# SUNNY-CP : a Sequential CP Portfolio Solver

Roberto Amadini, Maurizio Gabbrielli, Jacopo Mauro

# SUNNY-CP: a Sequential CP Portfolio Solver

[Tool Paper]

Roberto Amadini
University of Bologna/INRIA,
amadini@cs.unibo.it

Maurizio Gabbrielli
University of Bologna/INRIA,
gabbri@cs.unibo.it

Jacopo Mauro
University of Bologna/INRIA,
jmauro@cs.unibo.it

## ABSTRACT

The Constraint Programming (CP) paradigm allows to model and solve Constraint Satisfaction / Optimization Problems (CSPs / COPs). A CP Portfolio Solver is a particular constraint solver that takes advantage of a portfolio of different CP solvers in order to solve a given problem by properly exploiting Algorithm Selection techniques. In this work we present `sunny-cp`: a CP portfolio for solving both CSPs and COPs that turned out to be competitive also in the MiniZinc Challenge, the reference competition for CP solvers.

## Categories and Subject Descriptors

I.2 [**Computing Methodologies**]: Artificial Intelligence—*Constraint Programming, Algorithm Portfolios, Solvers and Tools.*

## 1. INTRODUCTION

*Constraint Programming* (CP) is a declarative paradigm that enables expressing relations between different entities in the form of constraints that must be satisfied. The main goal of CP is to model and solve *Constraint Satisfaction Problems* (CSPs) as well as *Constraint Optimization Problems* (COPs) [34]. Solving a CSP means finding a solution that satisfies all the constraints of a given problem. COPs can be instead regarded as generalized CSPs where we are not only interested in finding a solution, but also in minimizing (or maximizing) a given objective function. In the following, with the term '*CP problem*' we will refer either to a CSP or to a COP.

One of the more recent trends in CP —especially in the SAT field— is solving a given problem by using a *portfolio* approach [15, 33]. This is a general methodology that combines a number of different algorithms in order to get an overall better algorithm. More precisely, a *portfolio solver* uses a collection of $m > 1$ constituent solvers $s_1, \ldots, s_m$ in order to obtain a globally better solver. When a new, unseen problem $p$ comes, the portfolio solver tries to predict the best constituent solver(s) $s_1, \ldots, s_k$ (with $1 \leq k \leq m$) for solving $p$ and then runs such solver(s) on $p$.

Despite their proven effectiveness, portfolio solvers are rarely used in practice and usually restricted to the SAT field (e.g., see [6, 17, 23, 24, 27, 35, 42]). As far as the CP field is concerned, the first portfolio solver was CPHydra [31] that in 2008 demonstrated its effectiveness by winning the International Constraint Solver Competition. Unfortunately, the actual use of CPHydra is nowadays limited, since it uses a restricted number of dated constituent solvers and it can solve only CSPs encoded in the XML-based language XCSP [36]. A more recent portfolio approach is Proteus [21]. It does not rely purely on CSP solvers, but may decide to encode a CSP instance into SAT. Unfortunately, even Proteus is not able to solve COPs and still relies on the XCSP format.

More recently (in 2013) a portfolio solver built on top of Numberjack platform [18] attended the *MiniZinc Challenge* (MZC) [39], which is nowadays the only international competition for evaluating the performances of CP solvers. This approach does not discriminate between the constituent solvers but launches in parallel all of them. This solver, however, achieved rather poor results, perhaps due to the limited number of solvers and to some issues in parsing the *FlatZinc* language [28] (i.e., the input format of the MZC).

Another recent solver is `sunny-csp`, based on the SUNNY approach [4]. `sunny-csp` is a CSP portfolio solver that supports *MiniZinc* [30] —nowadays the de-facto standard to model CP problems— and XCSP. Originally SUNNY was intended to tackle only CSPs but, due to its performance and flexibility, it was also adapted to solve COPs [3, 5].

