereas the tile distribution of IcoSoKu uses only 14 types of tiles (this leaves room for a generalization). The authors then proceed to pose some questions inspired by IcoSoKu but characterized by the use of pairwise distinct tiles (again, accounting for rotations), instead of the distribution the game ships with: using constraint programming, and in particular using the `alldifferent` and `table` constraints, they verify that each instance of IcoSoKu can be solved using the 24 possible tiles, having each time 4 unused tiles. In the process, they avoid checking all possible instances using symmetry breaking constraints that we also use in Section 6.2.

3 Generalizing IcoSoKu to `3coSoKu`

A variant of IcoSoKu can be stated for every polyhedron with regular faces, or faces of the same size and shape. All five Platonic solids (Figure 3) fit into this category, but infinite more polyhedra can be the gaming field, defining the vertices and their connections. We focus on all polyhedra with triangular faces. In this way, we generalize IcoSoKu by posing the constraint satisfaction problem we call `3coSoKu` and that we prove to be NP-complete. We try to stay close to the original problem by imposing that each tile must be used only once and that there must be as many tiles as there are faces. Temporarily, we overlook the practical constraint of having the faces and the tiles of the same size and shape: this is addressed at the end of the Section.

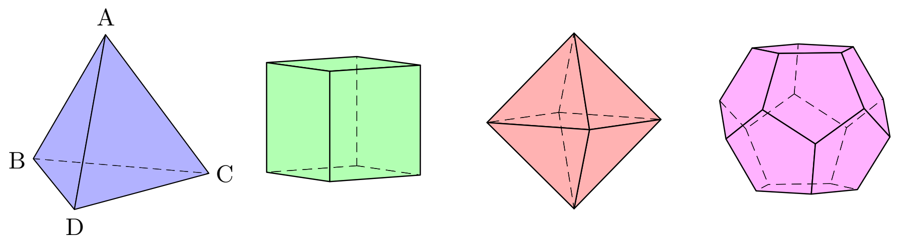

Fig. 3. The other Platonic solids can be the playing field of the game.

If $\mathcal{P}$ is a polyhedron with triangular faces, we refer to $V_{\mathcal{P}}$ as the set of the vertices of $\mathcal{P}$ and $F_{\mathcal{P}}$ as the set of the triples of vertices describing the faces of $\mathcal{P}$ ($F_{\mathcal{P}} \subseteq V_{\mathcal{P}}^3$). Given $v \in V_{\mathcal{P}}$, we also denote by $F_{\mathcal{P}}^v \subseteq F_{\mathcal{P}}$ the set of faces containing the vertex v. For example, the tetrahedron $\mathcal{T}$ in Figure 3 (left) is described by $V_{\mathcal{T}} = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$ and $F_{\mathcal{T}} = \{\mathrm{ABC}, \mathrm{ACD}, \mathrm{ADB}, \mathrm{BDC}\}$, with $F_{\mathcal{T}}^A = \{\mathrm{ABC}, \mathrm{ACD}, \mathrm{ADB}\}$.

The setup for a game of `3coSoKu` consists of the polyhedron, the capacities assigned to the vertices (the numbers on the yellow pegs of IcoSoKu) and the tiles containing triples of weights (number of black dots).

Definition 1 (`3coSoKu` instance). *Let $\mathcal{P}$ be a polyhedron with triangular faces. An instance* of `3coSoKu` *is a tuple $(\mathcal{P}, c, T)$ where*

- $c \colon V_{\mathcal{P}} \to \mathbb{N}$ *is a* (capacity) map *that assigns each vertex to an integer and*
- $T = \{\!|t_1, \ldots, t_n|\!\}$ *is a multiset of* tiles*, where each element $t_i \in \mathbb{N}^3$ is a triple of* weights *and $n = |F_{\mathcal{P}}|$.*

With $\mathcal{T}$ in Figure 3 (left), an instance of `3coSoKu` is $(\mathcal{T}, c, T)$ where: $c(\mathrm{A}) = 1, c(\mathrm{B}) = 2, c(\mathrm{C}) = 3, c(\mathrm{D}) = 4$ and $T = \{\!|(0,0,1), (0,0,2), (0,0,3), (1,1,2)|\!\}$.

Definition 2 (`3coSoKu` problem). *Given an instance $(\mathcal{P}, c, T)$ of* `3coSoKu`*, an* arrangement *of the tiles in T to the faces of $\mathcal{P}$ is a pair (ρ, σ) such that:*

- $\rho \colon T \to \{0, 1, 2\}$ *defines the rotation of each tile and*
- $\sigma \colon T \to F_{\mathcal{P}}$ *is a bijection assigning tiles to faces.*

An arrangement (ρ, σ) is a solution *of $(\mathcal{P}, c, T)$ if*

$$\sum_{\substack{t \in T,\ i \in \{0,1,2\}: \\ \pi_{(i+\rho(t)) \bmod 3}(\sigma(t)) = v}} \pi_i(t) = c(v) \qquad \forall v \in V_{\mathcal{P}}. \tag{1}$$

