A Cardinality Set Representation for MiniZinc

Sullivan Bitho

Supervisors:
Guido Tack, Jip J. Dekker

July 2023

Abstract. When modelling a problem using a constraint modelling language, it feels very natural to use sets. These problems includes the social golfer, scheduling, graph partitioning and much more. Using these high-level data structures considerably reduces modeling time and improves expressiveness and readability of the model. But many solvers on the market do not natively support making decisions about sets. This means that users have to manually translate sets into structures that the solvers can handle, like arrays of Boolean decisions. In this report, we will present a library for `MiniZinc` that automatically encode sets into a new promising representation.

Contents

Acknowledgement

I would like to express my sincerest gratitude to everyone who helped me during my internship. I feel very privileged to have been able to work with one of the best researchers in the *Constraint Programming* communities. I would like to thank Jip Dekker for always being there for me and sharing valuable suggestions. Guido Tack who always found time to meet with me, share his expertise and make suggestions. Charles Prud'homme for very interesting discussions on Choco and CP solvers. All the members of the *OPTIMA* team at Monash for their welcome, kindness and help (and for the fascinating discussions during lunches and coffee breaks). A particular thanks to Eric Monfroy, who made me love constraint programming. I would also like to thank Evgeny Gurevsky for making my internship at Monash University possible. Finally, Anthony Przybylski for being an amazing *ORO* master supervisor.

1 Introduction

The efficiency of solvers depends largely on the input model, and in particular on the choice of variable types and constraints used. In this project, we focused on models involving set decision variables: we investigate encoding these variables efficiently so that solvers could solve the model as quickly as possible. This report is organized as follow: we start by introducing literature review on constraint programming in Section 2. We then present our theoretical contribution in Section 3, before showing our numerical experiments in Section 4. Finally, Section 5 conclude our work. An internship overview is available in annex A.

The following vocabulary, notation and functions are used.

Vocabulary

– We have used the terms "encoding" and "representation" interchangeably.

Notations

– The element at index i in an array A will be denoted A_i.
– Let s be a set of elements. s_i denotes the i-th element of the array x, translated from s by a certain function f. This translated array is always considered in strictly ascending order.
– $a..b$ represent the closed interval $[a, b]$ and is used in this project to represent domain such that $e \in a..b \Leftrightarrow a \leq e \leq b$.

Functions In this report, we assume that we have the following functions:

– $card(s)$ returns the cardinality of the set s.
– $index\_set(t)$ returns the set of indexes of t. t can be an array or a set.
– $lb(s)$ and $ub(s)$ are respectively the function returning the lower bound and the upper bound of the set s.
– $occ(v, A)$ returns the number of occurrences of the value v in the array A.

2 Literature review

2.1 Constraint Programming

Constraint Programming (*CP*) [1] is a programming paradigm aimed at solving hard search problems. Constraint Programming integrates mathematical techniques, artificial intelligence and operation research techniques to tackle a given problem. The given problem is represented in a short, simple program, that states the variables and the constraints of the problem, called a *model*. Constraints are restrictions imposed on variables and the way in which they are linked. One type of real-world problem of particular interest to us here are sub-problems of operations research: constraint satisfaction problems (*CSPs*). These problems are considerably difficult to solve due to their combinatorial nature, involving a set of variables subject to constraints. The main objective of *CSPs* is to find an assignment of values to the variables that satisfies all the given constraints. Formally, CSP can be described as follows:
Let be a set of variables $X = \{X_1, X_2, ..., X_n\}$, where each X_i can take values in a domain D_i. Let be a set of constraints $C = \{C_1, C_2, ..., C_m\}$, where each C_j specifies a relationship between a subset of variables. The objective is to find an assignment of values to the variables X in such a way as to satisfy all the constraints C at the same time. This value assignment must respect the D_i domains for each variable X_i.

Early versions of constraint programming comes from *Constraint Logic Programming*, which was an extension of general *Logic Programming*. Logic programming is based on predicates, logical conjunctions, and backtracking search. Constraints can be seen as relations or predicates, a set of constraints can be viewed as the conjunction of the individual constraints, and backtracking search is a basic methodology for solving a set of constraints. Logic based implementation allows solvers to use the constraints efficiently to reduce the search space and make inferences. Those makes constraint solving very compatible with logic programming. We should note that research has been carried out into the integration of constraints in procedural and object-oriented languages [2]. A high-level declarative language based on logic (such as `ECLiPSe` [3], `MiniZinc` [4]) is particularly well suited to constraint programming. Indeed, high level declarative language allows programmers to specify the properties of a solution, rather than describing the steps required to calculate/create a solution, as in other paradigms.

2.2 Solving Constraint Satisfaction Problem

Most of the time, *CSP* are solved in a two phase algorithm, by iterative *searches* and *propagations.*

2.2.1 Search

Search consists of assigning values to the decision variables, from their domains of possible values. The search often uses a tree data structure to efficiently *backtrack* (i.e return) to a previous set of assignments, when a *branch* (i.e. alternative) is identified as *unsatisfiable* (i.e. violating one or more constraints). Such a tree search is called *Branch and Prune* [5].

Searching can solve a CSP, but used on its own, it is a very costly operation. Certain resolution techniques can considerably reduce search space and time. One such technique, presented below, is constraint propagation.

2.2.2 Constraint Propagation

Constraint Propagation [6] is a technique that can be used to solve a constrained problem. Constraint Propagation uses *propagators* which are usually fix-point algorithms: eliminations of impossible values in decision variables domains are computed as long as new information have been obtained by the previous propagation. Propagators use *filtering* techniques like *Arc Consistency* or *Bound Consistency* techniques to infer impossible values. An *Arc Consistency* algorithm guarantees that for each value of one variable, there is at least one compatible value in the domain of the other variable, taking into account the constraints linking them. A *Bound Consistency* algorithm reduces the intervals of valid values by propagating information from the bounds specified by the constraints to the domains of the variables. The propagated information leads to 2 possible scenarios: 1) - reduction of variables domains. 2) - unsatisfiable branch. Once the propagation has reached a fix-point, the search process is called again. The search process often leads to similar scenarios where assignments of a subset of variables lead to an unsatisfiable node, wasting time in the resolution. In Section 2.4 we describe a technique that avoids the repetition of such scenarios.

One particular class of constraint that have been used in this project, called *global constraints*, can significantly increase the efficiency of constraint propagation. The following Section gives an overview of some of the most commonly used ones.

2.3 Global Constraints

Global Constraints [7] are highly expressive constraints that capture intricate relationships between a non-fixed number of variables from the problem. Unlike primitive constraints, global constraints express relationships between multiple decision variables, *simultaneously.* Propagators for such constraints use state of the art algorithms based on graph theory (Max-Flow-Min-Cost), automata and regular expression, to identify patterns and substructures to leverage high-level knowledge, that often lead to a very fast convergence, allowing for more efficient pruning in the search space. Here are a few global constraints of particular interest to this project, The formalism used gives the domain set using tuples.

- `element`: The `element` constraint [8] constraints a relation between a value v, a variable index i, and an array of integers a, with $a = [x_1, x_2, ..., x_n]$, such that $x_i = v$. More formally:

$$\texttt{element}(i, v, x_1, x_2, ..., x_n) = \{(e, f, d_1, ..., d_n) \\ \mid e \in D(i), f \in D(v), \forall_{j \in 1..n}\ d_j \in D(x_j), f = d_e\}$$

- `all_different`: The `all_different` constraint is certainly the most studied global constraint ([9], [10] [11], [12], [13]). Let $x_1, x_2, ..., x_n$ be variables. The `all_different` global constraint ensure that the x_i are all different from each other. More formally:

$$\texttt{alldifferent}(x_1, x_2 ..., x_n) = \{(d_1, ..., d_n) \\ \mid \forall_{i \in 1..n}\ d_i \in D(x_i), \forall_{i \neq j}\ d_i \neq d_j\}$$

  Solvers that implement `all_different` will represent the constraints of the model as a *bipartite graph* ([14] [15]). State-of-the-art bound-consistency algorithms either identifies *Hall intervals* [13], or use an arc-consistency algorithm that focus on simpler structure of the graph. The time complexity of those algorithm are $O(n * log(n))$.

- `global_cardinality`: The global cardinality constraint, abbreviated `gcc`, is a generalization of `all_different`. It counts the number of occurrences of each variable from an array, in another array of variables. More formally, let a be an array of variables such that $a = [x_1, x_2, ..., x_n]$, each of those variables having their domains contained in the set of value $\{v_1, v_2, ..., v_m\}$. Let $\{c_{v_1}, ..., c_{v_m}\}$ be integer variables that count the number of occurrences of v_i in a, $\forall i \in \{1, .., m\}$. Then:

$$\texttt{gcc}(x_1, ..., x_n, c_{v_1}, ..., c_{v_m}) = \{(w_1, ..., w_n, o_1, ..., o_m) \\ \mid \forall_{j \in 1..n}\ w_j \in D(x_j), \forall_{i \in 1..m}\ occ(v_i, (w_1, ..., w_n)) = o_i, o_i \in D(c_{v_i})\}$$

  The `gcc` global constraint has proved particularly powerful in our encoding of set operations, as discussed below.

Beldiceanu et al. have presented a catalog of global constraints [16] where each constraint is explicitly described.

2.4 Lazy Clause Generation

Lazy Clause Generation (*LCG*) [17], is a technique used in modern constraint solvers (such as *Chuffed* [18], [19]) and SAT solvers to improve propagation efficiency. It involves the generation of additional clauses during the search process, particularly in response to conflicts encountered during the solving process. These additional clauses are generated “on the fly”, rather than trying to calculate all possible clauses in advance before the search begins, which would be impossible.
When a constraint solver encounters a conflict, it analyzes the *cause* of the conflict by examining the conflicting constraints and their interactions. From this analysis, the solver may ”learn” new clauses that capture the reason for the conflict. The *LCG* solver will then stores the *cause* in a separate database: an *explanation tree*. The solver then backtracks to a previous point in the search, often called a ”restart,” and resumes the search from there. During subsequent search steps, if the solver encounters a similar situation that could lead to the same conflict, it checks the clause database for any learned clauses that could help resolve the conflict. If such clauses exist, they are generated and added to the current set of constraints. This process is called “lazy generation” because clauses are generated only when needed.
This avoids generating a large number of redundant or unnecessary clauses reducing memory usage and solving time significantly.

2.5 MiniZinc

`MiniZinc` [4] [20] is an open-source constraint modeling language used for modeling combinatorial optimization. `MiniZinc` automate the solving of CSPs: users (i.e *modelers*) translate a problem into a model written in a specific language. This model is then compiled into an instance than solvers can solve. Finally, the answers is translated back to the user. `MiniZinc` is a declarative language such that one just need to declare constraints and relationships between variables. This makes it easy to express complex models in a concise and readable way. `MiniZinc` allows the modeler to use many solvers without the need to rewrite the model for each new solver, which facilitates experimentation. Our work concerns this last point: we want users to be able to use `MiniZinc`’s set structures without having to wonder whether their solver handles sets or not.

2.6 Sets in constraint programming

Significant research work has been devoted to the integration of sets in Constraint Programming. C. Gervet proposed *Conjunto*[21], a CLP language embedding sets. It is the first language to represent set variables by lower and upper domain bounds, with set inclusion as a partial ordering. In other words, set variables range over a finite set domains (set of sets), defined by a greatest lower bound set, and a least upper bound set (using linked list data structures). This representation is natural, expressive, and leads to efficient propagation using consistency techniques such as interval reasoning

([22],[23]). A key component of *Conjunto* is that it represents the cardinality of a set as a graded function f that satisfies $s_1 \subseteq s_2 \Rightarrow f(s_1) \leq f(s_2)$ for two sets s_1, s_2. But *Conjunto* lacks cardinality-level inferences, which are essential for a range of CSPs as it can be used to deduce more rapidly the non-satisfiability of a set of constraints. *Cardinal* [24] is a general finite sets constraint solver proposed after *Conjunto*, that actively uses information on set cardinality. It is a set of rewriting rules on a constraint store, that uses cardinality informations for set operations and functions. Constraints propagation on set bounds and set cardinality are made to efficiently prune the search space.
F. Lardeux, E. Monfroy et al. proposed to use the expressiveness of CSP languages and the efficiency of the SAT solver by automating the encoding of set constraints directly into SAT clauses [25].

3 Theoretical contribution

In this Section, we start by introducing some definitions regarding set variables and recall the existing set encoding. We then propose two new possible set encodings for set variables, with their corresponding operations and functions.

3.1 Set variables

Definition 3.1. A set domain D is a set of sets containing all possible set values of a set variable s.

Figure 2 illustrate a set domain D_x with 4 possible sets.