In this paper we merge these two lines of research by proposing `sunny-cp`: a new tool aimed at solving a generic CP problem. The aim of this paper is not to propose a new portfolio approach, but to describe and provide a flexible, configurable, and usable CP portfolio solver that can be set up and executed just like a regular individual CP solver. To the best of our knowledge, `sunny-cp` is the only sequential portfolio solver able to solve generic CP problems and it was the only portfolio solver that attended the MZC 2014.

The empirical evaluations in [3–5] have already proven the effectiveness of SUNNY algorithm when validated on heterogeneous and large benchmarks (i.e., about 500 or more problems) using a solving timeout of 1800 seconds. Conversely, in the MZC the time cap for constraint solving is restricted to 900 seconds and the test set is limited (100 instances each). Nevertheless, despite that the MZC is not the ideal scenario for a portfolio solver, we show that `sunny-cp` was competitive even in this setting.

*Paper Structure.* In Section 2 we recall the underlying SUNNY algorithm. In Section 3 we describe the architecture of `sunny-cp` while in Section 4 we examine the results it achieved in the MZC 2014. In Section 5 we discuss the related work while in Section 6 we draw some concluding remarks.

## 2. THE SUNNY ALGORITHM

In this section we provide an overview of SUNNY [4], the algorithm on which `sunny-cp` relies. For a more detailed explanation of SUNNY we refer the reader to [3–5].

SUNNY is a *lazy* portfolio approach originally tailored for CSPs. The basic idea behind SUNNY is to exploit instance similarities to run just a small but promising schedule of solvers. Given a CSP $p$ and a portfolio $\Pi$, it uses a *$k$-Nearest Neighbours* ($k$-NN) algorithm to select from a set of training instances a subset $N(p,k)$ of the $k$ instances closer to $p$ according, for simplicity, to the Euclidean distance. Then, on-the-fly, it computes a schedule of solvers by considering the smallest sub-portfolio $S \subseteq \Pi$ able to solve the maximum number of instances in the neighbourhood $N(p,k)$ within a timeout T and by allocating to each solver of $S$ a time proportional to the number of solved instances in $N(p,k)$. SUNNY also allocates to a *backup solver*, i.e., a solver of the portfolio aimed to handle exceptional circumstances like the premature failures of a constituent solver, an amount of time proportional to the number of instances not solved in $N(p,k)$ within the timeout.

Example 1 reports a running example of how SUNNY works on a given CSP.

**Example 1** *Let us suppose we have to solve a given CSP $p$ by means of a portfolio $\Pi = \{s_1, s_2, s_3, s_4\}$, where the backup solver is $s_3$, the solving timeout is $T = 1800$ seconds, the neighborhood of $p$ (of size $k = 5$) is $N(p,k) = \{p_1, ..., p_5\}$, and the runtimes of the solvers of $\Pi$ on $N(p,k)$ are defined as listed in Table 1.*

| | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ |
|---|---|---|---|---|---|
| $s_1$ | $T$ | $T$ | **3** | $T$ | **278** |
| $s_2$ | $T$ | **593** | $T$ | $T$ | $T$ |
| $s_3$ | $T$ | $T$ | **36** | **1452** | $T$ |
| $s_4$ | $T$ | $T$ | $T$ | **122** | **60** |

Table 1: Runtimes (in seconds). $T$ means the solver timeout.

*The minimum size sub-portfolios that allow to solve the most instances (i.e., 4 instances) are $\{s_1, s_2, s_3\}$, $\{s_1, s_2, s_4\}$, and $\{s_2, s_3, s_4\}$. SUNNY selects $S = \{s_1, s_2, s_4\}$ since it has a lower average solving time (1270.4 sec., to be precise). Since $s_1$ and $s_4$ solve 2 instances, $s_2$ solves 1 instance, and $p_1$ is not solved by any solver within $T$ seconds, the solving time window $[0, T]$ is partitioned in $2+2+1+1 = 6$ slots: 2 assigned to $s_1$ and $s_4$, 1 slot to $s_2$, and 1 to the backup solver $s_3$. After sorting the solvers by increasing solving time in the neighborhood, SUNNY executes in order the solvers $s_4$, $s_1$, $s_3$, and $s_2$ for respectively* 600, 600, 300, 300 *seconds.*