The `3coSoKu` *problem is the problem of finding a solution of $(\mathcal{P}, c, T)$.*

For each vertex v, equation (1) takes all the weights that end up near v after the rotation and placement of the tiles, using the modulo operation, and constraints their sum to be equal to the capacity of v. We can reformulate (1) into a handier equation that directly involves $F_{\mathcal{P}}^{v}$ instead of the tiles that are placed there (the latter is defined by the input while the former depends on the arrangement considered): let $\tau = \sigma^{-1}$; then the constraints for the `3coSoKu` problem are equivalent to

$$\sum_{\substack{f \in F_{\mathcal{P}},\, i \in \{0,1,2\}: \\ \pi_i(f) = v}} \pi_{(i-\rho(\tau(f))) \bmod 3}\big(\tau(f)\big) = c(v) \qquad \forall v \in V_{\mathcal{P}}. \tag{2}$$

Note how the rotation must be applied backwards in order to get the weight placed on the i-th vertex of face f, whereas in (1) the rotation is applied forward in order to find where the weights are placed. For example, if the polyhedron $\mathcal{P}$ is the tetrahedron $\mathcal{T}$ in Figure 3 (left), then the constraints defined by (2) become

$$\begin{aligned}
&\pi_{(-\rho(\tau(\mathrm{ABC}))) \bmod 3}\big(\tau(\mathrm{ABC})\big) + \pi_{(-\rho(\tau(\mathrm{ACD}))) \bmod 3}\big(\tau(\mathrm{ACD})\big) + \\
&\qquad\qquad \pi_{(-\rho(\tau(\mathrm{ADB}))) \bmod 3}\big(\tau(\mathrm{ADB})\big) = c(A) \\
&\pi_{(1-\rho(\tau(\mathrm{ABC}))) \bmod 3}\big(\tau(\mathrm{ABC})\big) + \pi_{(2-\rho(\tau(\mathrm{ADB}))) \bmod 3}\big(\tau(\mathrm{ADB})\big) + \\
&\qquad\qquad \pi_{(-\rho(\tau(\mathrm{BDC}))) \bmod 3}\big(\tau(\mathrm{BDC})\big) = c(B) \\
&\pi_{(2-\rho(\tau(\mathrm{ABC}))) \bmod 3}\big(\tau(\mathrm{ABC})\big) + \pi_{(1-\rho(\tau(\mathrm{ACD}))) \bmod 3}\big(\tau(\mathrm{ACD})\big) + \\
&\qquad\qquad \pi_{(2-\rho(\tau(\mathrm{BDC}))) \bmod 3}\big(\tau(\mathrm{BDC})\big) = c(C) \\
&\pi_{(2-\rho(\tau(\mathrm{ACD}))) \bmod 3}\big(\tau(\mathrm{ACD})\big) + \pi_{(1-\rho(\tau(\mathrm{ADB}))) \bmod 3}\big(\tau(\mathrm{ADB})\big) + \\
&\qquad\qquad \pi_{(1-\rho(\tau(\mathrm{BDC}))) \bmod 3}\big(\tau(\mathrm{BDC})\big) = c(D)
\end{aligned}$$

since the couples $f \in F_{\mathcal{T}}$, $i \in \{0,1,2\}$ such that $\pi_i(f) = \mathrm{A}$ are $(\mathrm{ABC}, 0)$, $(\mathrm{ACD}, 0)$ and $(\mathrm{ADB}, 0)$, the couples such that $\pi_i(f) = \mathrm{B}$ are $(\mathrm{ABC}, 1)$, $(\mathrm{ADB}, 2)$ and $(\mathrm{BDC}, 0)$, the couples such that $\pi_i(f) = \mathrm{C}$ are $(\mathrm{ABC}, 2)$, $(\mathrm{ACD}, 1)$ and $(\mathrm{BDC}, 2)$, and the couples such that $\pi_i(f) = \mathrm{D}$ are $(\mathrm{ACD}, 2)$, $(\mathrm{ADB}, 1)$ and $(\mathrm{BDC}, 1)$.

3.1 NP-completeness

We study the complexity of `3coSoKu`, proving that it is NP-complete. NP membership is immediate: given an instance of `3coSoKu` and an arrangement of its tiles to its faces, checking if the arrangement satisfies constraint (2) requires $O(n)$ arithmetic operations. To prove NP-hardness we will reduce one of Karp's original 21 NP-complete problems, the *partition problem*, (see [8, p. 97] and [2, pp. 60-61, 223]):

Input: A finite set A and an assignment $s(a) \in \mathbb{Z}^+$ for each $a \in A$.
Problem: Establish whether there is a subset $B \subseteq A$ such that

$$\sum_{a \in B} s(a) = \sum_{a \in A \setminus B} s(a).$$

Theorem 1. `3coSoKu` *is NP-complete.*

Proof. We already established that `3coSoKu` is in NP. To prove its NP-hardness we reduce Partition to `3coSoKu` showing how to transform its generic instance $A = \{a_1, \dots, a_n\}$ with $s(a_i) = s_i$ for $i \in \{1, \dots, n\}$ into an instance of `3coSoKu` $\big(\mathcal{P}, c, T\big)$, that we now describe, such that the latter admits a solution if and only if the former has a solution:

1. $\mathcal{P}$ is a bipyramid with $2n$ faces composed by merging two n-gonal pyramids base to base, such that the two apices correspond to sets B and $A \setminus B$ of a possible solution and the other $2n-2$ vertices are dummy vertices, as seen in Figure 4 (left);
2. c assigns capacity $\sum_{i=1}^{n} s_i/2$ to apices B and $A \setminus B$ and capacity 0 to all other vertices;
3. T is a multiset of $2n$ tiles made of n empty tiles $(0, 0, 0)$ and tiles $(s_i, 0, 0)$ for $i \in \{1, \dots, n\}$, as seen in Figure 4 (right).

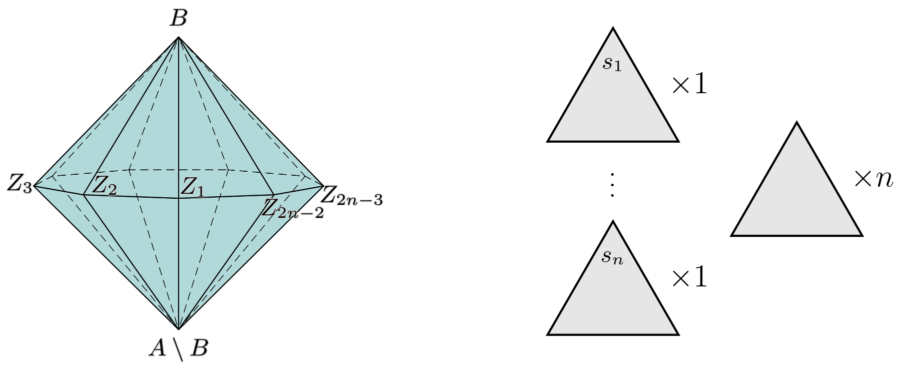

Fig. 4. The corresponding instance $(\mathcal{P}, c, T)$ of `3coSoKu` is such that $\mathcal{P}$ a bipyramid with $2n$ faces (left) and T has n empty tiles and tiles $(s_i, 0, 0)$ for $i \in \{1, \dots n\}$ (right).

It is easy to see that the Partition instance admits a solution B corresponding to sizes $q_1, \dots, q_k$, with $k \in \{1, \dots, n-1\}$ if and only if there is an arrangement (ρ, σ) that satisfies Equations (2), placing all the weights corresponding to sizes q_1, …, q_k near vertex B, placing the rest of the positive weights near $A \setminus B$ and arbitrarily placing all remaining $(0, 0, 0)$ tiles. The n tiles of cost zero are needed to compensate the difference of cardinality between B and $A \setminus B$. The reduction can clearly be carried on in $O\big(\log(\sum_{i=1}^{n} \log s_i)\big)$ space. □

We can now discuss more deeply the definition of `3coSoKu`, because it does not specify polyhedron $\mathcal{P}$ to be well-founded and constructible in three-dimensional space, especially so with equilateral triangles as faces. Some considerations:

– each instance defines linear constraints that are well defined even if $\mathcal{P}$ is not;
– in the reduction, the rotation of the tiles is not exploited;
– the constructed bypiramid is always a well defined polyhedron with faces of the same size and shape;
– Theorem 1 implicitly proves that the subset of instances characterized by strictly-convex triangular polyhedra (the bipyramid constructed has this features) is sufficient to make it NP-complete.

For these reasons, imposing constraints over the points above does not change the complexity of the problem. This reasoning leaves out general deltahedra, i.e. polyhedra with equal regular triangles as faces, to suffice for NP-completeness, since the only ones that are bipyramids are the triangular bipyramid, the octahedron and the pentagonal bipyramid [7]. So the NP-completeness of `3coSoKu` restricted to deltahedra remains an open problem.

4 The MiniZinc Encoding

In this section we present the MiniZinc encoding of `3coSoKu`. First, we need to define its instance $(\mathcal{P}, c, T)$. The model of $\mathcal{P}$ (see Listing 1.1) consists of:

– the integers `m` and `n`, the number of vertices and faces of $\mathcal{P}$;
– two sets `VERTEX` and `FACE` that contain the vertices and faces of $\mathcal{P}$;
– a matrix `vrtx` with n rows, one for each face of $\mathcal{P}$, and 3 columns such that `vrtx[f,j]`, `j` $\in \{0, 1, 2\}$ are the vertices of face `f` in clockwise order.

