**University of Udine – School for Advanced Studies**

DEPARTMENT OF MATHEMATICS, COMPUTER SCIENCE AND PHYSICS
Master's Degree in Computer Science

Mid-term paper

# CSP Modeling of IcoSoKu

Student:
**Nicola Rizzo**

Professor:
**Agostino Dovier**

**Academic Year 2019-2020**

# Contents

## 1 Introduction

IcoSoKu, shown in Figure 1, is a mechanical puzzle created by Andrea Mainini in 2009. Its generalization, 3SoKu, which we define and study in this paper, is interesting from a computational point of view since it is NP-complete. In Section 3 and 4 we model 3SoKu as a constraint satisfaction problem in MiniZinc, a constraint modeling language, and in Answer Set Programming, using the Potsdam Answer Set Solving Collection[1], and in Section 5 we compare the solvers on instances of IcoSoKu and use them to solve every possible IcoSoKu. All the code can be found on `github.com/nrizzo/3SoKu`.

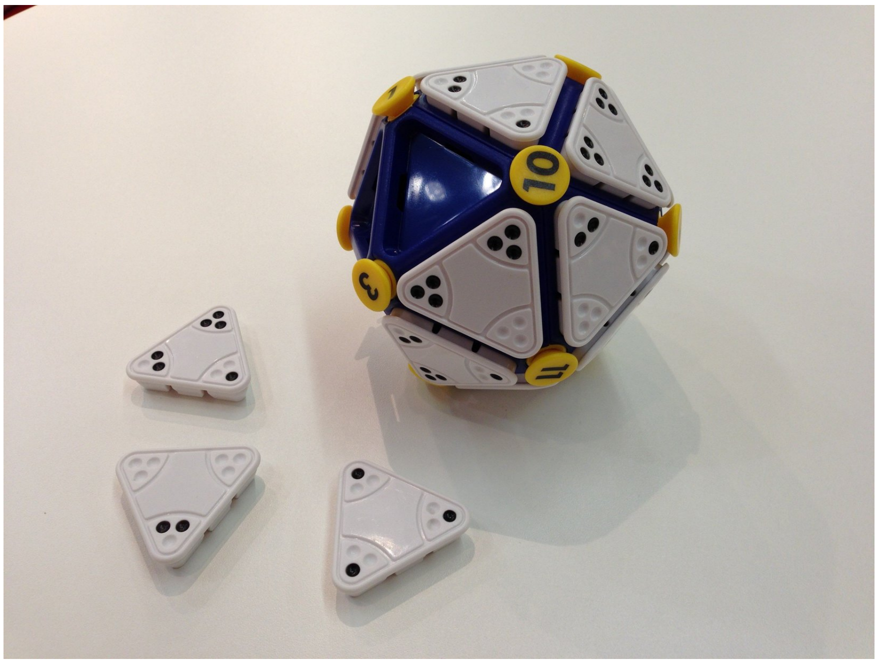


Figure 1: Photo of an IcoSoKu [1].

### 1.1 The puzzle

The puzzle is consisting of:

- a plastic blue icosahedron;
- 12 yellow pegs with the numbers from 1 to 12 written on their tips;
- and 20 triangular tiles presenting a number of black dots (from 0 up to 3) near each vertex.

[1]See `www.minizinc.org` and `potassco.org`.

The yellow tips are made to be inserted on the vertices of the icosahedron and the tiles can be placed onto the faces of the solid. The game is set up by placing all the pegs on the vertices in an arbitrary way. Then the goal is to place all the tiles on the icosahedron in a way such that the number of black dots surrounding each vertex is equal to the number of its peg. It is claimed that every setup of the pegs can be solved, but a justification as to why this is the case is not given. [1, 2]

Since an icosahedron has 12 vertices and 20 faces, all the pieces of the puzzle must be used. Also, the tiles are in the shape of an equilateral triangle, thus they can be rotated before being placed but they cannot be flipped: as seen in Figure 2, a tile with three different numbers of dots cannot be rotated into its specular version, while all the other tiles do not have this problem because they have an axis of simmetry and can be rotated to be their mirrored version.

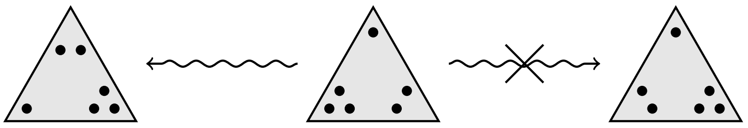
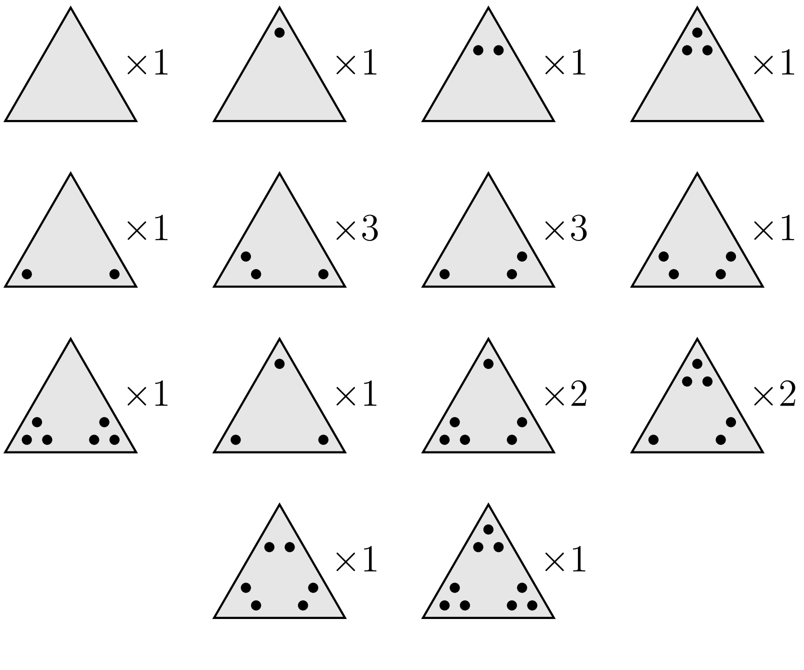


Figure 2: A tile with three different corners (center) can be rotated, as seen on the left, but not flipped, as seen on the right.

As we have said, the numbers on the pegs are the integers from 1 to 12 and it is easy to see that the total number of black dots must be 78. In fact, the tile configuration that comes with IcoSoKu is the following:



and by reading the corners in clockwise order we can describe this configuration as triples of integers, associated to the quantities of the corresponding tile:

- (0,0,0) ×1
- (0,0,1) ×1
- (0,0,2) ×1
- (0,0,3) ×1
- (0,1,1) ×1
- (0,1,2) ×3
- (0,2,1) ×3
- (0,2,2) ×1
- (0,3,3) ×1
- (1,1,1) ×1
- (1,2,3) ×2
- (3,2,1) ×2
- (2,2,2) ×1
- (3,3,3) ×1

Because the numbers on the pegs represent a quota of black dots that must be met (exactly), in the rest of the paper we will call them *capacities*. For the same reason, we will call *weight* the number of dots near a vertex of a tile. Then the problem of solving an IcoSoKu can be described as follows.

**Problem 1** (IcoSoKu)**.** Given an assignment of the each integer capacity in $\{1, \ldots, 12\}$ to a different vertex of the icosahedron, rotate and place each of the 20 tiles described above (as triples of non-negative integer weights) to a differenct face of the icosahedron in such a way that for every vertex, the sum of the weights surrounding it is equal to its capacity.

## 1.2 Naming vertices, faces and tiles

In the process of studying this problem we assign the letters from A to L to the vertices of the icosahedron, as seen in figure 3, and we refer to the faces using the three vertices that they involve in clockwise order from the outside of the solid. The 20 faces are:

- ABC;
- ACD;
- ADE;
- AEF;
- AFB;
- BFK;
- BKG;
- BGC;
- CGH;
- CHD;
- DHI;
- DIE;
- EIJ;
- EJF;
- FJK;
- GKL;
- GLH;
- HLI;
- ILJ;
- JLK;

Moreover, we will refer:

- to the tiles with the index they are presented with, starting from 1, if such index is necessary;

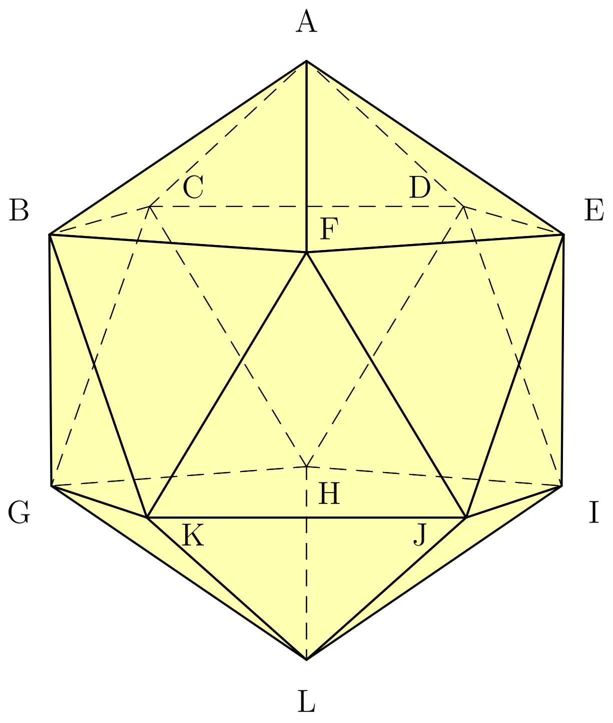


Figure 3: Assignment of letters to the vertices of the icosahedron.