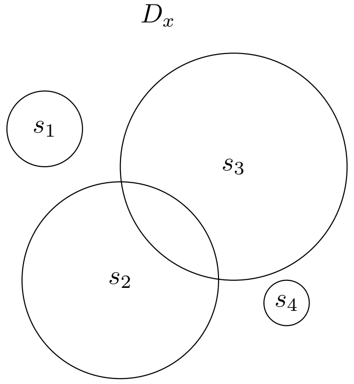


Figure 2: A visualization of a set domain D_x with its set elements s_i

The enumeration of all possible sets in the set domain grows exponentially with the set cardinality. In *Conjunto* [21] and many solvers, sets are represented by a lower and an upper bound set, so that only the (resp.) certain and possible values of the set are informed. This least upper bound set then defines an approximation of the set domain as the convex enclosure of the domain definition: a *set interval* [26].

Definition 3.2. A set interval I defines a lattice of sets, partially ordered by set inclusion, such that

$$I = [glb, lub]$$

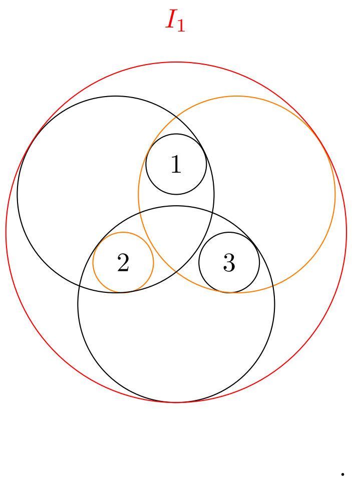
with $glb = \bigcap_{i=1}^{n} D_i$ the intersection of all sets inside D, and $lub = \bigcup_{i=1}^{n} D_i$ the union of all sets inside D.

The interval set is composed of all the sets "between" glb and lub i.e:

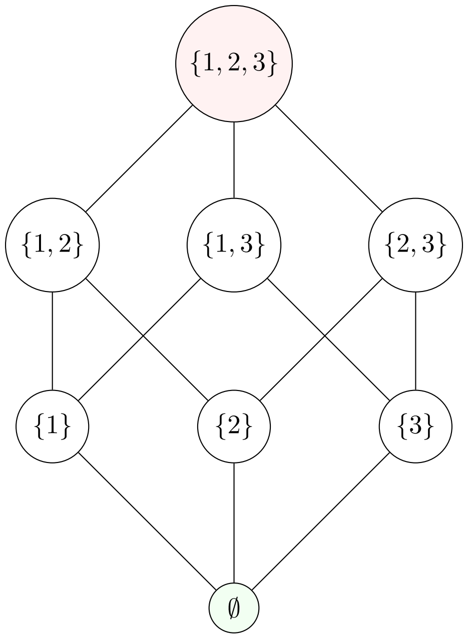
$$I = (\mathcal{P}(lub) \setminus \mathcal{P}(glb)) \cup glb$$

with $\mathcal{P}(s)$ being the power set of the set s. Note that in MiniZinc, a set variable is declared only using its lub, setting glb to the empty set $\{\}$ by default. A `MiniZinc` set variable thus range over $I = \mathcal{P}(lub)$.

As an example, let s_1 be a set variable belonging to the set domain $D_1 = \{\{2\}, \{1,3\}\}$. We approximate D_1 with the set interval $I_1 = [\{\}, \{1,2,3\}]$. In Figure 8 we represent I_1 such that $glb_1 = \{\}$ (green node), $lub_1 = \{1,2,3\}$ (red node). The two Figures below shows all possible sets that will be consider to represent s_1 in models that use bounded domain to represent sets. The Figure 8 illustrate two different vizualisation of I_1: on the left, a Venn diagram visualization, on the right, a lattice visualization.



(a) set variable s_1 range over I_1, the convex enclosure of D_1



(b) Lattice representation of I_1

Figure 3: Two different vizualisations of I_1

The cardinality of a set variable is a decision variable itself. It ranges from the cardinality of the *glb* set, to the cardinality of the *lub* set.

Definition 3.3. The cardinality C of a set variable s ranges in the integer domain $C_{min}..C_{max}$, where $C_{min} = card(glb(s))$ and $C_{max} = card(lub(s))$.

The *Conjunto* representation of a set variable is as follow:

- a greatest lower bound *glb*
- a least upper bound *lub*
- a lower bound cardinality C_{min}
- an upper bound cardinality C_{max}

In what follows, we present two different ways of encoding sets when no set definition exist.

3.2 Set variable encoding

A set is a collection of unique unordered elements. This means that all permutations of set values are symmetrical representations. As we only need one, a first interesting idea to encode sets is to use arrays. Indeed encoding sets using arrays will allow us to break symmetric representations, as each value can be constrained to a fixed index.

We first present an existing set encoding, using an array of Boolean variables, before proposing our representations, using arrays of integer variables.

3.2.1 Array of Boolean variables (*BoolSet*)

Intuitively, representing sets using arrays of Boolean variables seems a good idea since 1 - Boolean operations are very cheap and 2 - solvers will eventually translate high level constraints into low level *clauses*. A *clause* is a propositional formula formed from a finite collection of literals (atoms or their negations) and logical connectives. So using Boolean constraints will always facilitate solvers work. The array contains a Boolean variable for each possible value that could be present in the set. More formally, we define the array of Boolean variables representation as follow:

– Given a set variable s with a domain D,
– the function f_B, which encode a set into an array of Boolean variables, and f_B^{-1} its reverse function.
– Then let x be the array of Boolean variables encoding s, and we define

$$\begin{aligned} f_B(s) &= x \\ f_B^{-1}(x) &= s \\ x_i &= true \quad \Leftrightarrow i \in s, \forall i \in 1..card(s) \\ x_i &= false \;\; \Leftrightarrow i \notin s, \forall i \in 1..card(s) \end{aligned}$$

To illustrate the *BoolSet* representation, let consider the set variable s_2 with values ranging over its domain $D_2 = \mathcal{P}(\{1, 2, 3\})$ and a variable cardinality $C_2 \in \{0, 1, 2, 3\}$ ($C_{min} = 0, C_{max} = 3$). Below is the lattice corresponding to all possible arrays of integers representing the set variable s_2.

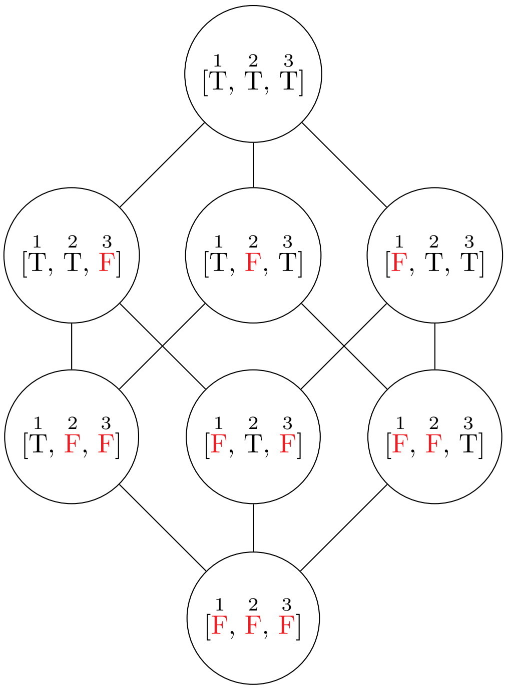


Figure 4: Lattice of all possible sets for s_2 represented as arrays of booleans

This encoding of sets is used by `MiniZinc` and other constraint modeling languages that incorporate sets like `GeCode` or `Choco`. We will call it *BoolSet* representation in the following. The advantage of this Boolean representation is that set operations such as Union, Intersection and others are very efficient, since we can simply use the logical operators AND and OR.
But this representation has its limits: let's consider a set s with a large domain least upper bound, and a small fixed cardinality, i.e. $|lub| = 1000$ and $|s| = 3$. In this situation, we have no choice but to create an array containing thousand Boolean variables to allow all possible representations of the set, even though only 3 elements will be in the final set. Any constraint that uses s now has to be defined to consider 1000 variables. So, for example, creating the sum of the element in s creates a very large expression. To cope with this scenario, we propose another representation of the set: an array of integer variable.

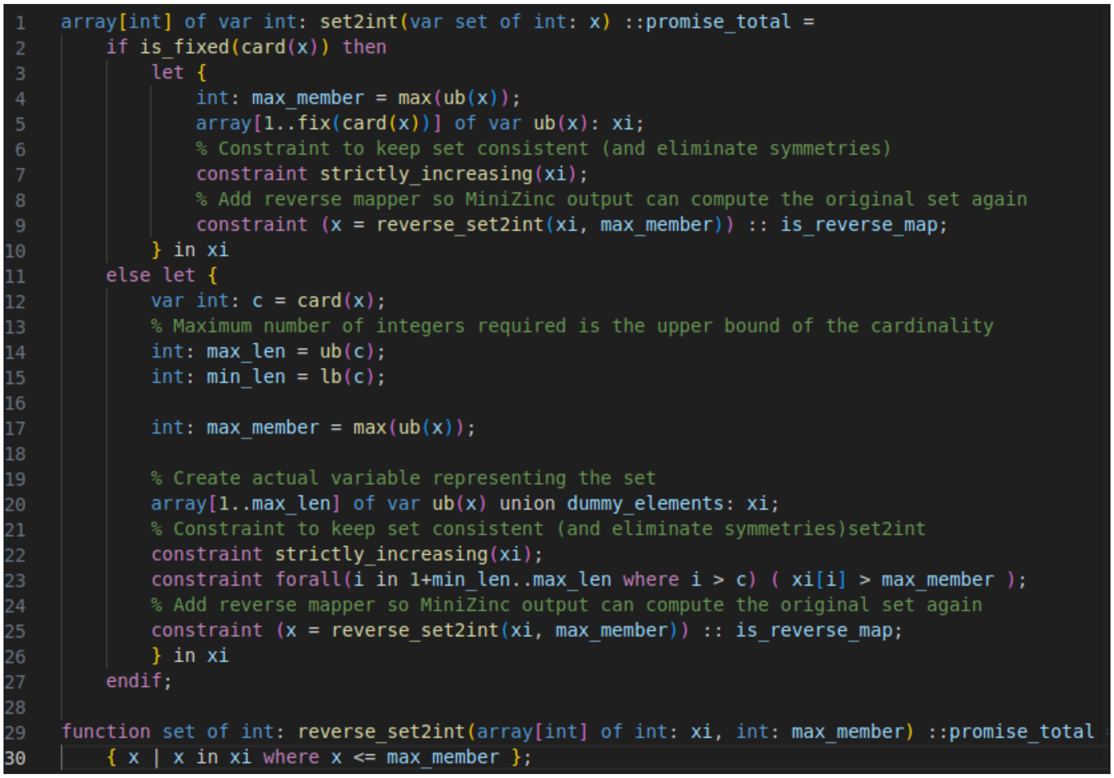
3.2.2 Array of integer variables (*CardSet*)

In this representation, we consider an array xi the size of the cardinality C of the variable set s. If the cardinality is variable, the array will be the size of the maximum possible cardinality C_max. The elements of xi are the elements of s, plus possibly *dummy* elements depending on the decided cardinality. Indeed, if the decided set has a cardinality smaller than its maximum possible cardinality, then the extra positions are filled with *dummy elements*. xi encodes s using the following representation: $xi = [glb(s)_i, x_j \mid i \in 1..|glb(s)|, x \in lub(s) \cup dummies, j \in (i+1)..|lub|]$. More formally, we define the array of integers representation as follow:

- Given a set variable s with a domain D and C_{min} and C_{max} respectively the minimum and maximum possible cardinality of s,
- the function f_C that encodes a set into an array of integer variables, and f_C^{-1} its reverse function.
- We define $m = C_{max} - card(s)$,
- and d_i, the dummy value for element y_i.
- Then let y be the array of integers encoding of s,
- $card(s)$ be the integer variable encoding the cardinality of s, such that

$$\begin{aligned}
f_C(s) &= y \\
f_C^{-1}(y) &= s \\
card(s) &\in C_{min}..C_{max} \\
y_i &= min(s \setminus \bigcup_{j=1}^{i-1} s_j), \forall i \in 1..card(s) \\
y_j &= d_k, \forall j \in (card(s)+1)..C_{max}, \forall k \in 1..m
\end{aligned}$$

We will take this opportunity to show you what the MiniZinc code looks like. For the sake of clarity, we will assume that we have an appropriate set of dummy elements. We renamed f_C `set2int` and f_C^{-1} `reverse_set2int`:



```
array[int] of var int: set2int(var set of int: x) ::promise_total =
    if is_fixed(card(x)) then
        let {
            int: max_member = max(ub(x));
            array[1..fix(card(x))] of var ub(x): xi;
            % Constraint to keep set consistent (and eliminate symmetries)
            constraint strictly_increasing(xi);
            % Add reverse mapper so MiniZinc output can compute the original set again
            constraint (x = reverse_set2int(xi, max_member)) :: is_reverse_map;
        } in xi
    else let {
        var int: c = card(x);
        % Maximum number of integers required is the upper bound of the cardinality
        int: max_len = ub(c);
        int: min_len = lb(c);

        int: max_member = max(ub(x));

        % Create actual variable representing the set
        array[1..max_len] of var ub(x) union dummy_elements: xi;
        % Constraint to keep set consistent (and eliminate symmetries)set2int
        constraint strictly_increasing(xi);
        constraint forall(i in 1+min_len..max_len where i > c) ( xi[i] > max_member );
        % Add reverse mapper so MiniZinc output can compute the original set again
        constraint (x = reverse_set2int(xi, max_member)) :: is_reverse_map;
        } in xi
    endif;

function set of int: reverse_set2int(array[int] of int: xi, int: max_member) ::promise_total
    { x | x in xi where x <= max_member };
```

Figure 5: MiniZinc `set2int` redefinition

To illustrate the *CardSet* representation, let consider the set variable s_2 again (defined in Section 3.2.1. Below is the lattice corresponding to all possible arrays of integers representing the set variable s_2.