The capacities c and the tiles T are modeled by array `cap` and matrix `weight` (see for instance Listing 1.2). The solution (ρ, σ) is modeled by array `tile`, that

```
int: m = 12;
int: n = 20;
enum VERTEX = {A, B, C, D, E, F, G, H, I, J, K, L};
enum FACE = {ABC, ACD, ADE, AEF, AFB, BFK, BKG,
             BGC, CGH, CHD, DIE, DHI, EIJ, EJF,
             FJK, GKL, GLH, HLI, ILJ, JLK};
array[FACE, 0..2] of VERTEX: vrtx = array2d(FACE, 0..2,
  [A, B, C,           A, C, D,          A, D, E,
                       ...
   H, L, I,           I, L, J,          J, L, K]);
array[FACE] of var 1..n: tile;
array[FACE] of var 0..2: rot;
```

Listing 1.1. Setting the icosahedron as playing field of the MiniZinc model.

assigns each face to a tile, and `rot`, that assigns each face to the rotation (0, 1, or 2) of the tile placed on it, as shown at the end of Listing 1.1. `tile` models

```
array[VERTEX] of int: cap = [1,11,5,10,6,7,9,2,3,8,4,12];
array[1..n, 0..2] of int: weight = array2d(1..n, 0..2,
  [0,0,0, 0,0,1, 0,0,2, 0,0,3, 0,1,1,
   0,1,2, 0,1,2, 0,1,2, 0,2,1, 0,2,1,
   0,2,1, 0,2,2, 0,3,3, 1,1,1, 1,2,3,
   1,2,3, 3,2,1, 3,2,1, 2,2,2, 3,3,3]);
```

Listing 1.2. Instance of IcoSoKu for the MiniZinc model.

```
function var int: vertex_sum(VERTEX: v) =
  sum (f in FACE, r in 0..2 where vrtx[f, r] == v)
      (weight[tile[f], [1,2,0,1,2][3 + r - rot[f]]]);
      % equivalent to  (3 + r - rot[f]) mod 3
constraint alldifferent(tile);
constraint forall (v in VERTEX) (vertex_sum(v) == cap[v]);
```

Listing 1.3. Constraints of the MiniZinc model.

```
Put tile (0, 2, 1) on face corresponding to pegs 1, 11, 5 (face ABC).
Put tile (0, 0, 2) on face corresponding to pegs 1, 5, 10 (face ACD).
Put tile (1, 2, 0) on face corresponding to pegs 1, 10, 6 (face ADE).
...
Put tile (0, 2, 1) on face corresponding to pegs 2, 12, 3 (face HLI).
Put tile (1, 3, 2) on face corresponding to pegs 3, 12, 8 (face ILJ).
Put tile (1, 3, 2) on face corresponding to pegs 8, 12, 4 (face JLK).
----------
```

Listing 1.4. Example output of the MiniZinc encoding for IcoSoKu.

$\tau = \sigma^{-1}$ instead of σ, since during the development of the encoding we have heuristically observed that this implicitly defines a faster search strategy. For the solution to be correct, all the tiles must be assigned to different faces, which is guaranteed using the predicate `alldifferent`, and the weights assigned to each vertex must add up to its capacity. The constraints corresponding to Equation (2) are easily enforced using the matrix `vrtx` and the modulo operation, as in Listing 1.3.[1] Symmetry breaking constraints that impose tiles of the form (x, x, x), with $x \in \{0, 1, 2, 3\}$, are not rotated and that impose a fixed order between duplicated tiles are also added (and are not shown in this paper).

Finally, Listing 1.4 shows the output of the encoding for an instance of IcoSoKu, obtained using the post-processing functions we have written to give instructions to the solution. We will explore in the future other ways to visualize the solutions.

5 The ASP Encoding

The input is similar to the input of the MiniZinc solver: the variant played consists of constants `m` and `n`, the vertices and faces of $\mathcal{P}$ given as input facts of predicates `vertex/1` and `face/1` and the projection of the vertices in their component is given in input extensionally with predicate `vrtx/3` (see Listing 1.5); capacities and tiles are given by the predicate `cap/2` and `weight/3`, such that $c(v) = i$ corresponds to fact `cap(v, i)` and the i-th tile being (x, y, z) corresponds to facts `weight(i, 0, x)`, `weight(i, 1, y)` and `weight(i, 2, z)`. The encoding of an instance of IcoSoKu is shown in Listing 1.6.

```
#const m = 12.
#const n = 20.
vertex(a; b; c; d; e; f; g; h; i; j; k; l).
face(abc; acd; ade; aef; afb; ... ; hli; ilj; jlk).
vrtx(a, abc, 0;  b, abc, 1;  c, abc, 2;
     a, acd, 0;  c, acd, 1;  d, acd, 2;
         ....
     h, hli, 0;  l, hli, 1;  i, hli, 2;
     i, ilj, 0;  l, ilj, 1;  j, ilj, 2;
     j, jlk, 0;  l, jlk, 1;  k, jlk, 2).
```

Listing 1.5. Setting the icosahedron as playing field of the ASP model.