- to the rotations of the tiles with the integers 0, 1 and 2 for describing clockwise rotations of 0, 120 and 240 degrees respectively.

So, for example, to assign the 6-th tile to face ABC with rotation 1 means taking tile $(0, 1, 2)$, rotating it by 120 degrees obtaining tile $(2, 1, 0)$, and placing weight 2 on the corner corresponding to vertex A, weight 1 on vertex B and weight 0 on C.

## 1.3 An existing IcoSoKu solver in JavaScript

An interesting benchmark for estimating the performance in solving IcoSoKus is Marzio De Biasi's IcoSoKu solver on `nearly42.org`: it is written in JavaScript and uses a backtracking algorithm to implement a heuristic (but deterministic) search of a solution. [3]

By analysing the code, it seems that the solution is found by:

1. selecting the free face $f$ such that $i(f)$ is minimum, with $i$ an index indicating how much the capacities of the relative vertices are yet to be filled (how many black dots are needed to fill the capacity);

2. trying placing one by one all available tiles on $f$, following the lexicographical order of their triples (being careful about equivalent tiles)

and recursively solving the IcoSoKu from these partial solutions.

In most cases the program solves the problem istantaneously but sometimes it can hang for a variable number of seconds before finding a solution (see the tests in Section 5).

## 2 Generalizing IcoSoKu to 3SoKu

It is easy to see that IcoSoKu, seen as a constraint satisfaction problem, admits a variant for every polyhedron with regular faces, or faces of the same size and shape. Platonic solids, seen in Figure 4, fit into this category, but infinite more polyhedra can be a gaming field, characterising the way the vertices are connected. So, in order to tackle a problem that is both general[2] and interesting we have chosen to study the following generalization of IcoSoKu, which we call 3SoKu, or $P$-SoKu, depending on the polyhedron $P$ with triangular faces that decides the playing field.

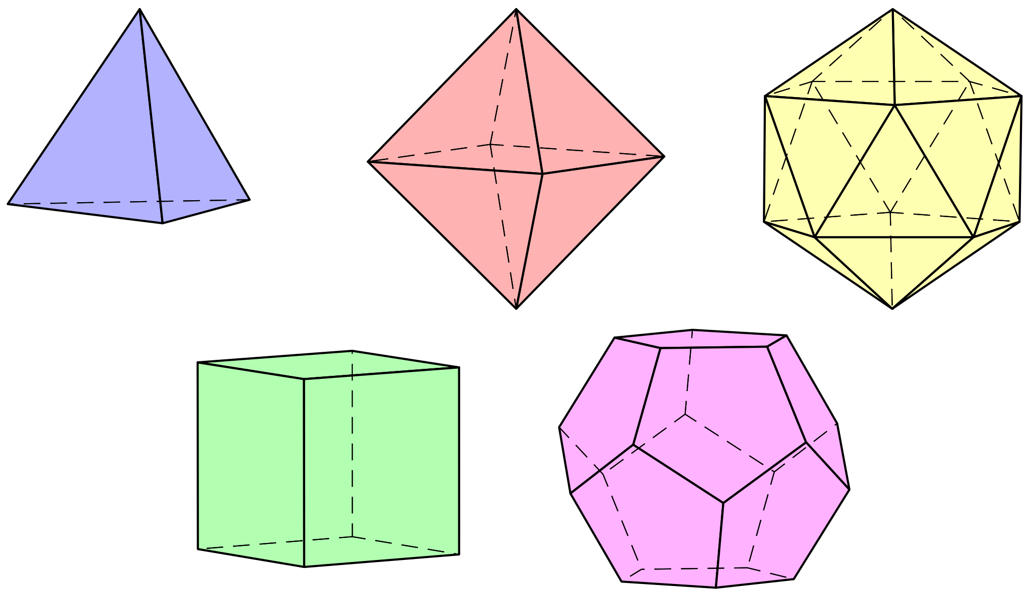


Figure 4: Platonic solid, for example, can be the playing field of the game, although we decide (arbitrarily) to admit only solids with triangular faces.

[2]IcoSoKu in its commercial incarnation is a simple problem from a complexity standpoint, since the number of different configurations of the pins is finite, albeit apparently big.

## 2.1 The problem

First, we formalize the generalization of the icosahedron, which is the playing field, as a set of vertices and a set of faces involving these.

**Definition 1.** A couple $(V, F)$ is a **playing field for 3SoKu** (or just playing field) if:

- $V$ is a finite set $\{v_1, \dots, v_m\}$ of $m$ vertices;
- $F$ is a finite set of $n$ triples of vertices describing the faces of the polyhedron, that is $F = \{f_1, \dots, f_n\} \subseteq \mathcal{P}(V \times V \times V)$; given a face $f = (v_0, v_1, v_2) \in F$, we use $\pi_i(f)$ to indicate its $(i+1)$-th element, with $i = 0, 1, 2$, that is $\pi_0(f) = v_0$, $\pi_1(f) = v_1$ and $\pi_2(f) = v_2$.

For convenience, we define for each vertex $v \in V$ the set

$$F_v \coloneqq \big\{(f, i) \mid \pi_i(f) = v\big\}$$

containing each occurrence of a vertex of the polyhedron, described by a face and an index.

For example, the shape of the tetrahedron is described by $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$ and $F = \{\mathrm{ABC}, \mathrm{ACD}, \mathrm{ADB}, \mathrm{BDC}\}$, when using a similar naming convention we used for the icosahedron, as shown in Figure 5.

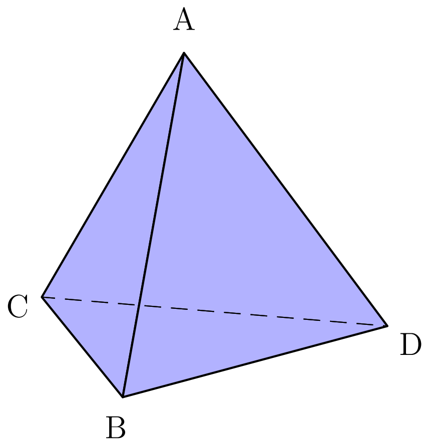


Figure 5: Naming the vertices of the tetrahedron with the letters A, B, C and D.

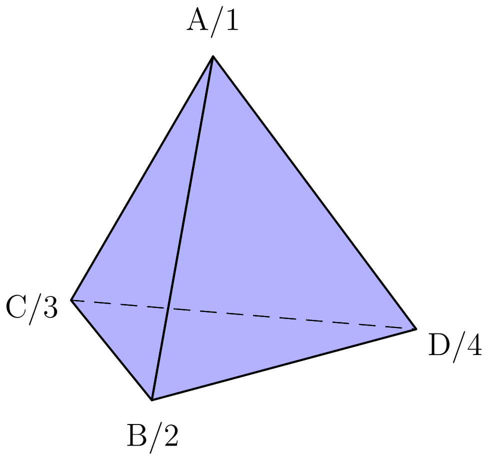
Next we can define the setup of a game of 3SoKu, adding two ingredients: the capacities assigned to the vertices, those were originally the yellow pegs, and the tiles, containing the weights that will serve to satisfy the capacities.

**Definition 2.** A **game of 3SoKu** is a triple $\big((V,F),c,T\big)$ composed of:

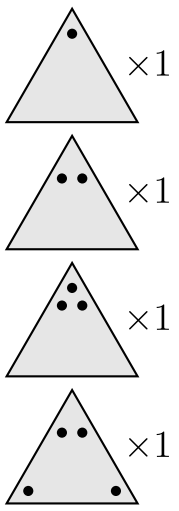
- a playing field $(V,F)$, with $m$ vertices and $n$ faces, that is $|V|=m$ and $|F|=n$;
- a map $c\colon V\to\mathbb{N}$, which we call **capacity map** of $V$, that assignes an integer weight to each vertex;
- a multiset $T=\{t_1,\dots,t_n\}$ of **tiles**, such that each element $t_i$ is a triple of integer and non-negative weights; given a tile $t=(w_0,w_1,w_2)$, we use $\pi_i(t)$ to indicate its $(i+1)$-th element, with $i=0,1,2$, that is $\pi_0(t)=w_0$, $\pi_1(t)=w_1$ and $\pi_2(t)=w_2$.

For example, the game of TetraSoKu represented in Figure 6 is $\big((V,F),c,T\big)$ where:

- $V=\{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D}\}$ and $F=\{\mathrm{ABC},\mathrm{ACD},\mathrm{ADB},\mathrm{BDC}\}$ as described previously;
- $c$ is such that $c(\mathrm{A})=1$, $c(\mathrm{B})=2$, $c(\mathrm{C})=3$ and $c(\mathrm{D})=4$;
- $T=\big\{(0,0,1),(0,0,2),(0,0,3),(1,1,2)\big\}$;



(a) Vertices, faces and capacities.