The promising performances achieved by SUNNY [4] on exhaustive benchmarks of CSPs have led to the development of `sunny-csp`, a CSP portfolio solver built on top of SUNNY by using 12 solvers (i.e., BProlog, Fzn2smt, CPX, G12/FD, G12/LazyFD, G12/MIP, Gecode, izplus, MinisatID, Mistral, and OR-Tools), a set of 155 features, and $k = 16$.

Motivated by the encouraging results of `sunny-csp`, SUNNY was adapted in order to deal with COPs. A modification was needed because when optimization problems are considered the dichotomy solved/not solved of CSP is no longer suitable. Indeed, a COP solver can yield sub-optimal solutions or even give the optimal one without being able to prove its optimality. In order to evaluate the performances of different COP solvers in [3] a new metric `score` was introduced to take into account the quality of the solutions. This metric gives to each solver a score in $[0.25, 0.75]$ linearly proportional to the distance between the best solution it finds and the best known solution. An additional reward (`score` $= 1$) is given if the solver is able to prove optimality while a punishment (`score` $= 0$) is given if the solver does not provide answers (or it gives an incorrect answer).

The adaptation of SUNNY from CSPs to COPs was therefore based on the notions of score and optimization time, instead of solved instances and solving time. In this case, SUNNY selects the sub-portfolio $S \subseteq \Pi$ that maximizes the `score` in the neighborhood and allocates to each solver a time in $[0, T]$ proportional to its total score in $N(p,k)$. In particular, while in the CSP version SUNNY allocates to the backup solver an amount of time proportional to the number of instances not solved in $N(p,k)$, in the COP version it assigns to it a slot of time proportional to $k - h$ where $h$ is the maximum `score` achieved by the sub-portfolio $S$. While for CSPs the final schedule is obtained by sorting the solvers by increasing solving time, for COPs the sorting is computed by considering the time needed to prove optimality.

Example 2 concludes the Section by providing an illustrative example of how SUNNY works on a given COP.

**Example 2** *Let us suppose that $\Pi = \{s_1, s_2, s_3, s_4\}$, the backup solver is $s_3$, $T = 1000$ seconds, $k = 3$, $N(p,k) = \{p_1, p_2, p_3\}$, and the scores/optimization times are defined as listed in Table 2.*

| | $p_1$ | $p_2$ | $p_3$ |
|---|---|---|---|
| $s_1$ | $(\mathbf{1}, 150)$ | $(0.25, 1000)$ | $(\mathbf{0.75}, 1000)$ |
| $s_2$ | $(0, 1000)$ | $(\mathbf{1}, 10)$ | $(0, 1000)$ |
| $s_3$ | $(1, 100)$ | $(0.75, 1000)$ | $(0.7, 1000)$ |
| $s_4$ | $(0.75, 1000)$ | $(0.75, 1000)$ | $(0.25, 1000)$ |

Table 2: (score, time) of each solver $s_i$ for every COP $p_j$.

*The minimum size sub-portfolio that allows to reach the highest score $h = 1+1+0.75 = 2.75$ is $\{s_1, s_2\}$. On the basis of the sum of the scores reached by $s_1$ and $s_2$ in $N(p,k)$ (resp. 2 and 1) the slot size is $t = T/(2 + 1 + (k - h)) = 307.69$ seconds. The time assigned to $s_1$ is $2 * t = 615.38$ while for $s_2$ is $1 * t = 307.69$. The remaining* 76.93 *seconds are finally allocated to the backup solver $s_3$. After sorting the solvers by increasing optimization time, SUNNY executes first $s_2$ for* 615.38 *seconds, then $s_3$ for* 76.93 *seconds, and finally $s_1$ for* 307.69 *seconds.*