The solution is modeled by functions `assign/2` and `rotate/2`, that indicate respectively which tile to put on each face and the rotation of each tile. To constraint the solution to be correct (see Listing 1.7):

[1] A reformulation without using the `mod` operator as suggested by a reviewer—that we would like to thank—proved to be an improvement on harder instances of IcoSoKu using Gecode's standard search strategy.

```
cap(a,1; b,11; c,5; d,10; ... ; j,8; k,4; l,12).
weight(1,  0, 0;  1,  1, 0;  1,  2, 0;
       2,  0, 0;  2,  1, 0;  2,  2, 1;
       3,  0, 0;  3,  1, 0;  3,  2, 2;
                      ...
       18, 0, 3;  18, 1, 2;  18, 2, 1;
       19, 0, 2;  19, 1, 2;  19, 2, 2;
       20, 0, 3;  20, 1, 3;  20, 2, 3).
```

Listing 1.6. Instance of IcoSoKu for the ASP model.

- we impose `assign` to be a bijection of the tiles to the faces;
- we impose each tile to have one and only one rotation;
- we use aggregate function `#sum` to discard all models in which the capacities of the vertices are not fulfilled exactly.

```
tile(1..n).
rotation(0..2).
1 { assign(T, F): face(F) } 1 :- tile(T).
1 { assign(T, F): tile(T) } 1 :- face(F).
1 { rotate(T, R) : rotation(R) } 1 :- tile(T).
:- #sum{ P,F : vrtx(V, F, A), assign(T, F), rotate(T, R),
    weight(T, (A - R + 3) \ 3, P) } != C, cap(V, C).
```

Listing 1.7. Rules to constraint the solution of the ASP model.

Also in this case we add symmetry breaking constraints as in the MiniZinc encoding (not shown in this paper). For the output to be readable, an auxiliary predicate `put/9` has been written: its values, to be read in groups of three, indicate the tile, the face where to put the tile and the corresponding capacities.[2] An example of the output for IcoSoKu is shown in Listing 1.8.

6 Tests and Experiments

We report on some tests on `3coSoKu`, available in the code repository http://clp.dimi.uniud.it/sw/, we have written in Bash. The comparative tests have been executed on a single-thread of an Intel Core i5-7400 @ 3.50 GHz running Arch Linux. The versions of the main tools we used are shown in Figure 5.

[2] Predicate `put/9` affects the grouding size, adding $O(n^2)$ statements: its removal must be considered when solving `3coSoKu` instances with a high number of tiles and faces.

```
clingo version 5.4.0
Reading from stdin
Solving...
Answer: 1
cap(a,1) cap(b,11) cap(c,5) cap(d,10) cap(e,6) cap(f,7) cap(g,9) cap(h,2)
     cap(i,3) cap(j,8) cap(k,4) cap(l,12) put(0,3,0,a,b,c,1,11,5)
     put(0,1,2,a,c,d,1,5,10) put(0,1,2,a,d,e,1,10,6) ...
     put(0,2,0,h,l,i,2,12,3) put(0,2,2,i,l,j,3,12,8) put(3,2,1,j,l,k,8,12,4)
SATISFIABLE
Models       : 1+
Calls        : 1
Time         : 0.102s (Solving: 0.04s 1st Model: 0.04s Unsat: 0.00s)
CPU Time     : 0.100s
```

Listing 1.8. Example output of the ASP model for IcoSoKu.

Software	Description	Version
MiniZinc Distribution	MiniZinc compiler and FlatZinc solvers	2.4.3
clingo	Answer Set System	5.4.0
Node.js	JavaScript run-time environment (v8 engine)	14.5.0
OpenSSL	Algorithms for test generation	1.1.1g
GNU **parallel**	Shell tool to execute jobs in parallel	20200322

Fig. 5. Software used in testing.

6.1 Comparative performance tests for IcoSoKu

To measure the performance of the solvers we have written a script that, starting from a seed, generates a batch of 100 instances of IcoSoKu and measures the performance of each solver on the whole batch. The generator uses `openssl` as `shuf`'s pseudo-random source.

The solvers we compare are: for the MiniZinc encoding[3], Gecode and Chuffed; for the ASP encoding, the clingo ASP system; De Biasi's JavaScript solver. We have not explored any optimization parameters of the models, but after some preliminary experiments we observed that the most successful search strategy of the Gecode solver for MiniZinc is a randomized search plus a restart after a low and constant number of nodes visited, implemented in Listing 1.9.

The times reported have been gathered, respectively, from `minizinc`'s own "Overall time" in its statistics (option `-s`), from `clingo`'s "CPU Time" already available, and by parametrizing function `icosolve` of the JavaScript solver to accept the capacities in input (taking care of the difference in the naming convention for the vertices) and with two invocations of `new Date()`.

[3] In the MiniZinc tests, for each instance of the batch the model is compiled to FlatZinc and is then fed into a solver. Looking at the different outputs of the first step, we have noted that the difference from instance to instance is only in the 12 FlatZinc instructions specifying the capacities (they involve constraint `int_lin_eq`). This means that the compilation to FlatZinc can be skipped by modifying the FlatZinc output of one instance. We have not exploited this trick to be fair to the other solvers.