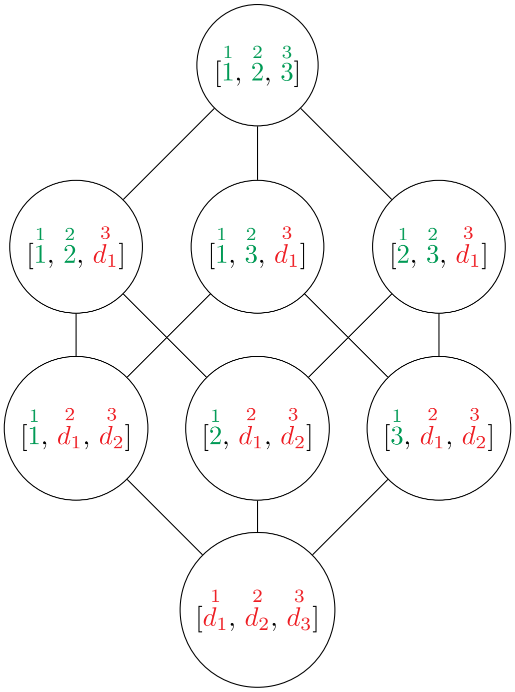

$[\overset{1}{1}, \overset{2}{2}, \overset{3}{3}]$
$[\overset{1}{1}, \overset{2}{2}, \overset{3}{d_1}]$
$[\overset{1}{1}, \overset{2}{3}, \overset{3}{d_1}]$
$[\overset{1}{2}, \overset{2}{3}, \overset{3}{d_1}]$
$[\overset{1}{1}, \overset{2}{d_1}, \overset{3}{d_2}]$
$[\overset{1}{2}, \overset{2}{d_1}, \overset{3}{d_2}]$
$[\overset{1}{3}, \overset{2}{d_1}, \overset{3}{d_2}]$
$[\overset{1}{d_1}, \overset{2}{d_2}, \overset{3}{d_3}]$

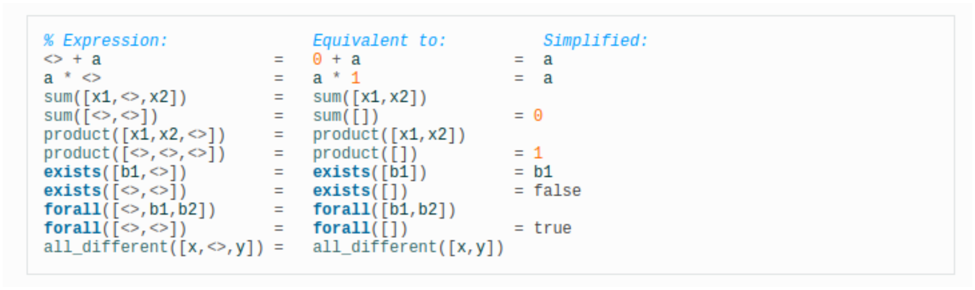
Figure 6: Lattice of all possible sets for s_2 represented as arrays of integers

The power of this representation is intrinsically linked to the constraints imposed on the cardinality of the set variable: the tighter the cardinality bounds, the least dummy elements we need, the more efficient this representation.

Dummy elements constituted a crucial point of attention in this project. Should we encode the dummy elements as integers outside the set's value domain, or should we use a single value, which do not belong to any set in the involved set domains? For the latter one, `MiniZinc` defines a special types at hand: *absent* types.

Absent type in Minizinc
An *option type decision variable* or *absent* type [4] represents a decision that has another possibility $\top$, represented in MiniZinc as $<>$ indicating the variable is absent.

Comparison operators return *true* if any of their arguments is absent. For instance, $3 \leq <>$ is *true*, as is $<> \leq 3$. However, note that equality between option type expressions is only *true* if both expressions have the same optionality: $<>=<>$ is *true*, but $3 =<>$ is *false*. This means that using `strictly_increasing`(*xi_opt*) alone will give



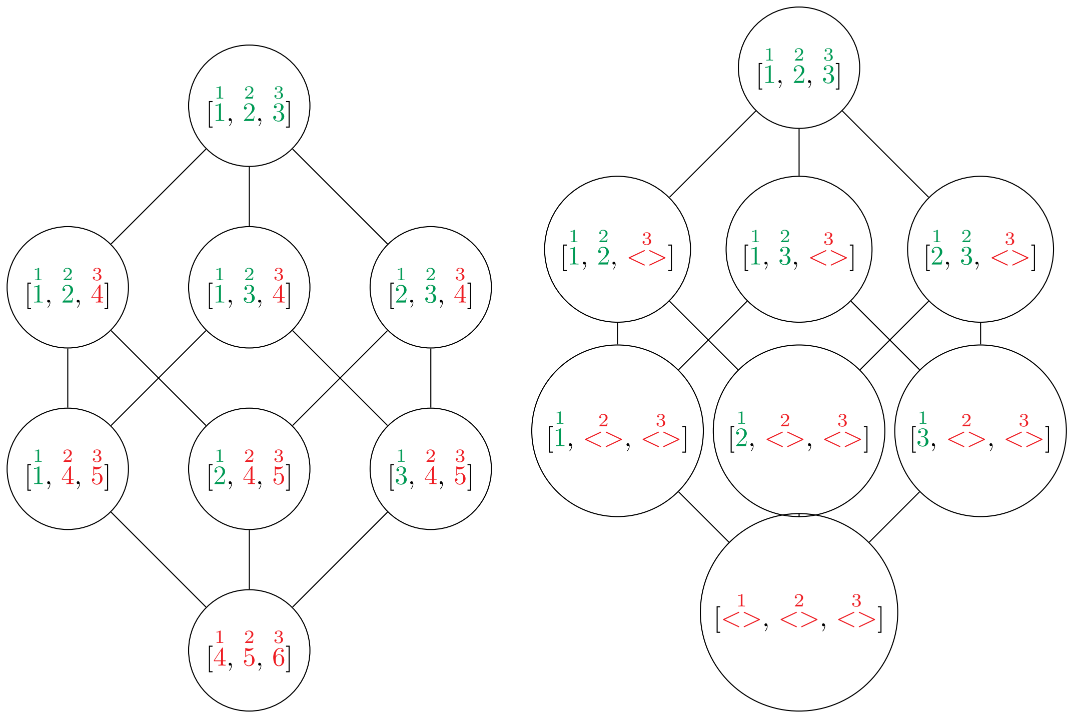
```
% Expression:               Equivalent to:        Simplified:
<> + a                   =  0 + a              =  a
a * <>                   =  a * 1              =  a
sum([x1,<>,x2])          =  sum([x1,x2])
sum([<>,<>])             =  sum([])            = 0
product([x1,x2,<>])      =  product([x1,x2])
product([<>,<>,<>])      =  product([])        = 1
exists([b1,<>])          =  exists([b1])       = b1
exists([<>,<>])          =  exists([])         = false
forall([<>,b1,b2])       =  forall([b1,b2])
forall([<>,<>])          =  forall([])         = true
all_different([x,<>,y])  =  all_different([x,y])
```

Figure 7: Caption

symmetrical solutions. We need to ensure that *xi_opt* values with indexes less than or equal to the cardinality of the set C, must appear, and that values with indexes greater than C must be absent, which we can easily do in `MiniZinc`.

In Figure 7 we give few other examples of how different functions and operators treat option types.

Both dummy encodings have their pros and cons. The appeal of using different integers as dummy elements comes from the possibility of using strictly increasing constraints, which breaks symmetric representations of the set. However, this also means that we need to keep a close eye on our dummy elements to ensure that none of them encroach on the real values of another set during set operations (more on this in Section 4.3). On the other hand, representing a dummy element as an absent value would facilitate the encoding of set operations, since no occurring value can take on the absent value. However, such a dummy element must be taken into account when using global constraints such as all_different or global_cardinality.
To experiment with these two representations, we have encoded them both. We called the encoding with n different integers for the n extra members *CardSet_nem*. The encoding with n absent values is called *CardSet_opt*. In Figure **??**, we illustrate set interval lattices for the corresponding *CardSet_nem* and *CardSet_opt* encodings.



(a) *CardSet_nem* (b) *Cardset_opt*

Figure 8: Lattice view of *CardSet_opt* and *CardSet_nem* set variable domain representation for s

Note that all dummy elements are at the end of the array. The `strictly_increasing` global constraint ensures that all the elements from xi are in a strictly increasing order.

3.2.3 Advantage of *CardSet* over BoolSet

In this Section we will see in which scenarios *CardSet* can be more performant than BoolSet. *BoolSet* power comes from its Boolean operations, but, unlike CardSet, it does not take advantage of cardinality information. We will see two scenarios: when set variables has a fix cardinality and when set variables has a bounded cardinality.

Let us start with the scenario with the tighest cardinality bound first: a fixed cardinality. As an example, let us compare an array of booleans (Table 1 1) with an array of integers (Table 2 2) to represent the set variable s_3 ranging from 1 to 1000, with a fixed cardinality of 3:

1	2	...	i	...	1000
b_1	b_2	...	b_i	...	b_n

Table 1: s_3 using a *BoolSet* representation

1	2	3
i_1	i_2	i_3

Table 2: s_3 using a *CardSet* representation

In the *BoolSet* representation, we need an array of 1000 booleans. In the *CardSet* representation, we only need an array of 3 integers i ranging from 1 to 1000.

Let us now see the situation where the cardinality is not fixed, but bounded. In this case, we need to introduce dummy elements for the *CardSet* representation, while the *BoolSet* representation remains unchanged. To illustrate this scenario let us consider the set variable s_4 with values ranging from 1 to 1000, and a cardinality ranging from 2 to 4.

1	2	...	i	...	1000
b_1	b_2	...	b_i	...	b_n

Table 3: s_4 using a *BoolSet* representation

1	2	3	4
i_1	i_2	i_3 or d_1	i_4 or d_1 or d_2

Table 4: s_4 using a *CardSet* representation

In this example, the upper bound cardinality is small, so *CardSet* is advantaged. But as the upper bound on the cardinality of the set variable increases, we are going to end up with a lot of dummy elements, which should reduce CardSet's performance. The fundamental question to be answered is: When should we encode a set decision variable using CardSet rather than using BoolSet? Experiments will help us settle our various hypotheses.

3.3 CardSet constraints encoding

Set operations and functions are the building blocks of set constraints. In this section we propose algorithms to encode them. We begin by exposing some difficulties experienced with the programming language used (MiniZinc). We then describe the algorithm used for the main set operations and functions.

3.3.1 Programming in MiniZinc: difficulties

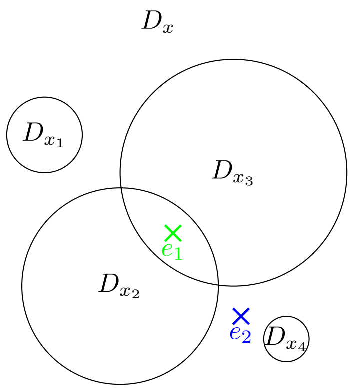
`MiniZinc` is a constraint modeling language well suited to high-level, expressive constraint declaration. But in this project, we aim to create a library *for* `MiniZinc` *in* `MiniZinc` language, using functions, loops and control structures, which is very different from simply declaring a model. `MiniZinc` is mainly based on declarative programming, which makes it very different from traditional programming languages for creating imperative functions. Debugging the functionality of our library proved particularly challenging.: because the variables are decision variables, they are not fixed until the solver has completed its search and proposed a solution. We cannot simply plot the variable content to check whether our function is doing a correct job or not. In addition, `MiniZinc` incorporates concepts such as *root*, *positive*, *negative* and *mixed* contexts for expressions, together with *promise total* function annotations (indication to the solver that the function has a defined result for all possible input values in its domain) which must be understood to understand the effects on the compilation and the compiler log errors. Finally, the usage of `MiniZinc`'s features in new ways uncovered new bugs in the `MiniZinc` compiler, which regularly made me ask whether the bug came from my implementation or from the compiler.