## 3. SUNNY-CP

In this Section we illustrate the architecture of `sunny-cp`. Since the MiniZinc 1.6 platform [28] is required for processing the MiniZinc input files, we initially set up the portfolio of `sunny-cp` with the solvers coming with this suite, namely: CPX, G12/FD, G12/LazyFD, and G12/CBC. Then, in order to get an heterogeneous portfolio, we included four more

solvers disparate in their nature: Gecode [13] (FD solver and gold medallist of the 2008–2012 MZCs), MinisatID [10] (SAT-based solver), Chuffed (Lazy Clause CP solver and best solver in the MZCs 2012–2014[1]), and G12/Gurobi (MIP solver). Clearly, it would have been possible to add a number of other solvers. However, from the experimental investigations conducted in [1, 3], it turned out that using too large portfolios may be ineffective or sometimes even harmful. We therefore decided to use just the eight solvers mentioned above, while still providing the opportunity to arbitrarily change the portfolio composition.

Figure 1 summarizes the step-by-step execution flow of the framework from the input CP problem to the final output outcome. The input of `sunny-cp` consists of the problem instance $p$ to be solved, the size $k$ of the neighbourhood used by the underlying $k$-NN algorithm, the solving timeout $T$, and the solver $B$ of the portfolio to be used as backup solver. Despite the portfolio described above is fixed, it is still possible for the end user to select only a subset of its solvers. In addition, as described below, the user can also specify the knowledge base to use for the solver(s) selection.

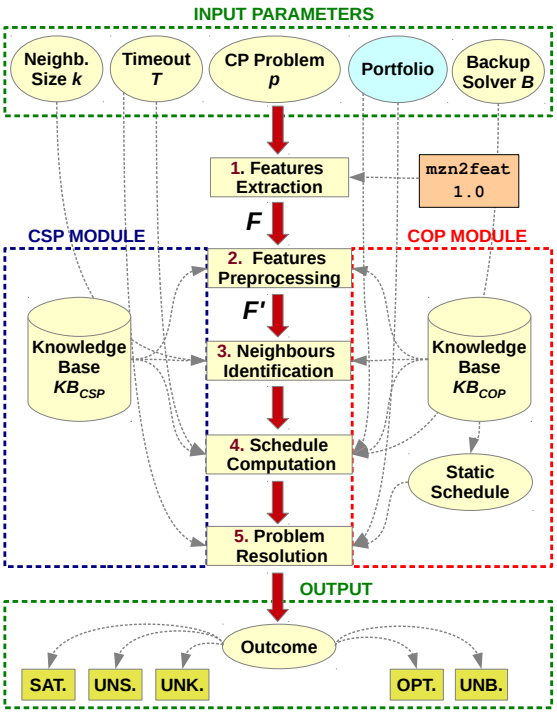


Figure 1: `sunny-cp` architecture.

The first step —that does not discriminate between CSPs and COPs— concerns the features extraction. Given a CP problem $p$ in input, this process identifies a *feature vector*, i.e., a collection of numerical attributes (e.g., number of variables, number of constraints, etc.) that characterize the instance $p$. Having a good set of features is a crucial step for boosting the performances of a portfolio solver. In our case, the features are used by the $k$-NN algorithm to find within the training set the $k$ instances closer to $p$ according to the Euclidean distance. Instead of using the whole set of 155 features of `sunny-csp` [4], `sunny-cp` makes use of the new features extractor `mzn2feat-1.0` [29]. This tool is essentially a new version of the extractor `mzn2feat` [2] designed to be more portable, light, fast, and flexible. Indeed, `mzn2feat-1.0` does not compute features based on graph measures since this process could be very time/space consuming. Moreover it does not compute solver-specific or dynamic features, thus allowing the features to be portable and the extractor to be decoupled from a particular solver and from the given machine on which it is executed. In total, `mzn2feat-1.0` extracts 95 features.