The results are shown in Figures 6 and 7, with the relative mean value drawn horizontally. The worst times are:

– 15.14 seconds for the plain MiniZinc model using Gecode;
– 0.45 seconds for the MiniZinc model using Gecode with the randomized search of Listing 1.9;
– 0.23 seconds for the MiniZinc model using Chuffed;
– 0.09 seconds for the ASP model;
– 47.61 seconds for the JavaScript solver.

The best model for IcoSoKu is the ASP one, but Chuffed and Gecode's randomized search are valid alternatives. Gecode's standard search is similar to the preexisting JavaScript solver, in the sense that in average it is faster but solving some instances takes a lot of time because the solvers explore a large subtree of the search space that has no solution.

6.2 Verifying IcoSoKu's solvability

We have challenged our encodings (and the constraint/ASP solvers used) to verify the conjecture that every possible instance of the commercial version of IcoSoKu admits a solution (i.e. fixing the icosahedron as playing field and using the set of tiles of Figure 2). The naive search space consisting of all the permutations of $\{1, \ldots, 20\}$ has been first pruned removing symmetric solutions:

1. W.l.o.g., we impose the capacity of vertex A to be equal to 1.
2. Once the capacity of vertex A is fixed, the icosahedron can still be rotated on its A–L vertical axis by any multiple of 72°. Thus, w.l.o.g. we can impose the capacity of vertex B to be less than those of vertices C, D, E, and F. This leaves for B any value in $\{2, \ldots, 8\}$.
3. We can exploit one last symmetry, involving both the icosahedron and the tile configuration of IcoSoKu. By flipping the icosahedron horizontally with the plane of reflection that goes through A, L and B, as shown in Figure 8, we can obtain a mirrored icosahedron imposable over the original. For each instance defined by points 1. and 2. we can flip their capacities instead of the vertices, obtaining a different instance that still fits the two constraints, so we can split these instances into mutually symmetric pairs. Thus for each instance there is a symmetrical one and also for each solution there is a symmetrical one: in fact, by flipping the whole tile configuration of IcoSoKu we get the same tile configuration, because the tiles with three different numbers of black dots all have their symmetric counterpart (see Figure 2). This allows us to divide by two the number of possible inputs.

Thus, with 7 possible capacities for vertex B and the whole set divided by two we have roughly 4M instances to check:

$$\frac{1}{2}\left(10! + \frac{9! \cdot 6!}{5!} + \frac{8! \cdot 6!}{4!} + \frac{7! \cdot 6!}{3!} + \frac{6! \cdot 6!}{2!} + 5! \cdot 6! + 4! \cdot 6!\right) = 3\,991\,680$$

```
solve :: int_search(tile, first_fail, indomain_random)
      :: restart_constant(100)
      satisfy;
```

Listing 1.9. Randomized search for the MiniZinc model.