(b) Tiles.

Figure 6: Representation of a game of TetraSoKu.

The last component of the game left to be formalized is the solving part, i.e. placing the tiles onto the faces, after having possibly rotated the pieces.

**Definition 3.** Given a game of 3SoKu $\big((V,F),c,T\big)$, an **arrangement** of the tiles in $T$ to the faces in $F$ is a couple $(r,p)$ such that:

- $r\colon T \to \{0,1,2\}$ describes the clockwise rotation of the tiles; that is, we say tile $t=(w_0,w_1,w_2)\in T$ becomes tile $(w_2,w_0,w_1)$ if $r(t)=1$, tile $(w_1,w_2,w_0)$ if $r(t)=2$ and stays the same if $r(t)=0$;
- $p\colon T \to F$ is a bijection describing the placement of the (rotated) tiles to the faces.

Given an arrangement $(r,p)$, for every tile $t=(w_0,w_1,w_2)\in T$ we say that its weight $w_i$, is placed near vertex $\pi_{i'}\big(p(t)\big)$, with $i' := i + r(t) \mod 3$, $i\in\{0,1,2\}$.

A possible arrangement for the game of TetraSoKu of Figure 6 is represented in Figure 7: it is easy to check that it is a correct solution in the sense of the goal of IcoSoKu, which we are now ready to describe formally.

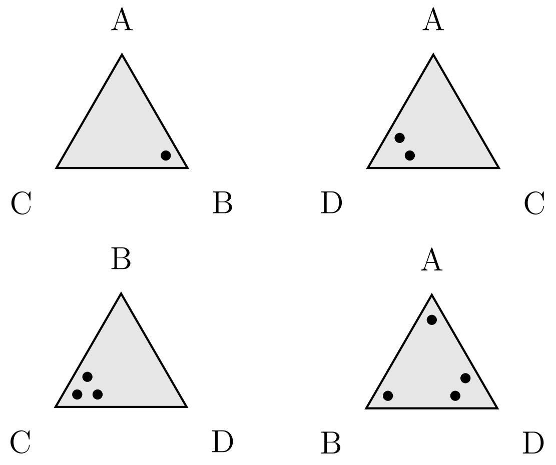


Figure 7: Arrangement for a game of TetraSoKu: the weights near vertices A, B, C and D add up to, respectively, 1, 2, 3 and 4.

**Problem 2** (3SoKu)**.** Given a game of 3SoKu $\big((V,F),c,T\big)$, find an arrangement $(r,p)$ of its tiles to its faces such that

$$\sum_{(f,i)\in F_v} \pi_{r(f,i)}\big(p^{-1}(f)\big) = c(v) \qquad \forall v\in V, \tag{1}$$

where $r(f,i) := i - r\big(p^{-1}(f)\big) \mod 3$, for each $f\in F$, $i\in\{0,1,2\}$.

The constraints of Equation (1) describe how the integer weights placed near each vertex $v$ must add up to its capacity $c(v)$. In fact, $p^{-1}(f)$ refers to the tile assigned to face $f$, of which we take the weight with index $i - r(p^{-1}(f))$: the minus sign can be explained observing that in rotating the tiles clockwise we increment (and loop back if necessary) the index of its weights, and that to find which weight is at index $i$ after the rotation we must subtract that rotation.

## 2.2 Considerations

There are two potential issues with the given definition of 3SoKu:

- Definition 1, that defines what a playing field is, allows the description of *bad* polyhedra that do not make sense in 3D space, or at all, such as $\big(\{\mathrm{A}\}, \{(\mathrm{A}, \mathrm{A}, \mathrm{A})\}\big)$, that corresponds to a polyhedron with one vertex and one face;
- the practical nature of the game requires that each tile can be perfectly placed on every faces, but in the formalization this feature gets weakened into the requisite for the tiles to be triangular, ignoring the actual shape of the tiles and of the faces.

These issues can be dismissed quickly because, as seen in Figure 7, checking the constraints of 3SoKu does not involve the plausibility of the polyhedron or the shapes of the tiles, and the models we will develop in Section 3 and 4 do not assume or exploit that. Anyway, even if we restrict ourselves to only using valid polyhedra as gaming field, the restriction of the faces to be triangular does not limit in an obvious way the complexity of the problem because, for example, there are infinite deltahedra, that is polyhedra with equilateral triangles as faces.[3] Also, in the next section we will show that by admitting only well-formed triangular polyhedra with the faces of the same shape and size the problem retains its complexity.

## 2.3 NP-completeness

We now study the complexity of 3SoKu, or to be more precise of its decisional version, that asks whether a given game of 3SoKu admits a solution or not. To proove that this problem is NP-complete, i.e. that every problem in the class NP can be reduced to 3SoKu, we will show that it actually is in NP and that an NP-complete problem is at least as hard as 3SoKu (using the weakest

[3]The number of strictly-convex deltahedra is finite (see [4]), but there is an infinite number of (non-strictly-)convex deltahedra and of non-convex deltahedra.

definition of reduction, that can be computed using logarithmic space, see [5, Chapter 8]).

Regarding the first task, it is easy to see that 3SoKu is in NP: given an instance of 3SoKU and an arrangement of its tiles to its faces, checking if the arrangement satisfies the constraints described by Equation (1) requires $O(n)$ additions, subtractions and equalities of the integers given in input; also, all possible arrangements can be generated in a non-deterministic fashion in polynomial time.

Next, the chosen problem we will reduce is one of Karp's original 21 NP-complete problems, the *partition problem* (referred also as numbering partitioning), but we will use Garey and Johnson's formulation of 1977 (see [6, p. 97] and [7, pp. 60-61, 223]).[4]

**Problem 3** (Partition)**.** Given a finite set $A$ and a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$, find is there a subset $A' \subseteq A$ such that

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

A reduction from Partition to 3SoKu is a function from strings to strings:

- that transforms an instance of Partition to a game of 3SoKu such that the resulting game admits a solutions if and only if the original instance does;
- and that is computable by a Turing Machine using $O(\log n)$ space, where $n$ is the size of the input.

This implies a *reasonable* coding of both instances, such as the binary positional notation for integers and a list of couples key-value for functions with a finite domain such as the capacity function of 3SoKu.

**Lemma 1.** Partition can be reduced to 3SoKu.

*Proof.* We want to transform a generic instance of Partition, described by sizes $s_1, s_2, \ldots, s_n$, into a game of 3SoKu $\big((V, F), c, T\big)$ such that the latter admits a valid configuration if and only if the former has a solution. The idea behind this is:

[4]In [6] and [7] polynomial-time reductions are used, and we do not know if it has been demonstrated that Partition is NP-complete using log-space reductions. If this is not the case, then our proof of NP-completeness will be valid just under the polynomial-time definition of reduction.

1. to use as playing field a bipyramid with $2n$ faces composed by fusing two $n$-gonal pyramids base to base, such that the two apices correspond to sets $A'$ and $A' \setminus A$ of a possible solution and the other $2n-2$ vertices are dummy vertices, as seen in Figure 8;
2. to assign capacity $\sum_{i=1}^{n} s_i/2$ to the apices $A'$ and $A' \setminus A$ and capacity 0 to the dummy vertices;
3. to build a set of $2n$ tiles divided in $n$ empty tiles $(0,0,0)$, and tiles $(s_i, 0, 0)$ for $i \in \{1, \ldots, n\}$, as seen in Figure 9.

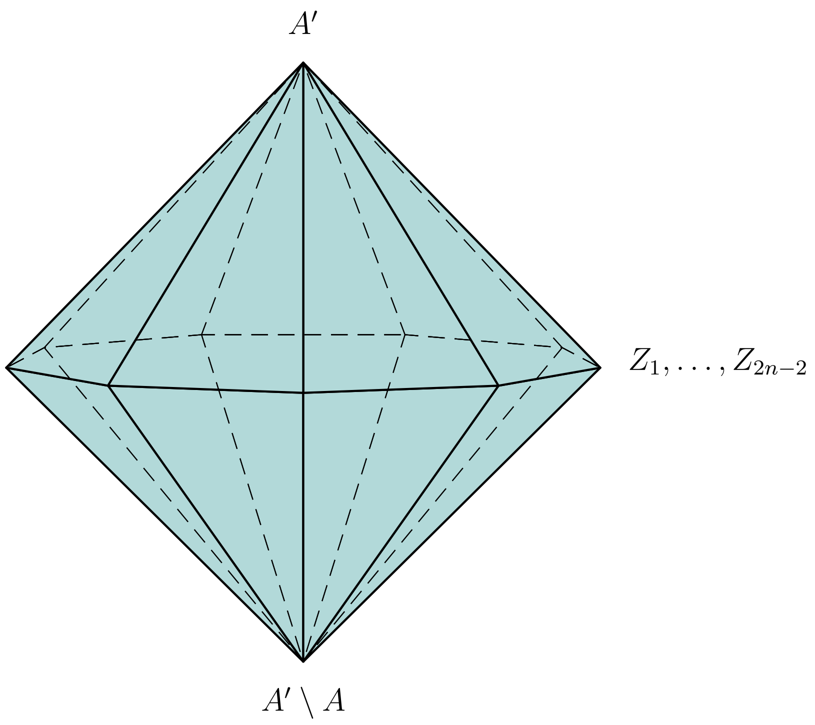


Figure 8: Scheme for the playing field for 3SoKu associated to an instance of partition with $n$ elements.