Many attempts at efficient encoding were made during the course of the project. At first, I was confident I could encode the different set operations and functions within a couple of weeks. In the end, it took me most of my internship. This is probably because I am new to `MiniZinc` and the use of global constraints, and that I was stuck in my old procedural programming ways. During the first semester, we had a whole subject dedicated to global constraint. At the beginning of my internship, I have watched and exercised on them with `MiniZinc` courses. But despite that, it took me a long time to get global constraints mindset and reflex, probably because I prefer not to use functions I do not understand. I knew they are convenient, but I did not realize how efficient they are. Friday May 5 marked a turning point in my constraint programming skills. This day, we had our weekly meeting: I presented Guido Tack and Jip Dekker with my work on encoding the intersection of sets. My idea was to use one cursor per array (cursor i for array A and cursor j for array B) to identify common values between A and B. In my implementation, I have taken advantage of the fact that the arrays are in strictly increasing order: if we cannot find an $A[i]= B[j]$ such that $B[j] < A[i]$, then we know that $A[i]$ cannot be in $f_C(A) \cap f_C(B)$, and we do not need to continue iterating over B. We can then increment the i cursor by 1, and continue the search for a common value. This is a very procedural approach, and it was pretty difficult to translate it in MiniZinc. My second supervisor, Jip Dekker, then suggested another approach: concatenate A and B into a temporary array C. It would then suffice to use the global cardinality constraint on C to return the set of values present twice in C. This global constraint approach proved to increase solving time performance by a factor of 7. Since then, I have started to read more about these fascinating functions, trying to understand how they work and revisiting the very useful catalog of global constraints written by N. Beldiceanu [16] (my global constraints professor) et al.

3.3.2 Membership

The membership constraint ensure that a given element e belongs to a set variable.

In the following illustration, we consider the set variable x associated to the set domain $D_x = \{s_1, s_2, s_3, s_4\}$. e_1 and e_2 are two elements of the same type as the elements of x or D_x.



We have the following relationships:

- $e_1 \in x \Rightarrow x = s_1 \vee x = s_2$
- $e_2 \in x \Rightarrow False$

Many different implementations of the membership function were tried before a suitably efficient model was found. My background in procedural programming led me to consider the use of a dichotomous algorithm, but this approach is not suited to the `MiniZinc` language, since we are dealing with logical constraints. I therefore opted for a naive approach consisting of checking for each value of xi (you cannot "break" loops in MiniZinc) that is not a dummy element whether it is equal to e. But there exist a much more efficient approach: Jip Dekker suggested using the global constraint `arg_max` (named `max_index` in Beldiceanu [16]), which returns the index of the variables corresponding to the maximum value of the given collection. After much debugging, here is the short algorithm we eventually use:

Algorithm 1 SET_IN

```
Require: A set $s$, a value $e$
Ensure: $e$ belongs to $s$.
1: CONSTRAINT $xi = \texttt{set2int}(s)$
2: CONSTRAINT $\forall j \in \texttt{index}(xi)$, $arr\_b_j = (xi_j == e)$
3: CONSTRAINT $arr\_b_{\texttt{index}(xi)+1} = True$
4: GLOBAL CONSTRAINT $\texttt{arg\_max}(arr_b) \leq card(s)$
```

In this algorithm, we create an array of boolean arr_b which represent whether the j-th value of xi is equal to the value e. An extra boolean value set to $True$ is added at the end of arr_b as it will allow us to make use of the `arg_max` global constraint. We then ensure that the index of the maximum element in arr_b is less or equal to the cardinality of the input set s. We can do that as, in MiniZinc, $False$ is encoded as 0 and $True$ as 1.

3.3.3 Equality

The equality function ensures that two sets contain exactly the same values. As our sets are ordered, each index of each set must contain the same value, so only one iterator is needed. It is therefore natural to use an element constraint. Here is the final SET_EQ algorithm used in the *CardSet* library:

Algorithm 2 SET_EQ

```
Require: A set $s_1$, a set of value $s_2$
Ensure: $s_1$ equals $s_2$.
1: CONSTRAINT $card(s_1) = card(s_2)$
2: CONSTRAINT $xi = \texttt{set2int}(s_1)$
3: CONSTRAINT $yi = \texttt{set2int}(s_2)$
4: CONSTRAINT $xi_j = yi_j, \forall j \in 1..\ \texttt{min}(|xi|, |yi|)$
```

3.3.4 Lesser or equal inequality

The lesser or equal inequality guarantees that a set A is lexicographically less or equal to a set B. Once again, my lack of knowledge of global constraints led me to manually decompose the constraints using an iterator. But then I remembered that a lexicographic global constraint `lex_lesseq` already existed.

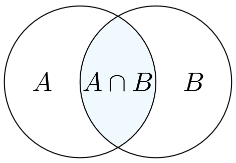
Algorithm 3 SET_LE

```
Require: A set $s_1$, a set of value $s_2$
Ensure: $s_1$ is lexicographically less or equal to $s_2$.
1: if $ub(s_1) = \emptyset$ then
2:     return $True$
3: else if $ub(s_2) =$ then
4:     CONSTRAINT $s_1 = \emptyset$
5: end if
6: CONSTRAINT $xi = \texttt{set2int}(s_1)$
7: CONSTRAINT $yi = \texttt{set2int}(s_2)$
8: GLOBAL CONSTRAINT $\texttt{lex\_lesseq}(xi, yi)$
```

3.3.5 Intersection

The intersection of two sets A and B is the set of common element shared by a set A and a set B. Formally:

$$A \cap B = \{x : x \in A \text{ and } x \in B\}$$



Venn diagram of the intersection of fixed sets A and B

The set intersection of two set variables compute one possible set intersection among all possible set intersections from all possible sets. Bellow is an illustration of set variables x and y belonging respectively to domains D_x and D_y intersecting for some subset of the possible sets.

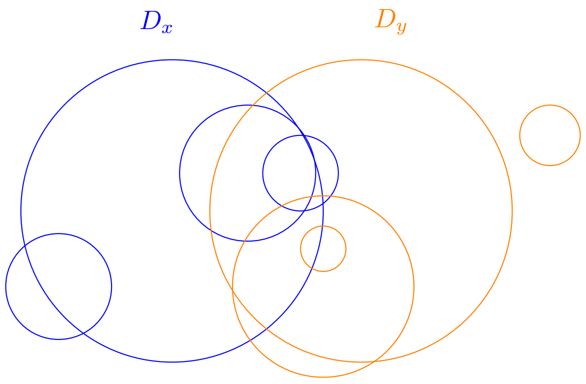


Figure 9: Venn diagram of set variables $x \subseteq D_x$ and $y \subseteq D_y$

The first idea proposed by Jip Dekker was to use a `table` global constraint. Table global constraint are extremely powerful constraint that are used to specify a relationship between multiple variables in a tabular form. The idea here was to create a table of indexes that informs which element are present in both arrays. We would then just use element constraints to enforce the values of the array to be equal at those informed indexes.

To help us illustrating the limitation of this approach, let consider the following example: Let xi and yi the array of integers representing the sets x and y, such that $xi = \{2, 4, 5, 7, 9, d_{11}\}$ and $yi = \{1, 3, 4, 7, 10, d_{21}, d_{22}\}$ (with d_{11} a dummy element for array xi, and d_{21}, d_{22} dummy elements for array yi). And zi the array representing z, the intersection of the sets x and y. Let $find$ be the table of indexes of common elements.

$find$ would thus look like this:

xi	yi
2	3
4	4

Figure 10: Table's indexes $find$ of xi and yi common values

The first common value (4), is at index 2 in array xi, and at index 3 in array yi. The second common value (7) is at index 4 in array xi, and at index 4 in array yi. We can now constraint the resulting array zi to be equal to the values of xi and yi at those indexes:
CONSTRAINT $zi_j = xi_{find_j} \wedge zi_j = yi_{find_j}, \forall j \in \texttt{index}(find)$.

The problem with this approach is that we do not constraint z_k values where $k > \texttt{index}(find)$. Additionally, we have no choice but to create the array $find$ of the size of the minimum upper bound cardinality of A and B, as we do not know in advance how many common values there will be. Which means we need to decide what default value to put in the remaining indexes that do not represent a common value. Let us put an absent value <> when there is no more common value. $min(ub(card(A)), ub(card(B)))$ being 6, our $find$ table will actually look like this:

xi	yi
2	3
4	4
<>	<>
<>	<>
<>	<>
<>	<>

Figure 11: Table's indexes $find$ of xi and yi common values

This would force us to constrain each element of the zi array (by iterating first on the first part of $find$ containing common values, then on the second part of $find$ containing non-common values), which we could do without this table constraint. The table constraint thus loses all its power.

I then worked on a way to go through each common value as efficiently as possible using the cursor approach. It was tedious and not very efficient. Fortunately, it was again Jip who came up with the idea of checking the common value in the concatenated array of the two sets. So it is very natural and powerful to use global cardinality to identify common values. If an element is present (at least) twice in the concatenated array, then it belongs to the resulting set, otherwise it does not. Below is the algorithandm used in the *CardSet* library:

Algorithm 4 SET_INTERSECT

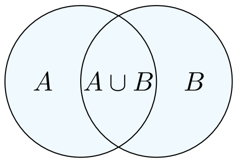
```
Require: A set $s_1$, a set $s_2$
Ensure: Return the set of common values shared by $s_1$ and $s_2$
 1: CONSTRAINT $xi$ = set2int($s_1$)
 2: CONSTRAINT $yi$ = set2int($s_2$)
 3: Let $s_3$ be the resulting set with $D_3$ its domain.
 4: CONSTRAINT $zi$ = set2int($s_3$)
 5: CONSTRAINT $xyi = [xi, yi]$
 6: Let $noccurs$ be the number of occurence of each value of $D_3$ in $xyi$.
 7: GLOBAL CONSTRAINT global_cardinality($xyi, D_3, noccurs$);
 8: CONSTRAINT $zi_j = D_{3_j}, \forall j : noccurs_j \geq 2$
 9: CONSTRAINT $zi_k \in dummies, \forall k : noccurs_k < 2$
10: return $s_3$
```

3.3.6 Union

The union of two sets A and B is the set of elements in A, in B, or in both A and B. Formally:

$$A \cup B = \{x : x \in A \text{ or } x \in B\}$$

.



Venn diagram of the union of fixed sets A and B

The set union of two set variables compute one possible set union among all possible set unions from all possible sets. We refer to Figure 5 to illustrate this: the variable set union returns the union of a decided (possible) pair of sets (x,y).
The algorithm is naturally very similar to SET_INTERSECT: we constrain the resulting set to contain the values that appear at least once in the concatenation of the x and y arrays.

Algorithm 5 SET_UNION

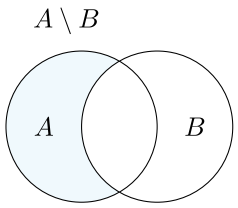
```
Require: A set $s_1$, a set $s_2$
Ensure: Return the set of values belonging to $s_1$ or $s_2$
 1: CONSTRAINT $xi = \texttt{set2int}(s_1)$
 2: CONSTRAINT $yi = \texttt{set2int}(s_2)$
 3: Let $s_3$ be the resulting set with $D_3$ its domain.
 4: CONSTRAINT $zi = \texttt{set2int}(s_3)$
 5: CONSTRAINT $xyi = [xi, yi]$
 6: Let $noccurs$ be the number of occurence of each value of $D_3$ in $xyi$.
 7: GLOBAL CONSTRAINT $\texttt{global\_cardinality}(xyi, D_3, noccurs)$;
 8: CONSTRAINT $zi_j = D_{3_j}, \forall j : noccurs_j \geq 1$
 9: CONSTRAINT $zi_k \in dummies, \forall k : noccurs_k < 1$
10: return $s_3$
```

3.3.7 Difference

The difference of two sets A and B is the set of elements in A which are not elements of B. Formally:

$$A \setminus B \Leftrightarrow x \in A \wedge x \notin B$$

.



Venn diagram of the difference of fixed sets A and B

To encode this operation we are once again going to invoke the global cardinality on the concatenation of xi and yi. Indeed, if the number of occurrences of a value from the upper bound domain of x is exactly one, it should appear in the resulting set.

Algorithm 6 SET_DIFF

```
Require: A set $s_1$, a set $s_2$
Ensure: Return the set of values belonging to $s_1$ but not to $s_2$
 1: CONSTRAINT $xi = \texttt{set2int}(s_1)$
 2: CONSTRAINT $yi = \texttt{set2int}(s_2)$
 3: Let $s_3$ be the resulting set with the same domain $D_1$ as $s_1$.
 4: CONSTRAINT $zi = \texttt{set2int}(s_3)$
 5: CONSTRAINT $xyi = [xi, yi]$
 6: Let $noccurs$ be the number of occurence of each value of $D_1$ in $xyi$.
 7: GLOBAL CONSTRAINT $\texttt{global\_cardinality}(xyi, D_1, noccurs)$;
 8: CONSTRAINT $zi_j = D_{1_j}, \forall j : noccurs_j = 1$
 9: CONSTRAINT $zi_k \in dummies, \forall k : noccurs_k \neq 1$
10: return $s_3$
```

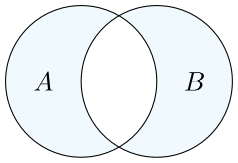
3.4 Global constraints redefinition

Constraining sets to be disjoint is very common in SET models. Below we propose efficient disjoint global constraint redefinitions for *CardSet*.