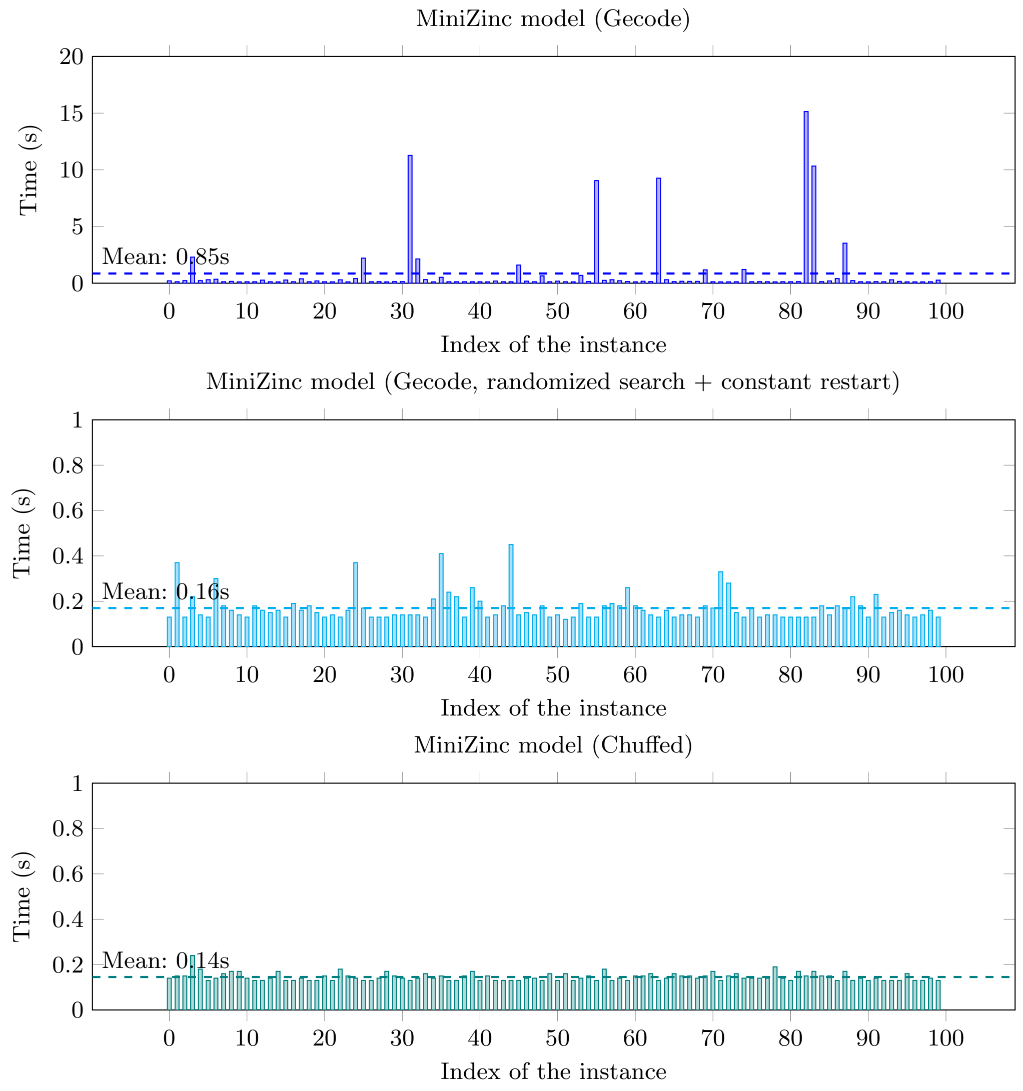

Fig. 6. MiniZinc performance tests on instances of IcoSoKu.

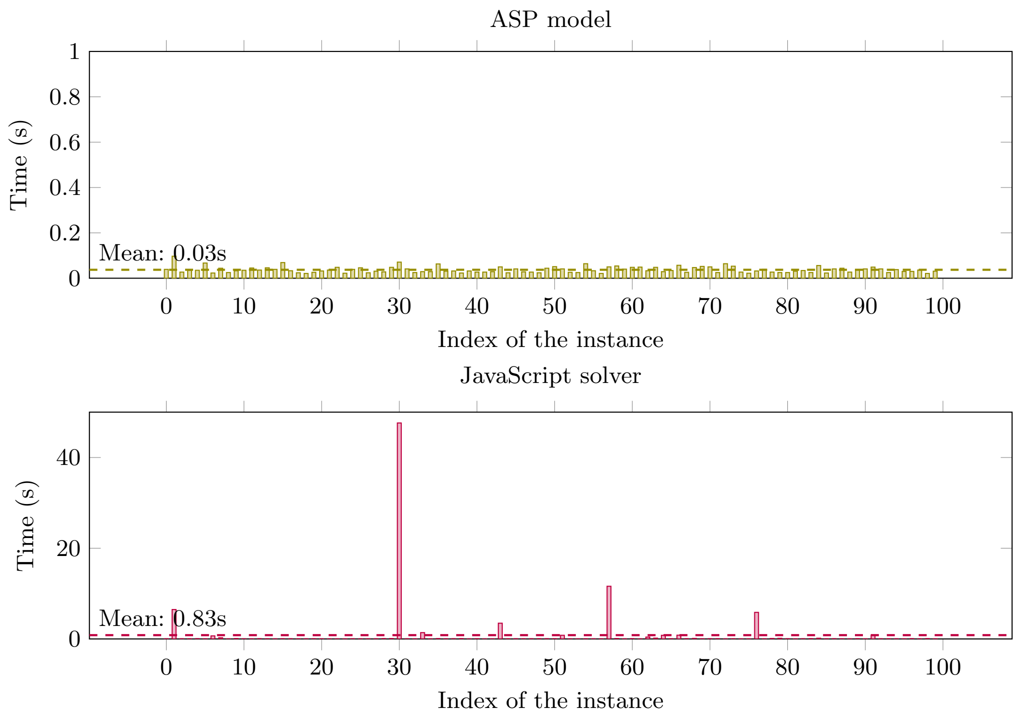

Fig. 7. ASP (top) and JavaScript (bottom) performance test on instances of IcoSoKu.

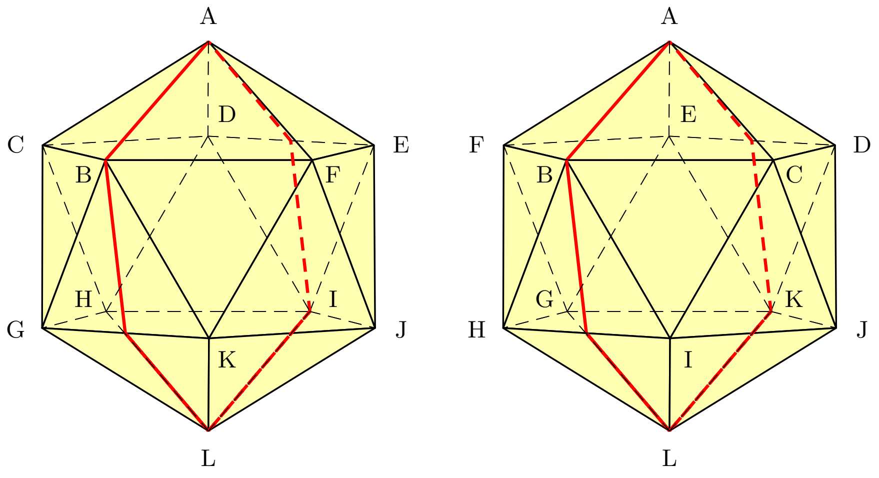

Fig. 8. The icosahedron (left) intersected with the plane through A, B and L (in red) and its mirrored version (right): this symmetry can be exploited with the symmetries of the tile configuration.