At step 2, the execution flow branches: if $p$ is a CSP instance, `sunny-cp` uses a knowledge base $KB_{CSP}$, otherwise the knowledge base $KB_{COP}$ is selected. Basically, a knowledge base can be seen as a map that associates to each CP problem a body of information relevant for the resolution process. Every CP problem of the knowledge base belongs to a training set of already known instances, and for each CP problem the relevant information are essentially two: its feature vector and the solving outcome of each constituent solver on it. `sunny-cp` already comes with default knowledge bases, constructed by collecting two different datasets $\Delta_{CSP}$ and $\Delta_{COP}$ of CSP and COP instances retrieved from the CP instances of the MiniZinc 1.6 benchmarks, the MZCs 2012/13, and the International CSP Solver Competitions. More in detail, $\Delta_{CSP}$ (resp. $\Delta_{COP}$) contains 5524 (resp. 4864) instances, to each of which is associated a feature vector of 78 (resp. 88) features and the performances of each solver of the portfolio on it. The feature vectors of $KB_{CSP}$ and $KB_{COP}$ differ in size since from the original 95 features extracted by `mzn2feat-1.0` we removed all the constant features. Moreover, since the performances of the $k$-NN algorithm can be drastically degraded by the presence of noisy or irrelevant features [11], all the features values are scaled in the range $[-1, 1]$. It is worth noticing that `sunny-cp` allows also the use of other knowledge bases given as input parameters. To ease the task of defining a customized knowledge base, `sunny-cp` comes with some utilities that help the user to assemble a knowledge base starting from a set of (not normalized) feature vectors and the performances of the solvers of the portfolio on the training instances.

The original feature vector $F$ of the instance $p$ is normalized in step 2 by exploiting the information of the corresponding knowledge base. The resulting normalized vector $F'$ is then used in step 3 to identify the $k$ nearest neighbours of $p$, i.e, the first $k$ instances of the selected knowledge base for which the corresponding normalized feature vector has a smaller Euclidean distance from the normalized vector $F'$.[2] The neighbourhood size $k$ is an input parameter which is set to $k = 70$ unless otherwise specified. We have chosen this default value since it is close to the square root of the default knowledge bases size. Indeed, the choice of $k$ is very critical: a simple initial approach consists in setting $k = \sqrt{n}$ where $n$ is the number of training samples [11].

In step 4, the SUNNY algorithm described in Section 2 is used to compute the schedule of solvers to be executed for solving $p$. The selected solvers will be executed sequentially in step 5, according to the given time limit. The default timeout $T$ is 1800 seconds (i.e., the one used in the last In-

[1]Chuffed is not eligible for prizes, being developed by the challenge organizers.

[2]The Euclidean distance is used for simplicity following the approach of CPHydra [31].

ternational CSP Competition [9]) while the default backup solver $B$ is Chuffed, since it has proven to be the best solver on the default training instances. However, both $T$ and $B$ are options that can be specified by the end user. If a solver aborts its execution prematurely (e.g., due to memory overflows or unsupported constraints) its remaining execution time is allocated to the next scheduled solver, if any, or to a not scheduled solver of the portfolio otherwise.

Note that in case $p$ is a CSP the scheduled solvers are executed independently, i.e., there is no cooperation and communication between them.[3] If instead $p$ is a COP instance, `sunny-cp` exploits the best solution found by a solver for narrowing the search space of the following ones. If $p$ is a CSP, `sunny-cp` may output three alternatives: satisfiable (a solution exists for $p$), unsatisfiable ($p$ has no solutions) or unknown (`sunny-cp` is not able to say anything about $p$). If $p$ is a COP, there are two more alternatives: `sunny-cp` can be able to prove the optimality of the solution found or even to prove the unboundedness of $p$.