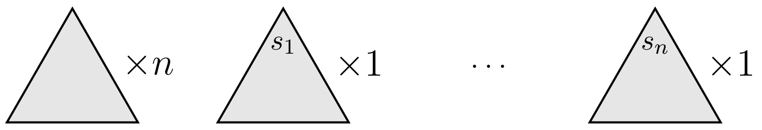


Figure 9: Representation of the tiles corresponding to instance $s_1$, ..., $s_n$ of Partition.

That translates to building the game of 3SoKu $\big((V,F),c,T\big)$ with

$$\begin{aligned}
V &= \{A',\ A'\setminus A,\ Z_1,\dots,Z_{2n-2}\},\\
F &= \big\{(A',Z_1,Z_2),(A',Z_2,Z_3),\dots,(A',Z_n,Z_1),\\
&\quad (A'\setminus A,Z_2,Z_1),(A'\setminus A,Z_3,Z_2),\dots,(A'\setminus A,Z_1,Z_n)\big\},\\
c(v) &= \begin{cases}\sum_{i=1}^n s_i/2 & \text{if } v = A',\ A'\setminus A\\ 0 & \text{otherwise,}\end{cases}\\
T &= \{t_1,\dots t_n,z_1,\dots,z_n\}\\
&\quad \text{where} \qquad t_i=(s_i,0,0),\ z_i=(0,0,0) \quad \forall i\in\{1,\dots n\}.
\end{aligned}$$

It is easy to see that if the Partition instance admits a solution $A'$ corresponding to sizes $q_1,\dots,q_k$, with $k\in\{1,\dots,n\}$, then there is an arrangement $(r,p)$ that satisfies the equations defined by (1), placing all the weights corresponding to sizes $q_1$, ..., $q_k$ near vertex $A'$, placing the rest of the positive weights near $A'\setminus A$ and arbitrarily placing all remaining $(0,0,0)$ tiles. Conversely, if the game of 3SoKu corresponding to instance $s_1,\dots,s_n$ admits a valid arrangement $(r,p)$, then this means that there are sizes $q_1$, ..., $q_k$ with $k\in\{1,\dots,n\}$ that add up to $\sum_{i=1}^n s_i/2$ near vertex $A'$, defining a solution for the original Partition instance.

We have left to show that the game of 3SoKu can be computed in an efficient manner: the transformation is quite straightforward and it can be easily computed by a Turing Machine with I/O using a constant number of pointers to the input and output and a constant number of integers in $1,\dots,2n$, except for the computation of $\sum_{i=1}^n s_i/2$, a quantity not necessarily explicit in the coding of the input. Assuming we can compute this value using logarithmic space, and that will be demonstrated by the next lemma, then Partition reduces to 3SoKu. □

**Lemma 2.** Given the positive integers $s_1$, ..., $s_n$, a Turing Machine with I/O can compute the value $s=\sum_{i=1}^n s_i$ using logarithmic space.

*Proof.* Assuming the input is reasonably encoded in space $O(l)$ with $l$ equal to $\sum_{i=1}^n \log_2 s_i$, we must find a Turing Machine with I/O that find $s$ using

$O(\log l)$ space. This can be done with a simple trick: we will compute and write to the output tape the solution $s$ from its least significant bit to its most significant, using the generalized algorithm of the simple addition of two numbers encoded in binary.

The machine operates in the following way. Initially it sets the counter $i$ to 0 carry $r$ to 0 (they each have their own tape for simplicity). Then, while $i$ is less or equal to the maximum number of bits occupied by $s_1, \ldots, s_n$:

1. it adds to the carry $r$ the $(i+1)$-th least significant bit of each $s_i$ with $i \in \{1, \ldots, n\}$;

2. it writes in the output tape the least significant bit of $r$;

3. it erases the least significant bit of $r$;

4. it increases $i$ by 1.

It can do this using a pointer to the input and one to the output and keeping $i$, $r$ and another integer to find the $(i+1)$-th bit of a size in input. $i$ is always less than the number of digits of the highest integer in input, so $i \leq l$ and its representation is $O(\log n)$. A generous upper bound for $r$ is $n \cdot l$, since at each iteration of the while loop $r$ is incremented by at most $n$ and divided by 2, and there are less than $l$ iterations, so a representation of $r$ takes space $O(\log l)$ as well. □

The next theorem follows.

**Theorem.** 3SoKu is NP-complete.

Note how the bipyramid built in the proof of Lemma 1 has isosceles triangles of the same size and shape as its faces: this means that imposing 3SoKu to be practically playable, imposing that all faces and tiles are of the same size and shape (eventually limiting the rotation of the tiles, although that would change in a non trivial manner the description of 3SoKu), does not in fact reduce the complexity of the problem.

# 3 The MiniZinc model

In this section we present the MiniZinc model, alongside its configuration to solve instances of IcoSoKu. In fact, the heart of the model is `3SoKu.mzn` but the file on its own it is incomplete because:

- it is missing the description of the variant played, polyhedron $P$, so it must import an appropriate file such as `variants/ico.mzn`;
- it is missing the description of the input, the capacities of the vertices and the weights of the tiles, that must be added too in a file such as `input-ico.dzn`.

## 3.1 Overview

First, the following parameters describing $P$ must be set to decide the variant of 3SoKu to play:

- the integers `m` and `n`, equal to the number of vertices and faces of $P$;
- two enumerated types `VERTEX` and `FACE` that indicate the vertices and faces of $P$;
- a matrix `vrtx` with $n$ rows and 3 columns such that there is a row for each face of $P$ and each row contains the vertices involved in the relative face in clockwise order (so for each face in `FACE` and each integer in $\{0, 1, 2\}$ `vrtx` returns a value in `VERTEX`).

Listing 1 shows how to set $P$ as the icosahedron.

The instance of $P$-SoKu to solve is given by the array `cap` of $m$ integers, that for each vertex in `VERTEX` indicates its capacity, and by the array `weight` with $n$ rows and 3 columns such that each row, indicated by an index of a tile (the integers in $\{1, \dots, n\}$), contains the three integer weights describing the tile in clockwise order, accessed with an index in $\{0, 1, 2\}$. An example of an instance of IcoSoKu is shown in Listing 2, as the tiles are those described in Section 1.1 and the capacities are a permutation of the numbers in $\{1, \dots, 12\}$.

The solution, as shown in Listing 3 is modeled by:

- the array `tile` of $n$ variables taking value in $\{1, \dots, n\}$, that describes for each `FACE` the index of the assigned tile;
- the array `rot` of $n$ variables in $\{0, 1, 2\}$ that indicates for each `FACE` the clockwise rotation of its associated tile.

```
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,  A, E, F,  A, F, B,
   B, F, K,  B, K, G,  B, G, C,  C, G, H,  C, H, D,
   D, I, E,  D, H, I,  E, I, J,  E, J, F,  F, J, K,
   G, K, L,  G, L, H,  H, L, I,  I, L, J,  J, L, K]);
```

Listing 1: Setting the icosahedron as playing field of the MiniZinc model.

```
array[VERTEX] of int: cap = [1,5,11,10,7,12,9,2,6,8,4,3];
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 2: Instance of IcoSoKu for the MiniZinc model.

```
array[FACE] of var 1..n: tile;
array[FACE] of var 0..2: rot;
```

Listing 3: Solution scheme of the MiniZinc model.

So if `tile[f]` has value $i$ and `rot[f]` has value $r$ then it means that the solution has the $i$-th tile assigned to face `f`, rotated clockwise $r$ times (0, 120 and 240 degrees for rotations, respectively, 0, 1 and 2).

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. This constraint is easily enforced using the matrix `vrtx` and the modulo operation, as seen in Listing 4.

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

constraint alldifferent(tile);
constraint forall (v in VERTEX) (vertex_sum(v) == cap[v]);
```

Listing 4: Constraints of the MiniZinc model.

The following constraints, breaking eventual simmetries between the tiles, can be added as seen in Listing 5: uniform tiles, such as $(1, 1, 1)$, do not need to be rotated; duplicate tiles can conserve their relative order when mapped to the tiles without losing possible solutions (here we assume a normal notation for describing a tile and we check their equivalence without rotating them).

```
constraint forall (t in 1..n)
  (if weight[t, 0] == weight[t, 1] /\
      weight[t, 1] == weight[t, 2]
   then forall (f in FACE where tile[f] == t)
               (rot[f] == 0)
   endif);

constraint forall (t1 in 1..n, t2 in 1..n where t1 < t2 /\
                        forall (i in 0..2)
                        (weight[t1,i] == weight[t2,i]))
                     (forall (f1 in FACE, f2 in FACE where
                            tile[f1]==t1 /\ tile[f2]==t2)
                           (f1 < f2));
```

Listing 5: Simmetry breaking constraints of the MiniZinc model.