3.4.1 disjoint

The *union disjointedness* ($\sqcup$) of two sets A and B is the union of A and B such that no element overlap. Formally:

$$A \sqcup B = (A \cup B) \setminus (A \cap B)$$



Venn diagram of the disjointness of the fixed sets A and B

The disjoint constraint on two variable sets requires that no elements overlap between them. This global constraint can be expressed in terms of intersections: as an empty intersection between the sets, or as a zero cardinality of their intersection. In Minizinc, `disjoint` is in fact defined as $s_1 \cap s_2 = \{\}$. To take full advantage of our cardinal representation, we can redefine this global constraint by concatenating the arrays of integers representing s_1 and s_2 into a single array of integer, then use the global constraint `all_different` to ensure that no elements overlap. In the special case where all sets share the same domain, we can use `gcc` over this shared domain to guarantee a single copy of each element, which can further improve our redefinition. In the latter case, we will use another definition of `gcc` which allows us to constrain that the occurrence of each element can vary from zero to one. So here is the algorithm for our redefinition of `disjoint` :

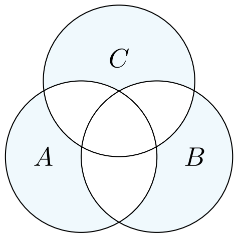
Algorithm 7 DISJOINT

```
Require: A set $s_1$, a set $s_2$
Ensure: There is no element overlapping between the sets in $s_1$ and $s_2$.
1: if $D(s_i)$ are all equals then
2:     CONSTRAINT $domain\_all = \texttt{array\_union}([ub(s_1), ub(s_2)])$
3:     GLOBAL CONSTRAINT
           $\texttt{gcc}(\texttt{array\_union}([set2int(s_1), set2int(s_2)])$,
               $domain\_all$,
               array of $domain\_all$ zeros,
               array of $domain\_all$ ones)
4: else
5:     GLOBAL CONSTRAINT
           $\texttt{all\_different}(\texttt{set2int}(s)_i \mid s \in [s_1, s_2], i \in \texttt{index\_set}(\texttt{set2int}(s),$
                                             $\texttt{set2int}(s)_i \neq dummy)$
6: end if
```

3.4.2 all_disjoint

The *union all disjointedness* ($\bigsqcup$) of a collection of sets S is the union disjointness of each sets s_i. Formally:

$$\bigsqcup S = s_i \sqcup s_2 \sqcup ... \sqcup s_n$$



Venn diagram of the all_disjointness of fixed the sets A, B and C

The all disjointedness of a collection of set variables ensures that no elements overlap between sets. Originally, `MiniZinc` `all_disjoint` is defined as $\bigcap_{s_i} = \{\}$. Here, using concatenation becomes even more efficient than in the redefinition of `disjoint`, as we now have not 2 but n sets. We have therefore extended the previous algorithm to n sets for `all_disjoint`. This redefinition seems particularly effective for the social golfer problem presented below.

3.4.3 `count_common_element`

The set disjointedness described above belongs to a more general type of constraint: the counting of common elements shared by two sets. This constraint is widely used in set models, and is defined as $card(s_1 \cap s_2)$. Theoretically, our representation is particularly effective in dealing with this type of constraint, since it explicitly represents cardinality. In the current version of *CardSet*, we have defined a constraint `count_common_element` to be called instead of declaring $card(s_1 \cap s_2)$. A future version should call this predicate automatically when the latter, more expressive declaration is used.

Algorithm 8 COUNT_COMMON_ELEMENT

```
Require: A set $s_1$, a set $s_2$
Ensure: Return the cardinality of the set of common element shared by $s_1$ and $s_2$
1: CONSTRAINT$values = \texttt{array\_union}([ub(s) \mid sin[s_1, s_2]])$
2: Let $noccurs$ be the number of occurence of each value in $values$
3: GLOBAL CONSTRAINT
       $noccurs = \texttt{gcc}([set2int(s)[i] | sin[x, y], iinindex\_set(set2int(s))]$ ,
           $values)$
4: return GLOBAL CONSTRAINT $\texttt{count}(noccurs, 2)$
```

4 Numerical experiments and validation

Finding problems that would benefit from our *CardSet* library and be relevant to users is a challenging problem in itself. Indeed, we need to find sufficiently general problems from the literature that resemble real modeling problems. Additionally, to prove the effectiveness of our library, we would ideally need as many models as possible. We are always on the lookout for new models that are relevant enough to be considered as benchmarks. So far, we have tested *CardSet* on 4 problems: the steiner triple problem, the social golfer problem, a work assignment problem, and a scheduling problem.

4.1 Experiments protocols

Experiments environment:
We have run our experiments on a remote server with the following characteristics:

- NECTAR's infrastructure
- 16 vCPUs
- 32 GB of RAM

In order to schedule our jobs, and collect statistics we used `mzn-bench` [18] python library. Jobs were automatically scheduled on *Slurm Workload Manager* [27] with an allocated memory of 4GB.

Modeling language
`MiniZinc` was used to model the three different problems.

Solvers
Each solver has its own strengths and weaknesses, depending on the nature of the problems to be solved. In order to better represent the efficiency of the *CardSet* library and to see how different solving techniques can impact resolution times, 4 solvers were used in the experiments:

– `Gecode`: One of the most prominent constraint programming systems providing state-of-the-art performance. `Gecode` features constraints on sets, allowing us to compare with its internal representation of sets.
– `Chuffed`: An open-source Constraint Programming solver with the distinctive feature of state-of-the-art Lazy Clause Generation.
– `Highs`: An open-source LP/MIP solver, implementing techniques for solving this kind of problem (Simplex, Interior Point, Branch-and-cut...) We were interested in Cardset's performance with this type of solver.
– `Choco`: Another free open-source constraint programming solver, with different solving approaches.

We had initially included `Choco` in our experiments, but a technical problem prevented us from collecting all the data. A correction in the very near future will enable us to update this report with the Choco results. One can still find the results collected so far in Annexe C

Budget
A budget of *timeout* = 180 seconds has been allocated to the solvers.

In the following results, a green color indicates that, for the associated solver, the associated set encoding would be a preferable choice regarding its relative performance. A result in orange is a result of interest.

4.2 Case study 1: The Steiner's Triple Problem

Our first case study is a very simple model describing a very complex combinatorial problem: the Steiner's Triple Problem [28]. This is a very well-known and well-studied hyper-graph problem, making it a perfect first "guinea pig" for CardSet.

4.2.1 Problem description and model

Steiner's ternary [29] problem consists in finding a set of $n*(n-1)/6$ triplets of distinct integer elements in $U = \{1, ..., n\}$ such that two triplets have at most one element in common. This very simple model seemed ideal for testing the effectiveness of our `count_common_element`:

Data

- An integer n.
- $nb = (n*(n-1))/6$

Decision variables

- *sets*: an array of nb set variables ranging from 1 to n.

Constraints

- CONSTRAINT $card(sets_i \cap sets_j) \leq 1, \forall i \in 1..nb, \forall j \in i+1..nb.$ (1)
- We also add the following symmetrie breaking
  CONSTRAINT $sets_i \geq sets_{i+1}, \forall i \in 1..(nb-1)$ (2)

For the *CardSet* encodings, constraints (1) are replaced by the `count_common_element` global constraint.

4.2.2 Experimental results

Steiner's triple problem quickly reaches our budget for small values of n. We therefore decided to add the UNSAT instances (in gray), which also allows us to focus the analysis of *CardSet* on its impact in terms of information to help solvers identifying UNSAT branches.

Hypothesis
We expect that `count_common_element` global redefinition of the constraints to contribute to a significant reduction in solution time. As the cardinalities of the set variables are fixed, we also expected CardSet to outperform BoolSet for the largest instances of the problem. Table 10 shows our experimental results.

Observations
It seems that CardSet performs better on `Chuffed` solver for this problem. We can see that *CardSet* performs as well as *BoolSet* for most solvers and instances. On the `Gecode` solver, *CardSet* is significantly faster than *BoolSet* on instances 08 and 09. `Chuffed` is the solver where *CardSet* performs best for this problem. *BoolSet* provides slightly

Table 5: Steiner's Triple Problem solving times comparison (in seconds).

Instances	Gecode				Chuffed			Highs		
	Set	BoolSet	CardSet		BoolSet	CardSet		BoolSet	CardSet	
			opt	nem		opt	nem		opt	nem
03	0.091	0.119	0.252	0.276	0.09	0.266	0.271	0.091	0.266	0.266
04	0.119	0.097	0.395	0.391	0.099	0.285	0.279	0.112	0.278	0.286
05	0.117	0.257	0.619	0.634	0.603	0.62	0.634	0.143	0.603	0.626
06	0.097	0.134	0.284	0.286	0.125	0.291	0.287	0.838	1.873	1.384
07	0.115	0.144	0.287	0.297	0.157	0.3	0.304	1.815	3.524	4.885
08	3.156	89.951	3.748	3.145	0.433	0.484	0.461	42.653	timeout	timeout
09	0.111	timeout	1.191	2.741	0.378	0.529	0.414	58.834	52.861	136.204
10	timeout	timeout	timeout	timeout	67.626	59.897	64.768	timeout	timeout	timeout
11	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
12	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
13	timeout	timeout	timeout	timeout	timeout	19.623	3.913	timeout	timeout	timeout
14	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
15	0.142	timeout	timeout	timeout	timeout	7.557	timeout	timeout	timeout	timeout
16	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
17	0.139	timeout	timeout	timeout	timeout	7.302	timeout	timeout	timeout	timeout
18	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout
19	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout	timeout

better result than *CardSet* on `Highs`, except for instance 08 where *BoolSet* found a solution in 42.653 seconds but *CardSet* representations are both timeout. Interestingly, *CardSet* is the only encoding to prove unsatisfiability in the allotted time for instance 13.

Analysis and conclusion:
We tried using the original constraint (1) with `card` function redefinition together with the `intersection` redefinition for $sets_i$ and $sets_j$. We noted a significant drop in performance (only instance 03 could be solved in the allotted time). Our `count_common_element` shows its effectiveness for this problem. For small instances, the *BoolSet* encoding is faster than *CardSet*, but for some larger instances, *CardSet* is faster (i.e. 10, 13, 15, 17). In conclusion, *BoolSet* is preferable for small instances of Steiner's Triple, while *CardSet* would be preferable for solving larger instances of this problem.

4.3 Case study 2: The Social Golfer Problem

The Social Golfer Problem (SGP) [30] is a $\mathcal{NP}$-hard combinatorial mathematics problem with many practical applications, notably for resource planning and allocation. It will make an excellent benchmark for our *CardSet* library, as it is naturally modeled using sets. In addition, we will be able to test our presumably very powerful `alldisjoint` global constraint redefinition. Here too, set variables have a fixed cardinality, which promises interesting results.

4.3.1 Problem description and model

Consider g golfers play every week for s weeks, divided into k groups of p golfers ($g = k.p$). The aim is to find a distribution of players in the groups, such that each player plays every week, over a maximum number of weeks, while respecting two constraints: a weekly assignment constraint which enforces each player to play in exactly one group each week and a sociability constraint which state that if $j1$ and $j2$ play in the same group in week s_1, then they cannot be in the same group in week s_2.

This problem can be modeled as an optimization problem, where the objective is to find an assignment of golfers that maximizes the number of weeks played. In this experiment, we will study the associated decision problem.

We experimented on the following model:

Data

- The number of groups n_group
- The size of each group n_per_group
- The number of rounds n_rounds
- The number of golfer $n\_golfers = n\_groups * n\_per\_group$
- The set of groups $groups = 1..n\_groups$
- The set of inside group positions $group = 1..n\_per\_group$
- The set of rounds $rounds = 1..n\_rounds$
- The set of golfers $golfers = 1..n\_golfers$

Decision variables

- The array that assign the groups of golfers to each round: $round\_group\_golfers$

Constraints

- *Group size constraint*: Each group has to have the right size.
  CONSTRAINT
  $card(round\_group\_golfers_{r,g}) = n\_per\_group, \forall g \in groups, \forall r \in rounds \quad (1)$
- *Disjoint constraint*: Each group in each round has to be disjoint.
  CONSTRAINT (GLOBAL)
  $\texttt{all\_disjoint}(round\_group\_golfers_{r,g}), \forall g \in groups, \forall r \in rounds \quad (2)$
- *Social constraint*: a pair of golfers can play in the same group in at most one round.
  CONSTRAINT
  $\sum_{r \in rounds} \sum_{g \in groups} (\texttt{bool2int}(\{a, b\} \subseteq round\_group\_golfers_{r,g}) \leq 1,$

$$\forall a < b \in golfers \tag{3}$$

4.3.2 Experimental results

In what follows, we show our experimental results for solving the SGP model given previously, on 28 instances: 20 instances come from the Cardinal paper [24], and 8 instances from the MiniZinc Benchmark repository.