We would like to underline that, according to the methodology introduced in [5], `sunny-cp` also allows to solve a given COP by first running a precomputed schedule of solvers in the first $C \leq T$ seconds. The purpose of this static schedule is essentially to find as many good solutions as possible in the first $C$ seconds, so as to ensure a "warm start" to the solvers of the dynamic schedule computed by SUNNY, that will be executed for the remaining $T - C$ seconds. The use of the static schedule might speed up the search, potentially allowing `sunny-cp` to outperform the best solver of the portfolio. The static schedule is empty by default, since its computation may involve a non-trivial off-line phase. Nevertheless, the user has the possibility to set his own static schedule or even to choose among other static schedules that we have already pre-computed.

`sunny-cp` is mainly written in Python. It requires the independent installation of the features extractor `mzn2feat-1.0` and of each constituent solver. The source code used in the MZC 2014 is available at [40].

## 4. VALIDATION

The performances of SUNNY against different state-of-the-art approaches have been extensively studied in [3–5] by using big test sets (i.e., about 500 instances or more) and a fairly large solving timeout ($T = 1800$ seconds). All these empirical evaluations has proven the effectiveness of SUNNY, in particular w.r.t. the constituent solvers of the portfolio that have been always greatly outperformed. In this section we show instead how `sunny-cp` behaved in the MZC 2014. In this setting things are different: the test set is far smaller (i.e., 100 instances, almost always satisfiable), the timeout is an half (900 seconds), and especially a different evaluation metric is adopted w.r.t. those described in [3–5].

Different metrics can actually be adopted to evaluate the effectiveness of CP solvers. Given a solving timeout $T$ and a test set of CSP instances, the performance of a solver is usually measured in terms of number of instances solved and average solving time [1, 37]. More formally, we can define the metrics `proven` and `time` as follows: if a solver $s$ is able to prove the (un-)satisfiability of $p$ in $t < T$ seconds then $\texttt{proven}(s, p) = 1$ and $\texttt{time}(s, p) = t$, otherwise $\texttt{proven}(s, p) = 0$ and $\texttt{time}(s, p) = T$ . A natural generalization for COPs can be obtained by setting $\texttt{proven}(s, p) = 1$ and $\texttt{time}(s, p) = t$ if $s$ proves in $t < T$ seconds the optimality of a solution for $p$, the unsatisfiability of $p$ or its unboundedness. Otherwise, $\texttt{proven}(s, p) = 0$ and $\texttt{time}(s, p) = T$, even when sub-optimal solutions are found.

The scoring procedure of MZC is instead based on a *Borda count* voting system [8] where each CP problem is treated like a voter who ranks the solvers. Each solver gets a score proportional to the number of solvers it beats. A solver $s$ scores points on problem $p$ by comparing its performance with each other solver $s'$ on problem $p$. If $s$ gives a better answer than $s'$ then it scores 1 point, if it gives a worse solution it scores 0 points. If $s$ and $s'$ give indistinguishable answers then the scoring is based on the solving time. In particular, $s$ scores either: 0, if it fails to find any solution or fails to prove unsatisfiability; 0.5, if both $s$ and $s'$ complete the search in 0 seconds; $\texttt{time}(p, s')/(\texttt{time}(p, s) + \texttt{time}(p, s'))$ otherwise.[4]

| SOLVER | SCORE |
|---|---|
| Chuffed-free* | 1324.02 |
| *OR-Tools-par (GOLD MEDAL)* | *1084.97* |
| Opturion CPX-free (SILVER MEDAL) | 1079.02 |
| **sunny-cp-presolve-open** | **1064.46** |
| *Choco-par (BRONZE MEDAL)* | *1005.61* |
| *iZplus-par* | *994.32* |
| **sunny-cp-open** | **967.14** |
| G12/LazyFD-free* | 782.78 |
| HaifaCSP-free | 779.72 |
| *Gecode-par* | *720.97* |
| SICStus Prolog-fd | 708.51 |
| Mistral-free | 703.56 |
| MinisatID-free* | 587.24 |
| Picat SAT-free | 586.64 |
| JaCoP-fd | 549.24 |
| G12/FD-free* | 526.26 |
| Picat CP-free | 402.88 |
| Concrete-free | 353.24 |

Table 3: MZC 2014 open track. Parallel solvers are in italics, while the solvers included in `sunny-cp` are marked with *.