## 3.2 Usage

As we have said, `3SoKu.mzn` must import the file describing $P$ and should be run loading the data of the instance from a data file, like in this next command.

```
$ minizinc --solver gecode 3SoKu.mzn input-ico.dzn
```

The standard output to the solution, showing the values of the arrays `tile` and `rot`, is not very readable so a function `instruction`, shown in Listing 6, has been written to print the instructions to the solution. Also other print functions (not shown here) have been written for the generation of LaTeX code to embed the solution in a document, as seen in figure 10, as a sequence of tables (the document is compiled with XeLaTeX).

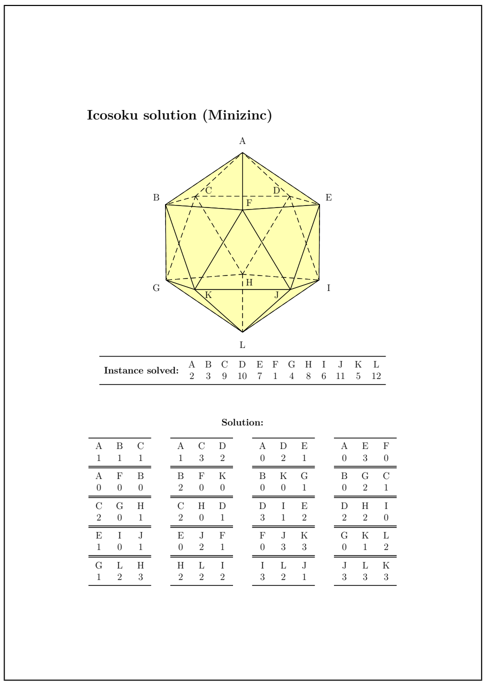
```
function string: instruction(FACE: f) =
  let { 0..3: x = weight[fix(tile[f]),
                         (3 - fix(rot[f])) mod 3],
        0..3: y = weight[fix(tile[f]),
                         (4 - fix(rot[f])) mod 3],
        0..3: z = weight[fix(tile[f]),
                         (2 - fix(rot[f])) mod 3],
        VERTEX: u = vrtx[f, 0],
        VERTEX: v = vrtx[f, 1],
        VERTEX: w = vrtx[f, 2] }
  in "Put tile (" ++ show(x) ++
             ", " ++ show(y) ++
             ", " ++ show(z) ++ ")" ++
     " on face (" ++ show(u) ++
             ", " ++ show(v) ++
             ", " ++ show(w) ++ ").\n";

output [ instruction(f) | f in FACE ];
```

Listing 6: Output instructions to the solution found by the MiniZinc model.

## 3.3 Search strategy

Our tests in Section 5 show that Gecode's standard exploration of the search tree of IcoSoKu instances has a performance similar to the Javascript solver, albeit slower, in the sense that often a solution is found very quickly but sometimes the solver can get stuck in a big subtree with no solution. The fact that placing a tile consists of the choice of two variables (the face and



**Icosoku solution (Minizinc)**

| **Instance solved:** | A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 9 | 10 | 7 | 1 | 4 | 8 | 6 | 11 | 5 | 12 |

**Solution:**

| | | |
|---|---|---|
| A | B | C |
| 1 | 1 | 1 |
| A | F | B |
| 0 | 0 | 0 |
| C | G | H |
| 2 | 0 | 1 |
| E | I | J |
| 1 | 0 | 1 |
| G | L | H |
| 1 | 2 | 3 |

| | | |
|---|---|---|
| A | C | D |
| 1 | 3 | 2 |
| B | F | K |
| 2 | 0 | 0 |
| C | H | D |
| 2 | 0 | 1 |
| E | J | F |
| 0 | 2 | 1 |
| H | L | I |
| 2 | 2 | 2 |

| | | |
|---|---|---|
| A | D | E |
| 0 | 2 | 1 |
| B | K | G |
| 0 | 0 | 1 |
| D | I | E |
| 3 | 1 | 2 |
| F | J | K |
| 0 | 3 | 3 |
| I | L | J |
| 3 | 2 | 1 |

| | | |
|---|---|---|
| A | E | F |
| 0 | 3 | 0 |
| B | G | C |
| 0 | 2 | 1 |
| D | H | I |
| 2 | 2 | 0 |
| G | K | L |
| 0 | 1 | 2 |
| J | L | K |
| 3 | 3 | 3 |

Figure 10: LaTeX document for a solution of IcoSoKu, describing where to put the tiles.

the rotation) does not seem to matter, as we have explored alternative models, and even adding redundant constraints to discard unsatisfiable partial solutions, exploiting the fact that all the weights are positive, helps pruning the tree a bit by having more failures and propagations but does not seem to affect performance.

Fortunately, Gecode's behaviour can be modified to exploit a possible statistical property of IcoSoKu: there always seems to be a very high number of solutions, probably due to the configuration of the tiles and the massive number of possible placements. In fact, compiling to FlatZinc the solver on the instance defined by `cap = [8,5,11,7,9,10,3,6,1,4,2,12]` and asking Chuffed to find all its solutions with the following commands, found more than 60 million different solutions (with the simmetry breaking constraints!) before maxing out our machine's 16 GBs of memory and having to be terminated prematurely.

```
$ minizinc --compile 3SoKu.mzn input-ico.dzn
$ fzn-chuffed --verbosity 2 --print-sol off -a \
  3SoKu.fzn
```

For this reason, a randomized search plus a restart after a low constant number of nodes visited, implemented in Listing 7, makes a good strategy for solving the problem, as will be shown in Section 5.

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

Listing 7: Randomized search for the MiniZinc model.

## 4 The ASP model

The heart of the model is `3SoKu.lp` but it is incomplete on its own since it lacks the description of polyhedron $P$, the playing field, that goes in the folder named `variants`, and the description of the instance to solve: these must be added to launch the grounder and the solver. In this section we present an overview of the model and how it has been configured to solve instances of IcoSoKu.

### 4.1 Overview

This model is similar to the MiniZinc one, as the specification of polyhedron $P$ consists of:

- constants `m` and `n`, equal to the number of vertices and faces of $P$;
- the vertices and faces of $P$, constants of the problem (so lowercase letters) identified by the functions `vertex/1` and `face/1`;
- function `vrtx/3`, such that for each face and index 0, 1 and 2 it indicates respectively the first, second and third vertex of the face in clockwise order (`vrtx(v, f, i)` means that `v` is the (`i`+1)-th vertex of face `f`).

A possible specification of the icosahedron is shown in Listing 9.

```
tile(1..n).
rotation(0..2).
```

Listing 8: Definition of the tiles and of the possible rotations for the ASP model.

The input to the problem is given by function `cap/2`, that for each vertex indicates its integer capacity, and by function `weight/3`, that for each tile

```
#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; bfk; bkg; bgc; cgh; chd;
   die; dhi; eij; ejf; fjk; gkl; glh; hli; ilj; jlk).
vrtx(a, abc, 0;  b, abc, 1;  c, abc, 2;
     a, acd, 0;  c, acd, 1;  d, acd, 2;
     a, ade, 0;  d, ade, 1;  e, ade, 2;
     a, aef, 0;  e, aef, 1;  f, aef, 2;
     a, afb, 0;  f, afb, 1;  b, afb, 2;
     b, bfk, 0;  f, bfk, 1;  k, bfk, 2;
     b, bkg, 0;  k, bkg, 1;  g, bkg, 2;
     b, bgc, 0;  g, bgc, 1;  c, bgc, 2;
     c, cgh, 0;  g, cgh, 1;  h, cgh, 2;
     c, chd, 0;  h, chd, 1;  d, chd, 2;
     d, die, 0;  i, die, 1;  e, die, 2;
     d, dhi, 0;  h, dhi, 1;  i, dhi, 2;
     e, eij, 0;  i, eij, 1;  j, eij, 2;
     e, ejf, 0;  j, ejf, 1;  f, ejf, 2;
     f, fjk, 0;  j, fjk, 1;  k, fjk, 2;
     g, gkl, 0;  k, gkl, 1;  l, gkl, 2;
     g, glh, 0;  l, glh, 1;  h, glh, 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 9: Setting the icosahedron as playing field of the ASP model.

```
cap(a,8; b,5; c,11; d,7; e,9; f,10; g,3; h,6; i,1; l,4;
   m,2; n, 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;
       4,  0, 0;  4,  1, 0;  4,  2, 3;
       5,  0, 0;  5,  1, 1;  5,  2, 1;
       6,  0, 0;  6,  1, 1;  6,  2, 2;
       7,  0, 0;  7,  1, 1;  7,  2, 2;
       8,  0, 0;  8,  1, 1;  8,  2, 2;
       9,  0, 0;  9,  1, 2;  9,  2, 1;
       10, 0, 0;  10, 1, 2;  10, 2, 1;
       11, 0, 0;  11, 1, 2;  11, 2, 1;
       12, 0, 0;  12, 1, 2;  12, 2, 2;
       13, 0, 0;  13, 1, 3;  13, 2, 3;
       14, 0, 1;  14, 1, 1;  14, 2, 1;
       15, 0, 1;  15, 1, 2;  15, 2, 3;
       16, 0, 1;  16, 1, 2;  16, 2, 3;
       17, 0, 3;  17, 1, 2;  17, 2, 1;
       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 10: Instance of IcoSoKu for the ASP model.