Hypothesis
This problem benefits our *CardSet* library, thanks to our *CardSet* `all_disjoint` which uses a redefinition of `global cardinality`. We therefore expect our CardSet to perform well in most cases. Table 6 shows our experimental results. Graphs are available in Annexe B

Table 6: Social Golfers Problem solving times comparison (in seconds).

Instances	Gecode				Chuffed			Highs		
	Set	BoolSet	CardSet		BoolSet	CardSet		BoolSet	CardSet	
			opt	nem		opt	nem		opt	nem
2_5_4	0.538	180	0.971	0.855	26.187	0.723	0.741	3.904	5.282	4.731
2_6_4	110.548	180	0.813	0.82	180	0.931	0.877	7.136	8.836	6.718
2_7_4	180	180	1.143	0.947	180	1.28	1.086	7.629	14.091	11.193
2_8_5	180	180	2.292	2.169	180	2.877	2.95	31.566	29.368	27.801
3_2_2	0.196	0.262	0.279	0.292	0.205	0.28	0.288	0.22	0.345	0.303
3_5_4	0.597	180	1.249	1.071	1.367	1.434	1.299	18.779	11.249	8.847
3_6_4	78.311	180	1.132	0.96	11.852	1.296	1.201	12.303	14.664	11.888
3_6_6	0.974	180	2.27	2.252	180	2.652	2.522	180	180	180
3_7_4	180	180	1.627	1.428	180	2.845	2.134	23.542	24.281	18.407
4_4_4	11.362	180	0.562	0.558	180	0.667	0.629	23.343	41.882	36.813
4_5_4	28.813	180	0.927	1.069	180	1.582	4.102	56.214	69.483	135.641
4_6_5	180	180	180	180	180	33.687	66.195	180	180	180
4_7_4	180	180	2.09	1.841	129.746	2.744	2.693	48.488	85.643	119.109
4_9_4	180	180	4.942	4.459	180	7.357	8.371	80.628	75.274	75.88
5_4_3	180	180	180	180	180	180	180	180	180	180
5_4_4	0.971	180	0.824	0.594	180	0.874	0.656	59.038	180	180
5_5_4	180	180	180	180	180	4.713	9.595	180	180	180
5_7_4	180	180	180	180	84.537	180	180	180	180	180
5_8_3	180	180	2.05	1.996	180	3.472	2.328	21.215	25.582	28.837
5_8_4	180	180	180	180	180	4.343	4.055	180	52.679	180
6_4_3	180	180	180	180	180	180	180	180	180	180
6_5_3	180	180	180	180	45.007	89.591	32.312	180	180	180
6_5_5	180	180	3.905	4.481	180	180	180	180	180	180
7_5_3	8.872	180	180	180	180	180	180	180	180	180
7_5_5	180	180	180	180	180	180	180	180	180	180
7_7_7	180	180	180	180	180	180	180	180	180	180
8_5_2	15.689	180	0.668	0.833	2.765	0.695	0.922	7.372	5.542	5.182
9_8_8	180	180	180	180	180	180	180	180	180	180

Observations
A first observation is that *CardSet* library seems to dominate the *BoolSet* library on both `Gecode` and `Chuffed` solvers. The native set encoding of `Gecode` outperforms the *BoolSet* library, but is itself significantly outperformed by the two *CardSet* libraries. The biggest performance gain for *CardSet* concerns `GeCode`, where most instances are timeout using the *BoolSet* library. On `Chuffed`, *CardSet* significantly outperform *BoolSet* on most instances. This is also where *CardSet* achieves the best overall results compared to other solvers. The smallest gain is for the MIP solver, where *BoolSet* and *CardSet_opt* achieve similar results for most instances. Interestingly, the performance comparison of the *CardSet* libraries is different for each solver: *CardSet_opt* slightly dominates

CardSet_nem with the solver `Gecode`, *CardSet_opt* and *CardSet_nem* perform similarly overall with the solver `Chuffed`, and *CardSet_nem* slightly outperforms *CardSet_opt* with the `Highs` solver.

Analysis and conclusion:
This Social Golfer Problem benchmark shows the contribution of our *CardSet* library. Solving times on both `GeCode` and `Chuffed` are significantly improved when using our library rather than *BoolSet*.

4.4 Case study 3: The Worker Assignment Problem

We have created an interesting version of the Worker Assignment Problem [31] for our *CardSet* library, with *many-to-many* relationships between variables (which means we cannot switch the representation of decision variables from a *many-to-1* to a *1-to-many*). In addition, we model variable set cardinality, so that dummy elements will be involved.

4.4.1 Problem description and model

Let us consider a corporate IT project. The project is divided into t sub-parts (tasks T). The project has w resources (workers W). The management team wants to assign between n_{min} to n_{max} different workers to each task, and each worker to m_{min} to m_{max} different tasks. In addition, each worker has individual qualifications such that he or she can only perform a subset of tasks in T. Each task can only be assigned to a subset of possible teams, because some of the workers hate each other. The problem is to find an assignment of workers to tasks such that all constraints are satisfied.

We experimented on the following model:

Data

- A set T of t tasks
- A set W of w workers
- A number of workers assigned to each task $n \in n_{min}..n_{max}$
- A number of tasks assigned to each worker $m \in m_{min}..m_{max}$
- An array of hatred H giving in H_i the set of workers hated by worker i.
- An array of qualification Q giving in Q_i the set of tasks workers i is qualified to do.

Decision variables

- An array of *teams* representing the set of workers assigned to each task: $Assign\_Task$

Constraints

- *Workforce constraint*: Between n_{min} and n_{max} workers per task.
  CONSTRAINT $Card(Assign\_Task_t) \in n_{min}..n_{max}, \forall t \in T, \forall w \in W$ (1)
- *Productivity constraint*: Between m_{min} and m_{max} tasks per worker.
  CONSTRAINT (GLOBAL)
  `nval`(w,`array_set_concatene`($Assign\_Task_t$)) $\in m_{min}..m_{max}, \forall t \in T$ (2)
- *Anti-hate constraint*: Assigned workers to a task cannot hate each other.
  CONSTRAINT $Assign\_Task_t \in Stable\_Team, \forall t \in T$ (3)
- *Qualification constraint*: Workers assigned to a task have to be qualified for it.
  CONSTRAINT $w \in Q_t, \forall w \in Assign\_Task_t, \forall t \in T$ (4)

For the *CardSet* encodings, constraints (1) are replaced by the `count_common_element` global constraint.

4.4.2 Experimental results

The instances were automatically generated by a python script. Their names describe the parameters of the instance:

$$t\_w\_n_{min}\_n_{max}\_m_{min}\_m_{max}\_hr\_qr$$

with *hr* the *hatred rate*, which is the probability that two workers will hate each other, and *qr* the qualification rate, which is the probability that each worker will be qualified for each task. Probabilities range from 0 to 100.

The following instances proposed small to medium set variable cardinality domains: the cardinality varies from 3 to 5 elements, in other instances it varies from 2 to 10. Some instances have only one element in their set variable cardinality domain, which means the cardinality is fixed for that instance.

We have experienced a bug while running the benchmark for the CardSet_nem library on this problem. A fix will allow us to provide the data soon.

Hypothesis
In this study case, set variables have variable cardinalities, which means that dummy elements are going to be involved in the encodings. We then expect to have lower performance for wider cardinality ranges instances. Table 7 shows our experimental results.

Table 7: Worker Assignment Problem solving times comparison (in seconds).

Instances	Gecode				Chuffed			Highs		
	Set	BoolSet	CardSet		BoolSet	CardSet		BoolSet	CardSet	
			opt	nem		opt	nem		opt	nem
m_100_100_4_100_4_100_5_50	4.133	38.315	n/a	n/a	44.349	n/a	n/a	84.379	n/a	n/a
m_10_100_2_10_2_10_5_50	0.594	3.723	timeout	n/a	3.938	timeout	n/a	7.509	n/a	n/a
m_10_10_3_10_3_10_5_50	0.391	0.24	2.352	n/a	0.363	1.579	n/a	0.29	11.954	n/a
m_10_10_3_4_3_4_5_60	0.229	0.303	45.222	n/a	0.241	0.533	n/a	0.322	1.977	n/a
m_10_12_3_10_3_10_15_75	1.465	0.251	timeout	n/a	0.325	2.244	n/a	0.354	24.382	n/a
m_10_3_1_1_3_10_0_100	0.392	0.248	0.232	n/a	0.292	0.252	0.61	0.195	0.332	0.625
m_15_12_3_5_3_5_30_75	4.606	5.231	timeout	n/a	0.328	1.445	n/a	0.676	5.283	n/a
m_15_18_3_5_3_5_15_80	2.272	0.469	timeout	n/a	0.78	2.806	n/a	0.821	47.31	n/a
m_20_18_3_6_3_6_15_80	0.456	0.464	timeout	n/a	0.565	3.63	n/a	0.987	timeout	n/a
m_20_18_3_6_3_6_30_80	timeout	0.555	timeout	n/a	0.554	7.423	n/a	2.204	37.962	n/a
m_30_20_3_5_3_5_10_50	timeout	timeout	3.105	n/a	timeout	4.782	n/a	1.31	91.76	n/a
m_3_3_2_2_2_2_0_100	0.368	0.203	0.335	n/a	0.19	0.379	0.298	0.199	0.382	0.407
m_5_10_2_3_2_3_5_100	2.0	0.707	67.274	n/a	0.31	1.153	0.514	0.311	1.004	0.56
m_7_10_3_5_3_7_15_80	0.695	0.243	timeout	n/a	0.251	0.764	n/a	0.316	3.715	n/a

Observations
Our first observation concerns the poor performance of our *CardSet_opt* library compare to *BoolSet*. From the actual instance, the width of the cardinality domain does not seem to affect performance, but we do not have enough data to fully assess this. Interestingly, instance m_30_20_3_5_3_5_10_50 is solved significantly quicker by *CardSet_opt* than by *BoolSet* with `Gecode` and `Chuffed`. Once again, *CardSet* seems to perform better on `Chuffed`.

Analysis and conclusion:
For this many-to-many problem, we might have expected better performance from *CardSet*. We can't conclude on the impact of dummy elements from this single experiment. We would need to test many more instances of this problem to draw any conclusions. Nevertheless, the instance generator developed for this problem enables many more experiments to be carried out: it allows one to adapt the difficulty level of the instance for *CardSet* by providing the appropriate parameters.

4.5 Case study 4: The Talent Scheduling Problem

This case study presents a pure scheduling problem. It involves fewer set counting set constraints and more "scheduling" set constraints, i.e set unions and intersections to represent temporalities. Additionally, unlike the previous CSP case study, this is an optimization problem.

4.5.1 Problem description and modelisation

The talent scheduling problem [32] concerns a producer who wants to make films. Each film contains several scenes. Each scene must be shot by one or more actors. The director can only shoot one scene per day. The actors' salaries are calculated by the day. In this problem, we can only hire each actor consecutively. For example, we cannot hire an actor on the first and third days, but not on the second. During the hiring period, producers still have to pay actors, even if they do not take part in the shoot. The aim of talent scheduling is to minimize the actors' total salary by adjusting the sequence of scenes.

Data

- $numActors$ (number of actors)
- $numScenes$ (number of scenes)
- $Actors$: set of $1..numActors$ actors
- $Scenes$: set of $1..numScenes$ scenes
- ia: array of $numActors \times Scenes$ booleans ($\{0,1\}$ definition of actors in scenes)
- a: array of $numScenes$ set of $Actors$ $= [j|jinActorswhereia_{j,i} = 1|iinScenes]$ (actors for each scene)
- d: array of $numScenes$ integers (duration of each scene)
- c: array of $numActors$ integers (cost of each actor)

Decision variables

- s: array of $numScenes$ $Scenes$ variable Scenes: s;

Auxiliary variables

- bef: array of $numScenes$ set variables of $Actors$ (actors appearing before scene t)
- aft: array of $numScenes$ set of variables $Actors$ (actors appearing after scene t)
- dur: array of $numScenes$ set of Actors (actors on set at scene t)
- $cost$: integer variable

Constraints

- Each scene scheduled once
  CONSTRAINT `alldifferent`(s) (1)
- CONSTRAINT $bef_1 = \{\} \wedge aft_{numScenes} = \{\}$ (2)
  $\wedge ((bef_{t+1} = a_{s_t} \cup bef_t) \wedge (aft_t = a_{s_{t+1}} \cup aft_{t+1}), \forall t \in 1..(numScenes - 1))$
- CONSTRAINT $dur_1 = a_{s_1} \wedge (dur_t = bef_{t+1} \cap aft_t, \forall t \in 2..(numScenes - 1))$
  $\wedge dur_{numScenes} = a_{s_{numScenes}}$ (3)
- CONSTRAINT $cost = \sum_{i \in Scenes} \sum_{j \in Actors} c_j * d_{s_i} * \texttt{bool2int}(j \in dur_i)$ (4)

– CONSTRAINT $s_1 < s_{numScenes}$ (5)

Constraint (1) ensures that all scenes are different. Constraint (2) represents that no actors are hired before scene 1, and after scene *numScenes*, and constraint the array of set of actors *bef* and *aft*. Constraint (3) states that if an actor has a scene after scene t and had a scene before t, he must be there at scene t. Constraint (4) compute the cost objective. Constraint (5) is a symmetry breaking, enforcing that the elements in the solution array are ordered.