The results of the *open search category* of the MZC 2014 are summarized in Table 3. Two versions off `sunny-cp` attended the competition: *sunny-cp-open* and *sunny-cp-presolve-open*. The first is the default `sunny-cp` solver as described in Section 3. The second is instead the variant that runs for every COP a static schedule in the first 30 seconds. In particular the solvers Chuffed, Gecode and CPX are executed for 10 seconds each.

Before discussing the results, it is appropriate to make a few remarks. First, the open class includes also parallel solvers. This can be disadvantageous for sequential solvers like `sunny-cp` that does not exploit more computation units. Second, in the MZC the solving time $\texttt{time}(p, s)$ refers to the time needed by $s$ for solving the FlatZinc model $p_s$ resulting from the conversion of the original MiniZinc model $p$ to $p_s$, thus ignoring the solver dependent time needed to flatten

[3] `sunny-cp` views the constituent CSP solvers as "black boxes'. The lack of a standard protocol to extract and share knowledge between solvers makes it hard to communicate potentially useful information like no-goods and cuts.

[4] For more details about MZC rules we refer the interested reader to the MiniZinc website [28].

$p$ in $p_s$. The choice of discarding the conversion penalizes a portfolio solver. Indeed, given the heterogeneous nature of our portfolio, `sunny-cp` can not use global redefinitions that are suitable for all its constituent solvers. Therefore, differently from all the other solvers of the open search track, the result of `sunny-cp` were computed by considering not only the solving time of the FlatZinc models but also all the conversion times of the MiniZinc input, including an initial conversion to FlatZinc required for extracting the features of $p$. Moreover, as we detail later, the grading methodology of MZC further penalizes a portfolio solver because in case of ties it may assign a score that is disproportionate w.r.t. the solving time difference. This is a drawback for `sunny-cp` since, even if it selects the best constituent solver, it requires an additional amount of time for extracting the features and for the solver selection. This holds especially in the presence of a clear dominant solver, like Chuffed for the MZC 2014. All these difficulties have been recognized by the organizers, which awarded to sunny-cp an *honorable mention*.[5]

Let us now analyze the results. As said, if on the one hand having Chuffed in the portfolio was undoubtedly advantageous for us, on the other hand this can also be counterproductive. Indeed, it is impossible to beat it in the numerous times in which it is the best solver, even if `sunny-cp` selects it to solve the instance. Moreover, note that the other solvers of `sunny-cp` enrolled in the Challenge have not achieved excellent performance: G12/LazyFD is $8^{\text{th}}$, MinisatID is $13^{\text{th}}$ and G12/FD is $16^{\text{th}}$. Gecode-par is $10^{\text{th}}$, but `sunny-cp` didn't use this version since Gecode-par is a parallel solver.[6] Finally, G12/CBC, G12/Gurobi, and CPX have not attended the challenge.[7] This setting is inevitably detrimental for a portfolio. There is a clearly dominant constituent solver, while the others have a rather low contribution. To boost the performances of `sunny-cp` we could have used an *'ad hoc'* training set for the MZC[8] but we instead preferred to measure its performance by using the default knowledge base because we believe that such solver is more robust, less prone to overfitting, and more suitable for scenarios where Chuffed is not the dominant solver. Indeed, despite the MZC is surely a valuable setting for evaluating CP solvers, it is important that a solver is designed to be good at solving (class of) CP problems rather than being well ranked at the MZC.

sunny-cp-presolve-open achieved the $4^{\text{th}}$ position, reaching 97.32 points more than sunny-cp-open ($7^{\text{th}}$). As shown also in [5], the static schedule resulted in an increase in performance. Here, however, the score difference is mainly due to the higher speed of sunny-cp-presolve-open in case of indistinguishable answer, rather than the higher quality of the solutions. Indeed, in just 9 cases sunny-cp-presolve-open gave a better answer than sunny-cp-open, while there are also 6 cases in which sunny-cp-open is better.