(an integer in $\{1, \dots, n\}$) and each index 0, 1 and 2 indicates the weight of the corresponding corner. An instance of IcoSoKu is shown in Listing 10.

The solution is modeled by functions `assign/2` and `rotate/2`, that indicate respectively what face each tile is assigned to and the rotation of each tile. To constraint the solution to be correct:

- 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 auxiliary functions `contribute` and `vertex_sum` to impose the capacities of the vertices to be met exactly, as seen in Listing 11.

```
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).

contribute(T, V, P) :- assign(T, F), rotate(T, R),
  vrtx(V, F, A), weight(T, (A - R + 3) \ 3, P).

vertex_sum(V, S) :- vertex(V),
  S = #sum { P,T : contribute(T, V, P) }.

S = C :- vertex_sum(V, S), cap(V, C).
```

Listing 11: Rules to constraint the solution of the ASP model.

Finally, we can add simmetry breaking constraints regarding the tiles: uniform tiles should not be rotated and equivalent tiles should conserve their relative order when mapped to the faces of $P$. The rules implementing these are shown in Listing 12.

```
rotate(T,0) :- weight(T,0,I), weight(T,1,I), weight(T,2,I).

F1 < F2 :- assign(T1, F1), assign(T2, F2), T1 < T2,
  weight(T1, 0, I1), weight(T2, 0, I1),
  weight(T1, 1, I2), weight(T2, 1, I2),
  weight(T1, 2, I3), weight(T2, 2, I3).
```

Listing 12: Simmetry breaking rules of the ASP model.

## 4.2 Usage

For the model to function correctly, it is needed to add to `3SoKu.lp` the variant of 3SoKu played and the input like in the following command.

```
$ clingo 3SoKu.lp variants/ico.lp input-ico.lp
```

For the output to be readable, a rule defining function `put/6` has been added, shown in Listing 13, to generate the instructions to the solution, and a Bash script has been written (not shown here) to transform this output of the solver to tables of `pgfplotstable` to embed them in a LaTeX document (compiled with XeLaTeX).

```
put(V1, V2, V3, W1, W2, W3) :- assign(T, F), rotate(T, R),
  vrtx(V1, F, 0), vrtx(V2, F, 1), vrtx(V3, F, 2),
  weight(T, (3 - R) \ 3, W1),
  weight(T, (4 - R) \ 3, W2),
  weight(T, (2 - R) \ 3, W3).
```

Listing 13: Rule to the function showing the solution for the ASP model.

## 5 Tests and experiments

All the tests have been executed single-threaded on an Intel Core i5-7400 @ 3.30 GHz running Arch Linux. The version of the most important software used is shown in figure 11.

| Software | Description | Version |
|---|---|---|
| MiniZinc Distribution | MiniZinc compiler and FlatZinc solvers | 2.3.1 |
| clingo | Answer Set System | 5.3.0 |
| Node.js | JavaScript run-time environment (v8 engine) | 11.15.0 |

Figure 11: Software used in testing.

## 5.1 Performance tests for IcoSoKu

To measure the performance of the solvers we have written a bash script that starting from a seed generates[5] a batch of 100 tests (the file `batch`) and measures for each instance how much time the solvers take to solve the problem. More precisely:

- for the MiniZinc solver the time measured is its own "Overall time" from the program's statistics (option `-s`)[6];
- for the ASP solver we used Clingo's "CPU Time", already available;
- for the JavaScript solver, the function `icosolve` has been parameterized to be called with the capacities of the vertices as inputs and the time has been measured by using `new Date()` before and after calling `icosolve` and printing the difference of the two times in seconds.

For the ASP and JavaScript solvers a bash script `random-ico-instance.sh` has been written: if it is called with twelve arguments, specifying the capacities of the vertices according to our naming in figure 3, it uses them to modify the instance for IcoSoKu, othwerwise it generates a random instance[7]; it does not modify the tile configuration. For the MiniZinc solver the instance `input-ico.dzn` is modified directly by the script (in Section 5.2 it will be passed instead as a command line argument, with option `-D`, after having removed the line from `input-ico.dzn`).

The results are shown in the bar charts below, with the relative mean value drawn horizontally. They have the same scale but some results go beyond the grid, so for completeness the worst times are:

- 146.33 seconds for the plain MiniZinc model using Gecode;
- 0.51 seconds for the MiniZinc model using Gecode with the randomized search;
- 0.28 seconds for the MiniZinc model using Chuffed;

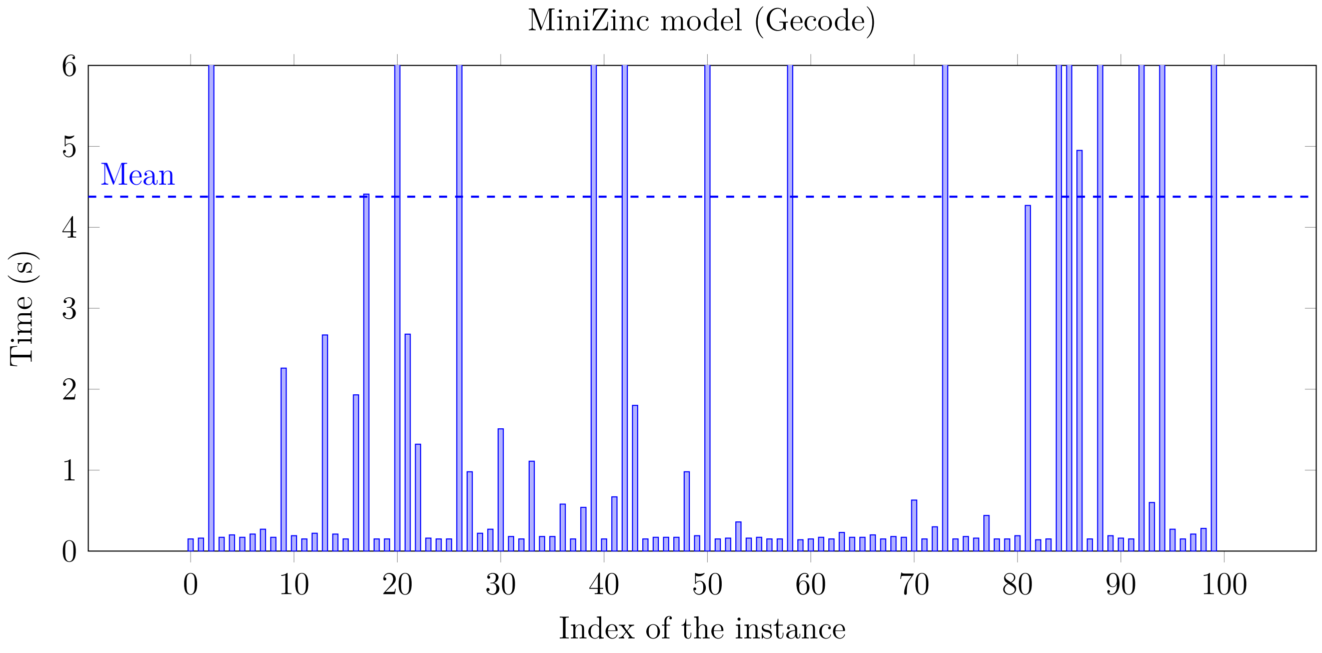
[5]It uses `openssl` as `shuf`'s pseudo-random source in a similar way as in `www.gnu.org/software/coreutils/manual/html_node/Random-sources.html`, accessed 30 March 2020.

[6]An even faster option would have been to skip the translation to FlatZinc, since the tiles do not change and the capacity of each vertex is defined in one single line using the constraint `int_lin_eq`; this option can be explored in the future if solving a high number of instances of 3SoKu is needed.