4.5.2 Experimental results

We have experienced a bug while running the benchmark for the CardSet_nem library on this problem. A fix will allow us to provide the data soon.

Hypothesis
We expect this problem to be very challenging for our *CardSet* libraries, as the models involve constraints for which our encoding is not efficient at this stage, compare to *BoolSet* (namely `intersection` and `union`).

The instances used for this experiment can be found in the `MiniZinc` benchmarks suite [33]. Table 8 shows our experimental results.

Table 8: Talent Scheduling Problem solving times comparison (in seconds).

Instances	Gecode				Chuffed			Highs		
	Set	BoolSet	CardSet		BoolSet	CardSet		BoolSet	CardSet	
			opt	nem		opt	nem		opt	nem
concert	1.926	1.551	49.586	n/a	1.54	67.929	n/a	14.482	timeout	n/a
small	0.238	0.316	0.621	n/a	0.272	0.602	n/a	2.682	56.592	n/a
tiny	0.388	0.327	0.547	n/a	0.299	0.557	0.391	0.301	0.593	0.468
MobStory	timeout	timeout	timeout	n/a	timeout	timeout	n/a	timeout	timeout	n/a
film103	timeout	timeout	timeout	n/a	timeout	timeout	n/a	timeout	timeout	n/a
film105	timeout	timeout	timeout	n/a	timeout	timeout	n/a	timeout	timeout	n/a
film114	timeout	timeout	timeout	n/a	timeout	timeout	n/a	timeout	timeout	n/a
film116	timeout	timeout	timeout	n/a	timeout	timeout	n/a	timeout	timeout	n/a
film117	timeout	timeout	timeout	n/a	timeout	timeout	n/a	timeout	timeout	n/a
film118	timeout	timeout	timeout	n/a	timeout	timeout	n/a	143.777	timeout	n/a
film119	timeout	timeout	timeout	n/a	timeout	timeout	n/a	timeout	timeout	n/a

Observations
It turns out that this problem is challenging for every libraries on every solvers. Most of instance are timeout.

Analysis and conclusion:
This problem is particularly hard to solve, even when we increase the timeout to 1800 seconds. In this stage, we cannot really conclude on how worse the actual versions of *CardSet* are compare to *BoolSet* or *GeCode set* library. In a next work, we propose to soften some constraints of this problem in order to test more how *CardSet* perform on Time Scheduling Problems.

4.6 Conclusion of experiments

On fixed set cardinality problems, *CardSet* shows improved performance compared to *BoolSet*. Overall, *CardSet* seems to perform better on `Chuffed`. It also seems that *CardSet_opt* outperform *CardSet_nem*. With the current data collected, we cannot give a definitive conclusion on the efficiency of *CardSet* for problems involving variables with non-fixed cardinality. To draw more clear conclusion about *CardSet* efficiency on bounded set cardinality, more benchmark should be conducted in the future.

5 Conclusion

We propose a new set encoding for set variables to complement the existing Boolean (*BoolSet*) set encoding: an array of integers named *CardSet*. The size of the represented array corresponds to the upper cardinality of the set variable, and its values range over the set domain values. This representation allows different encodings for set constraints, making it possible to use cardinality information, which could not be used as much in the *BoolSet* representation. This encoding shows promising efficiency to help solvers dealing with bounded cardinality set variables. Further work is both ongoing and being considered to advance the *CardSet* library itself, as well as to investigate broader add-ons related to set encoding.

5.1 CardSet: works in progress

Our ongoing work is to further investigate the results of the benchmark tests, to refine our encodings. We also plan to automate the selection of the appropriate noset library according to the input model.

5.1.1 Benchmark on more models and solvers

We aim to expand our experimentation efforts by testing our *CardSet* library on a broader range of models and solvers. While we have successfully demonstrated the efficacy of our library in specific scenarios, it is imperative to assess its performance and robustness. `Choco` was originally intended to be part of our set of solvers to be tested, but technical problems prevented us from adding it before the project deadline.

5.1.2 Library optimization

The exploration of various models benchmark might uncover areas where our *CardSet* library can be enhanced further. Our Talent Scheduling case study reveals some potential weakness in some set function redefinition, especially the intersection and union redefinitions. Maybe a fuzzy set encoding approach could lead to some interesting results?

5.1.3 Automatic library selection

We plan to automate the process of selecting the most appropriate encoding library based on the characteristics of the input model. This automated selection would take into account factors such as decision variables, the nature of constraints and solver preferences to dynamically choose between the libraries `BoolSet`, `CardSet_nem` and `CardSet_opt`.

5.2 Sparse sets and BoolSet

During development of the new *CardSet* library, we realized that the *BoolSet* representation of sets with non-contiguous domains, i.e. sets with holes in their domain, could be improved. The current *BoolSet* representation uses contiguous integer indices to represent the elements of a set. When the set has sparse values, this can lead to inefficient use of memory and resources. We consider using heuristics to dynamically create the indexes. Note that the *CardSet* representation has a significant advantage in this context because it relies on the cardinality of the sets rather than the domain of the set.

5.3 Integrating Lazy Clause Generation on set variables for `Chuffed` solver

Currently, Chuffed's LCG do not handle set variables. A challenging yet very interesting future project would be to incorporate no good decision learning for set variables.

5.4 Concluding

To conclude, this project has been an extremely rewarding and fascinating experience for me. There is a lot of future work and improvements waiting to be explored, but *CardSet* seems to show very interesting potential, and promises interesting further research.

References

[1] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[2] Thom Frühwirth, Laurent Michel, and Christian Schulte. "Constraints in procedural and concurrent languages". In: *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 2006, pp. 453–494.

[3] Mark Wallace, Stefano Novello, and Joachim Schimpf. "ECLiPSe: A platform for constraint logic programming". In: *ICL Systems Journal* 12.1 (1997), pp. 159–200.

[4] Nicholas Nethercote et al. "MiniZinc: Towards a standard CP modelling language". In: *13th International Conference on Principles and Practice of Constraint Programming, volume 4741 of LNCS (editor C. Bessiere)*. Springer. 2007, pp. 529–543.

[5] Pascal Van Hentenryck, David McAllester, and Deepak Kapur. "Solving polynomial systems using a branch and prune approach". In: *SIAM Journal on Numerical Analysis* 34.2 (1997), pp. 797–827.

[6] Marc B Vilain and Henry A Kautz. "Constraint propagation algorithms for temporal reasoning." In: *Aaai*. Vol. 86. 1986, pp. 377–382.

[7] Willem-Jan van Hoeve and Irit Katriel. "Global constraints". In: *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 2006, pp. 169–208.

[8] Pascal Van Hentenryck and Jean-Philippe Carillon. "Generality versus specificity: An experience with AI and OR techniques". In: *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*. 1988, pp. 660–664.

[9] Jean-Charles Régin. "A filtering algorithm for constraints of difference in CSPs". In: *AAAI*. Vol. 94. 1994, pp. 362–367.

[10] Willem-Jan Van Hoeve. "The alldifferent constraint: A survey". In: *arXiv preprint cs/0105015* (2001).

[11] Christine Solnon. "Alldifferent-based filtering for subgraph isomorphism". In: *Artificial Intelligence* 174.12-13 (2010), pp. 850–864.

[12] Jena-Lonis Lauriere. "A language and a program for stating and solving combinatorial problems". In: *Artificial intelligence* 10.1 (1978), pp. 29–127.

[13] Alejandro López-Ortiz et al. "A fast and simple algorithm for bounds consistency of the alldifferent constraint". In: *IJCAI*. Vol. 3. 2003, pp. 245–250.

[14] Armen S Asratian, Tristan MJ Denley, and Roland Häggkvist. *Bipartite graphs and their applications*. Vol. 131. Cambridge university press, 1998.

[15] Christian Bessiere et al. "Propagating conjunctions of alldifferent constraints". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 24. 1. 2010, pp. 27–32.

[16] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. *Global constraint catalog, (revision a)*. 2012.

[17] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. "Propagation= lazy clause generation". In: *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*. Springer. 2007, pp. 544–558.

[18] Geoffrey Chu et al. *Chuffed, a lazy clause generation solver*. https://github.com/chuffed/chuffed. 2016.

[19] Geoffrey G Chu. *Improving combinatorial optimization*. University of Melbourne, Department of Computer Science and Software Engineering, 2011.

[20] Peter J. Stuckey et al. *MiniZinc is a free and open-source constraint modeling language*. https://github.com/MiniZinc. 2013.

[21] Carmen Gervet. "Conjunto: constraint logic programming with finite set domains". In: *ILPS*. 1994.

[22] Carmen Gervet. "Interval propagation to reason about sets: Definition and implementation of a practical language". In: *Constraints* 1 (1997), pp. 191–244.

[23] Graeme Gange, Peter J Stuckey, and Vitaly Lagoon. "Fast set bounds propagation using a BDD-SAT hybrid". In: *Journal of Artificial Intelligence Research* 38 (2010), pp. 307–338.

[24] Francisco Azevedo. "Cardinal: A finite sets constraint solver". In: *Constraints* 12.1 (2007), pp. 93–129.

[25] Frederic Lardeux et al. "Set constraint model and automated encoding into SAT: application to the social golfer problem". In: *Annals of Operations Research* 235.1 (2015), pp. 423–452.

[26] Garrett Birkhoff. *Lattice theory*. Vol. 25. American Mathematical Soc., 1940.

[27] Danny Auble et al. *Slurm: A Highly Scalable Workload Manager*. https://github.com/SchedMD/slurm. 2023.

[28] AD Forbes, Mike J Grannell, and Terry S Griggs. "Steiner triple systems and existentially closed graphs". In: *the electronic journal of combinatorics* 12.1 (2005), R42.

[29] Charles J Colbourn. *CRC handbook of combinatorial designs*. CRC press, 2010.

[30] Markus Triska. *Solution methods for the social golfer problem*. na, 2008.

[31] Reza Ramezanian and Abdullah Ezzatpanah. "Modeling and solving multi-objective mixed-model assembly line balancing and worker assignment problem". In: *Computers & Industrial Engineering* 87 (2015), pp. 74–80.

[32] TCE Cheng, JE Diamond, and Bertrand MT Lin. "Optimal scheduling in film production to minimize talent hold cost". In: *Journal of Optimization Theory and Applications* 79.3 (1993), pp. 479–492.

[33] Guido Tack et al. *A suite of MiniZinc benchmarks*. https://github.com/MiniZinc/minizinc-benchmarks. 2013.

[34] Jay Liebowitz. *The handbook of applied expert systems.* cRc Press, 2019.

[35] Kenneth H Rosen. *Discrete mathematics and its applications.* The McGraw Hill Companies, 2007.

[36] Jan Komorowski et al. “Rough sets: A tutorial”. In: *Rough fuzzy hybridization: A new trend in decision-making* (1999), pp. 3–98.

[37] J Correas, S Estevez Martin, and Fernando Saenz-Perez. “Enhancing set constraint solvers with bound consistency”. In: *Expert Systems with Applications* 92 (2018), pp. 485–494.

[38] Neng-Fa Zhou, Mark Wallace, and Peter J Stuckey. *The dom event and its use in implementing constraint propagators.* Tech. rep. Citeseer, 2006.

[39] YY Yao. “A comparative study of fuzzy sets and rough sets”. In: *Information sciences* 109.1-4 (1998), pp. 227–242.

[40] Christopher Jefferson et al., eds. *CSPLib: A problem library for constraints.* http://www.csplib.org. 1999.

A Internship overview

The main purpose of this appendix is to share my particular internship chronology and this google drive folder, where I've stored all the documents related to this report.

A.1 Monash University

Monash University is a public research university located in Melbourne, Australia. Founded in 1958, it is named after Sir John Monash. Monash University is consistently ranked as one of the world's leading universities for both research and teaching excellence. It is divided into two campuses: one in Clayton (where I did my internship) and the other in central Melbourne. The Clayton campus offers a very pleasant environment for students, with vast green spaces and gardens, with a wide variety of restaurants, grocery stores, hairdressers, pharmacies, cinemas and more.

A.1.1 OPTIMA

OPTIMA stands for Optimisation Technologies, Integrated Methodologies, and Applications. It is a world-class research group directed by Professor Kate Smith-Miles, composed of 19 Chief Investigators, working on 11 active research projects. OPTIMA is in collaboration with Monash University, the University of Melbourne and nine major industries including Boeing Australia, Melbourne Water, Future Fibre Technologies or ENGIE. This is the team I had the chance to joined for my internship.