We have written a small program to generate the corresponding instances and using GNU **parallel** [14] we have parallelized our ASP model[4]: using four threads, in 15 hours, 57 minutes and 50 seconds all instances were checked, empirically verifying the claim of the author of the game. We also repeated the benchmark using **parallel**'s option `--joblog` to log the solving time of each instance. The process took five extra minutes, with 0.23 seconds the worst solving time and 0.05 seconds the average solving time. 98% of instances took less than 0.1 seconds and 61% of instances took less time than the average. During development, this test has also been executed successfully with our MiniZinc models, using Chuffed and using Gecode's randomized search.

7 Conclusions

We have defined `3coSoKu`, a generalized version of IcoSoKu and show its NP-completeness, even when imposing that the playing field is well formed and that the faces are of the same size and shape. Some questions remain unanswered: is `3coSoKu` still NP-complete if we impose the playing field to be a deltahedron, i.e. a polyhedron with equilater triangles as faces? Partition admits a pseudo-polynomial time algorithm (see [2, p. 223]); is it the case for `3coSoKu` as well?

From experimental testing on instances of IcoSoKu, the best approaches we have found to solve instances of IcoSoKu use our ASP model and our MiniZinc model with solver Chuffed: our interpretation is that the nogood learning of the former and the lazy clause generation of the latter is an effective tool to prune the search tree of `3coSoKu`. Gecode's randomized search plus constant restart is also valid, probably because there is a high number of solutions for each instance of IcoSoKu, and could inspire a practical strategy for the game. Counting the number of solutions with our solvers proved to be a difficult task because of the size of the search tree. Exploring approximately 2% of the search tree for the first instance of the batch of tests, we found more than 200 million different solutions (with symmetry breaking constraints).

A profound combinatorial reason for the existance of a solution to every instance of IcoSoKu has not been found (not looked for, actually), but using our models and some symmetry breaking considerations, we managed to solve every possible instance in 16 hours with common hardware, proving the conjecture.

References

1. Dovier A., Formisano A., Pontelli E.: An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. J. Exp. Theor. Artif. Intell. 21(2): 79-121 (2009)
2. Garey M. R., Johnson D. S.: Computers and Intractability, a Guide to the Theory of NP-Completeness. W. H. Freeman and Company (1979)

[4] In order to shave off as much time as possible, we removed print predicate `put/9` and suppressed the output of the solution entirely, relying on `clingo`'s output to know if there exists a solution or not.

3. Gebser M., Kaminski R., Kaufmann B., Lindauer M., Ostrowski M., Romero J., Schaub T., Thiele S., Wanko P.: Potassco User Guide. 2nd edn, version 2.2.0. University of Potsdam (2019)
4. Gebser M, et al.: The First Answer Set Programming System Competition. Proc of LPNMR, LNCS 4483, pp 3–17 (2007)
5. IcoSoKu online solver page, http://www.nearly42.org/games/icosoku-solver/. Last accessed 30 March 2020
6. IcoSoKu puzzle product page, https://www.recenttoys.com/recent-toys-icosoku-puzzle/. Last accessed 30 March 2020
7. Johnson N.: Convex Polyhedra with Regular Faces. Canadian Journal of Mathematics, 18, 169–200 (1966). https://doi.org/10.4153/CJM-1966-021-8
8. Karp R. M.: Reducibility among Combinatorial Problems. In: Complexity of computer computations, pp. 85–103. Springer, Boston, MA (1972)
9. Liu K., Löffler S., and Hofstedt, P.: Exploring Properties of Icosoku by Constraint Satisfaction Approach, in post proc of *Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2019*, LNCS 12057, pp 99–105, 2020
10. Mainini A.: Profile page in Spiele-Autoren-Zunft e.V. Game Designers Association's website, https://www.spieleautorenzunft.de/authors-details.html?member=241. Last accessed 30 March 2020
11. Papadimitriou C. H.: Computational Complexity. Addison-Wesley (1994)
12. Stuckey P. J., Becket R. , Fischer J.: Philosophy of the MiniZinc challenge. Constraints 15 (3), 307-316 (2010)
13. Stuckey P. J., Marriot K., Tack G.: The MiniZinc Handbook 2.3.1. https://www.minizinc.org
14. Tange O.: GNU Parallel 2018. https://doi.org/https://doi.org/10.5281/zenodo.1146014 (2018)