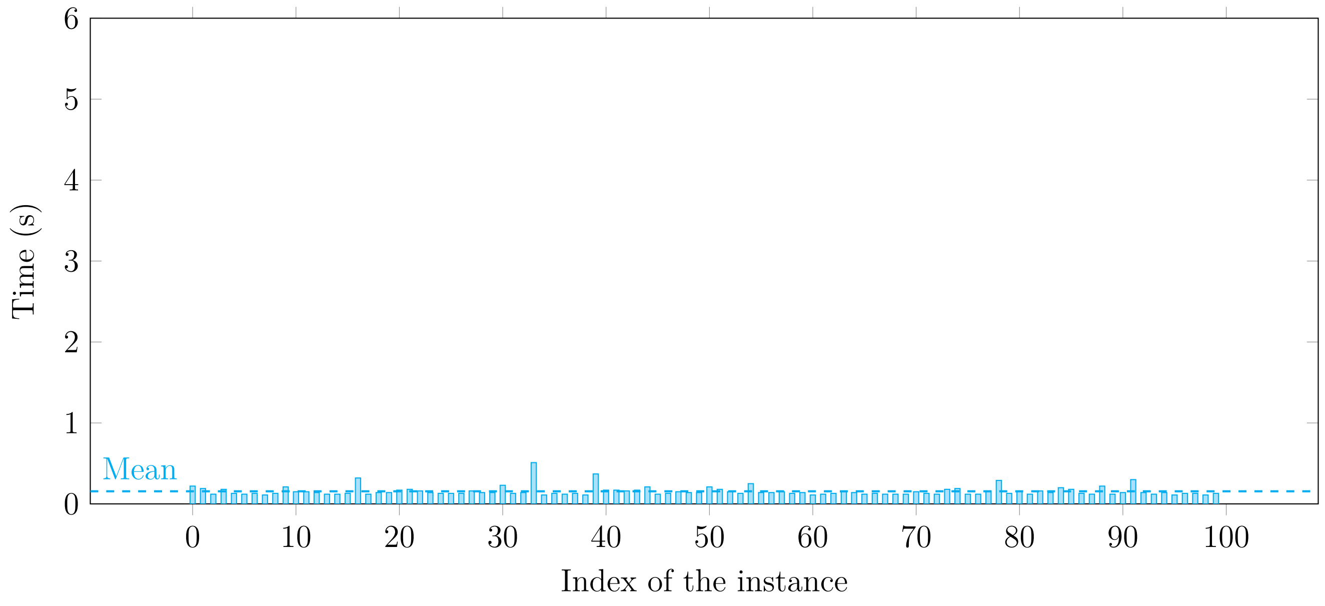
[7]In the case of the JavaScript solver, the script takes care of the conversion to the different naming of the vertices of the icosahedron used by De Biasi. [3]

- 1.20 seconds for the ASP model;
- 53.40 seconds for the JavaScript solver.

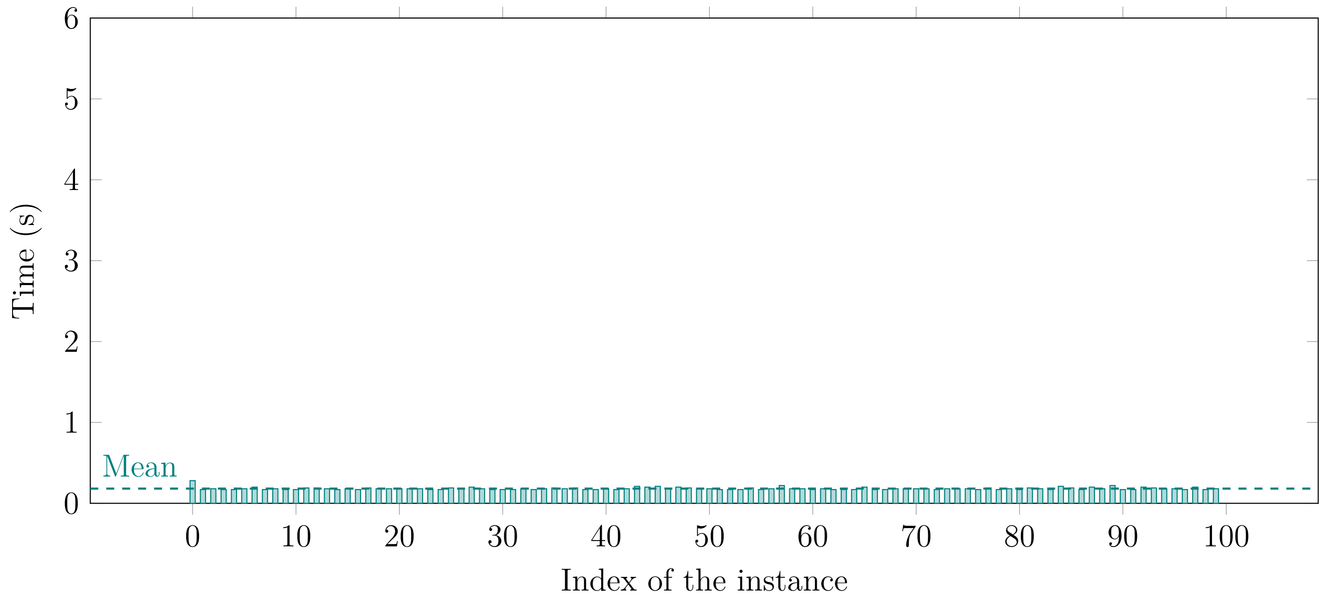
The most consistently fast models are MiniZinc ones, the Chuffed version and the Gecode version with the randomized search; the models that show the biggest difference in solving times are the MiniZinc one with Gecode's standard search and the JavaScript solver, even if it is istantaneous in many cases.