A.2 My internship

My home university being in France and my internship being in Australia, my internship was a bit special. In the following lines, I'll describe a little more about my internship.

A.2.1 Chronology

The chronology of my internship can be divided into three parts, each corresponding to a different geolocation.

February to April: France (Remote) For the first three months before starting my internship, I searched for articles and books on my subject: ”Efficient encoding and propagators for set constraints”. I asked Eric Monfroy and Peter Stuckey for documentation, which gave me a ”starting list”. I tried to decipher as much as I could, always discovering new concepts as I delved deeper into the articles. I collected my notes on a shared google drive folder (subfolder MEMO).

May to July: Australia (Monash) I arrived in Melbourne the weekend before Monday May 1st. I was finally able to meet Guido Tack and Jip Dekker, and quickly obtained my access card, office and working environment. My first week was very productive, and it was only that week that I really understood my work objective. Every week, I had a meeting with Guido Tack and Jip Dekker, who helped me a lot by making relevant suggestions.

July to August: South east asia (Remote) At the end of July, my three-month electronic visa expired. So I booked a flight to Vietnam, where I have family. Only later did I learn that I didn't need to leave Melbourne, that I could apply for a visa extension and get a bridging visa, but anyway, I've booked the flight already. In Vietnam, my objective was to run the benchmarks, format the data collected and add it to the report. And writing the report, which was of course central. I had bugs in my benchmarks, but thanks to Jip (almost) everything worked in the end!

I made sure to regularly update a daily tracking file in which I wrote down my tasks for the day. This shared "Journal de bord" can be consulted here.

A.2.2 Difficulties

I encountered two main difficulties during this internship: programming a library for minizinc and connecting to the Internet. The former is explained in this report, but I haven't said anything about the quality of the internet connection in Southeast Asia. I quickly realized that turning on the cameras would not be part of our remote meetings. The biggest problem was that I couldn't share my screen either, and even voice eventually became too data-heavy to be streamed. So part of the meeting was conducted using keyboards and instant messaging. Fortunately, this didn't stop me making progress on my benchmark and report, thanks to Jip Dekker's detailed answers to my questions!

B Graphs

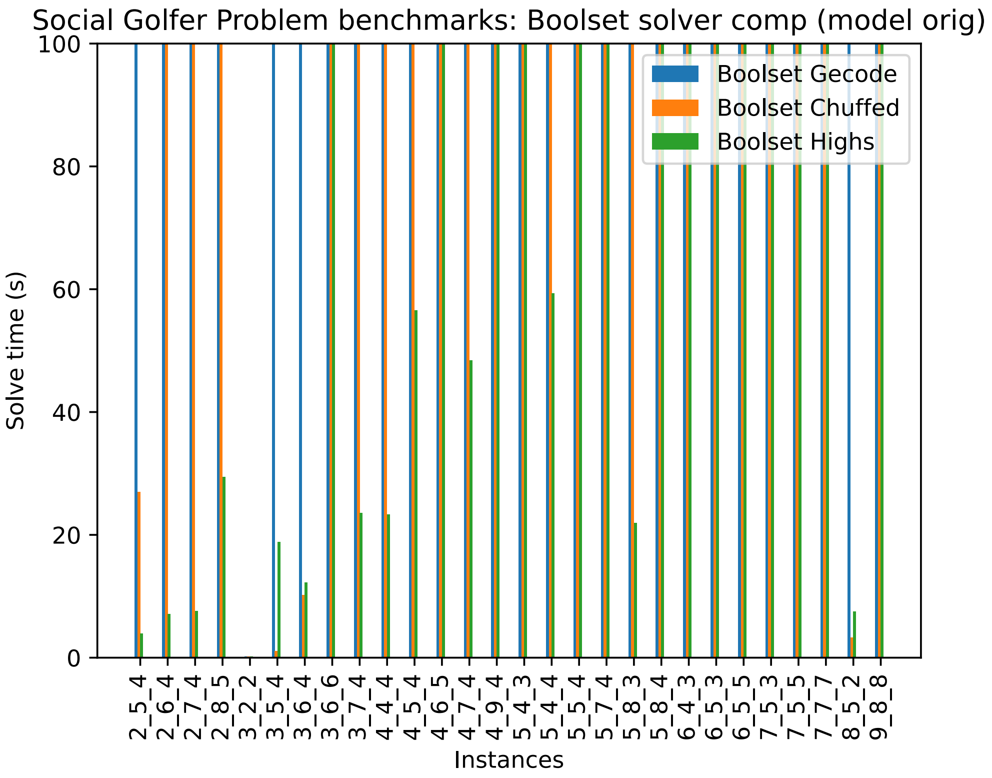


Figure 12: SGP BoolSet

C Gecode versus Choco solver on Steiner and Golfers problems

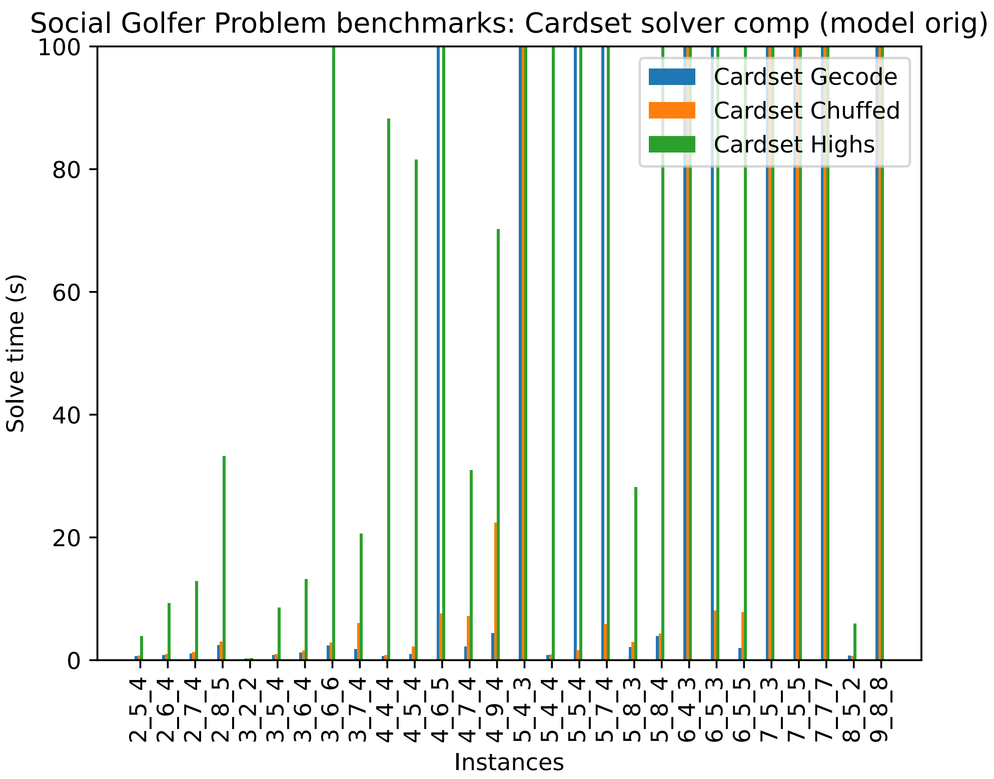


Figure 13: SGP CardSet

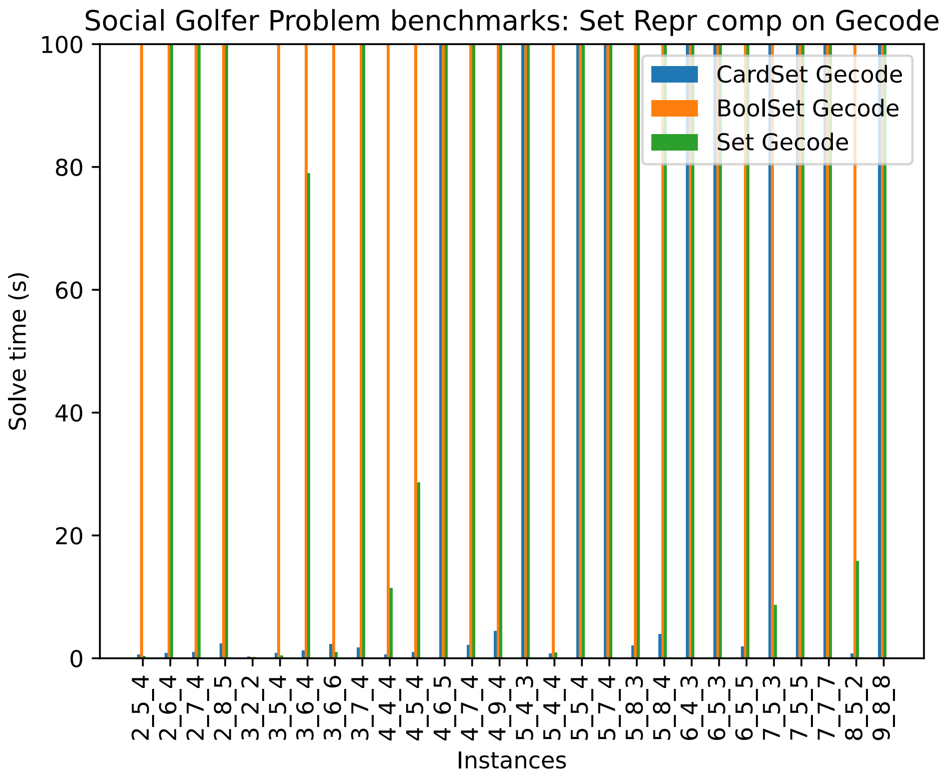


Figure 14: SGP on Gecode: CardSet vs BoolSet vs Set

Table 9: Steiner's Triple Problem solving times comparison (in seconds).

Instance	Gecode				Choco			
	set	boolset	CardSet		set	boolset	CardSet	
			opt	nem			opt	nem
03	0.091	0.119	0.252	0.276	0.949	1.27	1.045	1.309
04	0.119	0.097	0.395	0.391	1.105	2.455	1.77	1.644
05	0.117	0.257	0.619	0.634	1.14	2.748	1.956	1.783
06	0.097	0.134	0.284	0.286	1.144	2.006	1.589	1.74
07	0.115	0.144	0.287	0.297	1.754	3.958	2.952	2.412
08	3.156	89.951	3.748	3.145	13.866	66.681	31.851	12.88
09	0.111	180	1.191	2.741	180	32.029	63.315	180
10	180	180	180	180	180	180	180	180
11	180	180	180	180	180	180	180	180
12	180	180	180	180	180	180	180	180
13	180	180	180	180	180	180	180	180
14	180	180	180	180	180	180	180	180
15	0.142	180	180	180	180	180	180	180
16	180	180	180	180	180	180	180	180
17	0.139	180	180	180	180	180	180	180
18	180	180	180	180	180	180	180	180
19	180	180	180	180	180	180	180	180

Table 10: Social Golfers Problem solving times comparison (in seconds).

Instance	Gecode				Choco			
	set	boolset	CardSet		set	boolset	CardSet	
			opt	nem			opt	nem
2_5_4	0.538	180	0.971	0.855	180	180	101.689	11.842
2_6_4	110.548	180	0.813	0.82	180	180	180	180
2_7_4	180	180	1.143	0.947	180	45.181	180	180
2_8_5	180	180	2.292	2.169	180	180	180	180
3_2_2	0.196	0.262	0.279	0.292	1.492	1.655	1.338	1.531
3_5_4	0.597	180	1.249	1.071	180	180	180	180
3_6_4	78.311	180	1.132	0.96	180	180	180	180
3_6_6	0.974	180	2.27	2.252	180	180	180	180
3_7_4	180	180	1.627	1.428	180	180	180	180
4_4_4	11.362	180	0.562	0.558	180	180	180	180
4_5_4	28.813	180	0.927	1.069	180	180	180	180
4_6_5	180	180	180	180	180	180	180	180
4_7_4	180	180	2.09	1.841	180	180	180	180
4_9_4	180	180	4.942	4.459	180	180	180	180
5_4_3	180	180	180	180	180	180	180	180
5_4_4	0.971	180	0.824	0.594	180	89.876	180	180
5_5_4	180	180	180	180	180	180	180	180
5_7_4	180	180	180	180	180	180	180	180
5_8_3	180	180	2.05	1.996	180	180	180	180
5_8_4	180	180	180	180	180	180	180	180
6_4_3	180	180	180	180	180	180	180	180
6_5_3	180	180	180	180	180	180	180	180
6_5_5	180	180	3.905	4.481	180	180	180	180
7_5_3	8.872	180	180	180	180	180	180	180
7_5_5	180	180	180	180	180	180	180	180
7_7_7	180	180	180	180	180	180	180	180
8_5_2	15.689	180	0.668	0.833	180	7.722	180	180
9_8_8	180	180	180	180	180	180	180	180

`GeCode` seems to outperformed `Choco` on *CardSet* for most instances of the 2 problems.