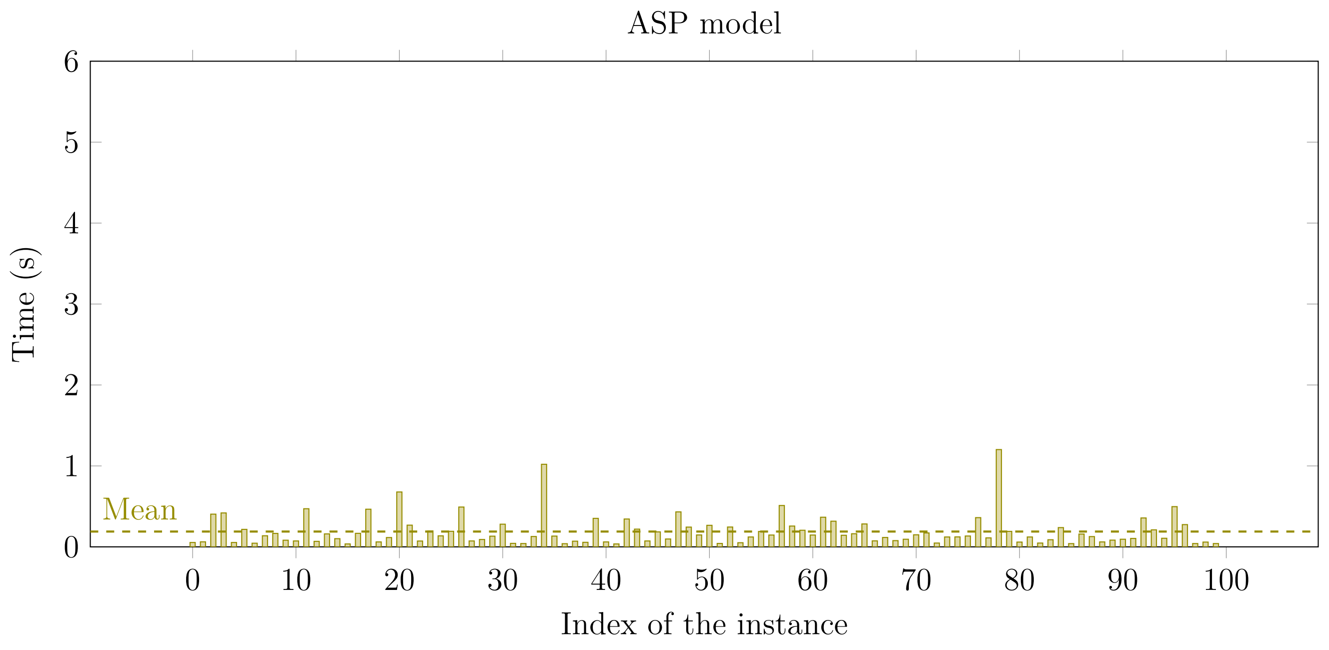


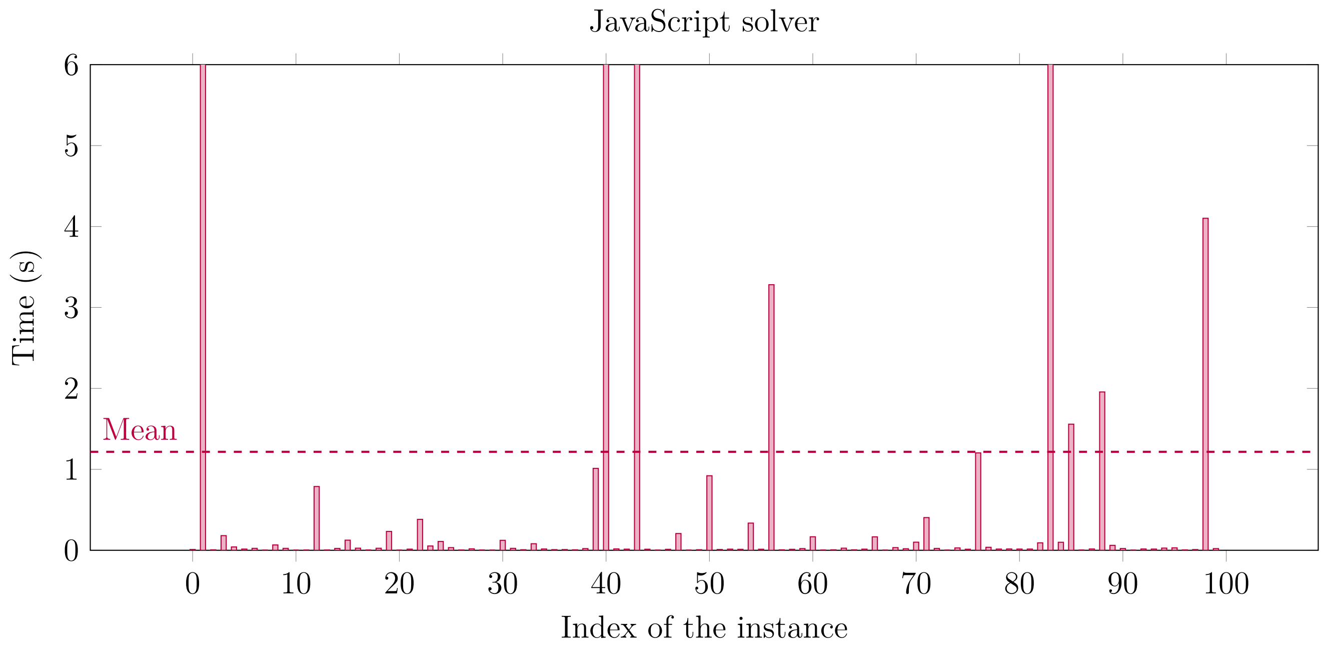
MiniZinc model (Gecode, randomized search + constant restart)



MiniZinc model (Chuffed)




ASP model
Time (s)
Mean
Index of the instance
0
1
2
3
4
5
6
0
10
20
30
40
50
60
70
80
90
100


JavaScript solver
Time (s)
Mean
Index of the instance
0
1
2
3
4
5
6
0
10
20
30
40
50
60
70
80
90
100

## 5.2 Verifying IcoSoKu's solvability

Finding a solution to every possible instance of IcoSoKu is feasible using our MiniZinc model, even using the randomized search strategy, by being careful about the simmetries of the problem. In fact, not every instance corresponding to a permutation of the elements in $\{1, \ldots, 20\}$ needs to be solved:

- We can impose the capacity of vertex A to be equal to 1, because if it is not the case we can always rotate the icosahedron (or rename the vertices) to have this situation, solve the modified instance and rotate (or rename) it back to have the solution.

- Having fixed the capacity of vertex A, the icosahedron can still be rotated on its A-L vertical axis so for the same reason we can impose the capacity of vertex B to be less than the capacities of vertices C, D, E and F.

- We can even exploit the simmetry of the icosahedron combined with the (very suspicious) simmetries in the tile configuration of IcoSoKu. In fact, by flipping the icosahedron horizontally with the plane of reflection that goes through A, L and B, shown in Figure 12, we can obtain a mirrored icosahedron, exactly imposable over the original, such that every IcoSoKu solution corresponds to a solution to its mirrored version using the same tile configuration (because the tiles with at least two equal weights remain the same tile when mirrored and the tiles with three different weights all have their mirrored counterpart). Instead of mirroring the vertices of the icosahedron we can just mirror their capacities: under this perspective every instance considered above (vertex A has capacity 1 and vertex B's capacity is less than the capacities of B,C,D and F) has its mirrored version still corresponding to the restrictions we have just imposed, and viceversa. For each pair we need to solve only one, for example the first in lexicographical order.

This leaves us with 7 possible capacities that vertex B can assume and

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

instances to check.

So a small C program (not shown here) has been written to generate all these instances (printing the corresponding lines specifying `cap`) and the following command has been run (first removing from `input-ico.dzn` the

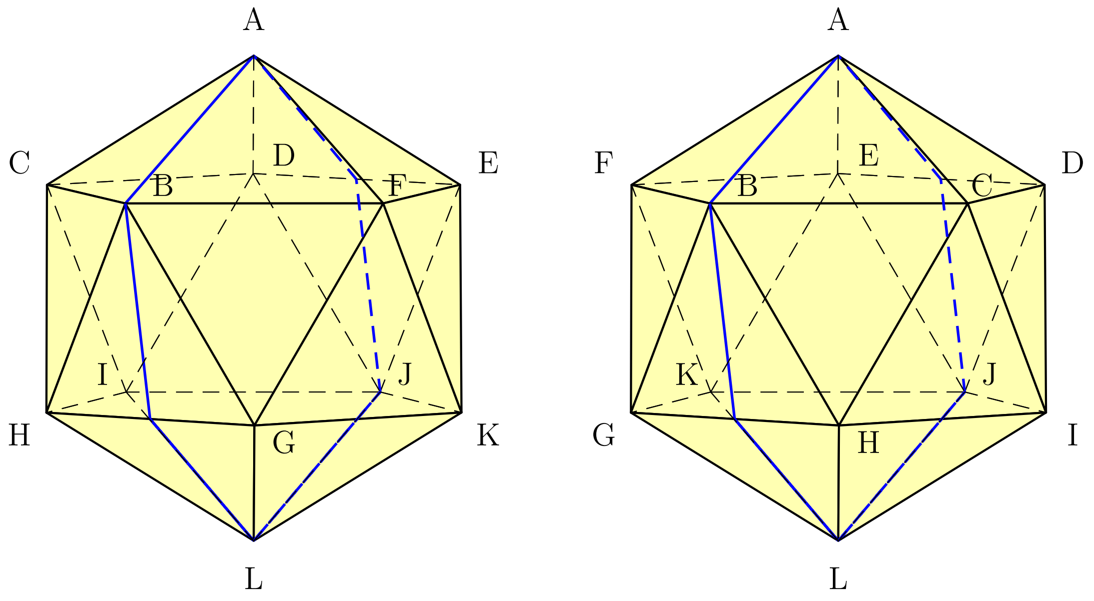


Figure 12: The icosahedron (left) intersected with the plane through A, B and L (in blue) and its mirrored version (right).

line about `cap`) to solve at the same time as many instances as our machine could handle:

```
$ ./good_instances | \
  xargs -P 0 -n1 --delimiter '\n' \
  minizinc randomsoku.mzn input-ico.dzn -D
```

The program terminated in approximately five days on a machine mounting an Intel i5-7400 clocked at 3.30 GHz, verifying that every IcoSoKu can be solved and also that the randomized search reliably finds a solution.

## 6 Conclusions

3SoKu, the generalized version of IcoSoKu that stays true to the original game by having as playing field a polyhedron with triangular faces, is NP-complete, even when imposing that the playing field is well formed and that the faces are of the same size and shape. The problem we reduced to 3SoKu is the Partition problem, one of Karp's 21 NP-complete problems of 1972. Some interesting questions remain unanswered:

- is 3SoKu still NP-complete if we impose the playing field to be a deltahedron (a polyhedron with equilater triangles as faces) that can be constructed in 3D space?
- Partition admits a pseudo-polynomial time algorithm (see [7, p. 223]), is it the case for 3SoKu as well?

The best approaches we have found to solve instances of IcoSoKu use our MiniZinc model:

1. with Chuffed as the solver;
2. with Gecode as the solver and a randomized search, exploiting the fact that probably there are a high number of IcoSoKu solutions (many millions!).

Even if the second method is not efficient for every class of instances of 3SoKu, Chuffed's lazy clause generation probably is the right tool to prune very efficiently the search tree of the problem at hand.

A profound combinatorial reason for the presence of a solution to every instance of IcoSoKu has not been found, but thanks to our models, to the simmetries of the icosahedron and to the (suspicious) simmetries in the tile configuration we managed to solve every possible instance, showing that such a computation is feasable with common hardware. On the other hand we do not how many different solutions there usually are, since counting the solutions to just one instance of IcoSoKu requires a lot of memory due to their high number.

## References

[1] *IcoSoKu puzzle*, Recent Toys Int., accessed 30 March 2020, `https://www.recenttoys.com/recent-toys-icosoku-puzzle/`

[2] *Andrea Mainini's profile*, Spiele-Autoren-Zunft e.V. Game Designers Association, accessed 30 March 2020, `https://www.spieleautorenzunft.de/authors-details.html?member=241`

[3] *IcoSoKu online solver*, Marzio De Biasi's site, accessed 30 March 2020, `http://www.nearly42.org/games/icosoku-solver/`

[4] *Convex Polyhedra with Regular Faces.* Norman W. Johnson, Canadian Journal of mathematics, 1966.

[5] *Computational Complexity.* Christos Papadimitriou, 1994. Addison Wesley.

[6] *Reducibility among Combinatorial Problems.* Richard M. Karp, 1972. Complexity of Computer Computations. The IBM Research Symposia Series. Springer, Boston, MA.

[7] *Computers and Intractability: a Guide to the Theory of NP-Completeness.* Michael R. Garey, Davis S. Johnson, 1979. W. H. Freeman and Company

[8] *The MiniZinc Handbook 2.3.1.* Peter J. Stuckey, Kim Marriott and Guido Tack.

[9] *Potassco User Guide (Second edition)*, version 2.2.0. Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, Sven Thiele and Philipp Wanko, University of Potsdam.