If we just focus on the `proven` and `time` metrics introduced earlier, we can observe significant variations in the rank. Table 4 shows that, by considering `proven`, the two versions of `sunny-cp` would be $2^{\text{nd}}$ and $3^{\text{rd}}$. Chuffed is still the best solver, but sometimes it is outperformed by `sunny-cp`. For instance, unlike Chuffed, `sunny-cp` is able to solve all the 25 CSPs of the competition. Again, note that `sunny-cp` is not biased towards Chuffed. Figure 2 shows that the number of times (in percentage) a constituent solver of `sunny-cp` completes the search. As can be seen, almost all the solvers find an optimal solution at least one time. The contribution of Chuffed is not massive: about one third of the instances. Looking at the average `time` results of Table 4, we notice that `sunny-cp` is overtaken by Opturion. This is not surprising: `sunny-cp` proves more optima, but Opturion is on average faster.

| SOLVER | #proven | SOLVER | time |
|---|---|---|---|
| Chuffed-free | 68 | Chuffed-free | 331.27 |
| **sunny-cp-presolve-open** | **59** | Opturion CPX-free | 461.05 |
| **sunny-cp-open** | **57** | **sunny-cp-presolve-open** | **469.86** |
| Opturion CPX-free | 53 | **sunny-cp-open** | **488.58** |
| OR-Tools-par | 50 | OR-Tools-par | 491.29 |
| G12/LazyFD-free | 46 | G12/LazyFD-free | 511.28 |
| MinisatID-free | 46 | Choco-par | 553.83 |
| Picat SAT-free | 44 | Gecode-par | 563.65 |
| Choco-par | 43 | MinisatID-free | 566.24 |
| SICStus Prolog-fd | 41 | iZplus-par | 570.30 |
| iZplus-par | 41 | Picat SAT-free | 570.31 |
| Gecode-par | 40 | SICStus Prolog-fd | 598.65 |
| HaifaCSP-free | 39 | HaifaCSP-free | 599.85 |
| Mistral-free | 37 | Mistral-free | 606.75 |
| JaCoP-fd | 31 | JaCoP-fd | 659.17 |
| G12/FD-free | 27 | G12/FD-free | 693.99 |
| Concrete-free | 22 | Concrete-free | 737.02 |
| Picat CP-free | 17 | Picat CP-free | 777.84 |

Table 4: MZC results considering `proven` and `time` metrics.

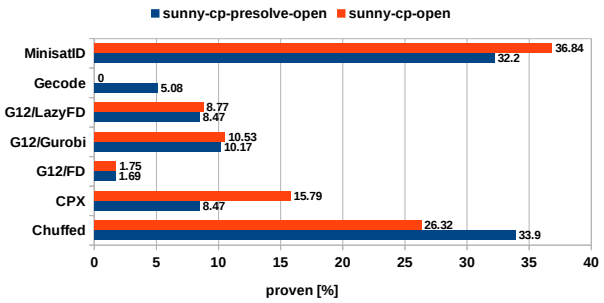


Figure 2: Solvers contribution in terms of `proven`.

Looking at the results on the individual instances, on 9 problems `sunny-cp` is able to outperform all its constituent solvers participating in the MZC. For 4 instances it is gold medalist, and for the *cyclic-rcpsp* problem (5 instances) it is the overall best solver of the competition.

We conclude this Section by proposing an alternative ranking system. As previously mentioned, we realized that, in case of indistinguishable answer between two solvers, in the MZC the gain of a solver depends on the runtime ratio rather than the runtime difference. For instance, let us suppose that two solvers $s_0$ and $s_1$ solve a problem $p$ in 1 and 2 second respectively. The score assigned to $s_0$ is $2/3 = 0.667$ while $s_1$ scores $1/3 = 0.333$. The same score would be reached by the two solvers if $\texttt{time}(p, s_1) = 2 * \texttt{time}(p, s_0)$. Hence, if for example $\texttt{time}(p, s_0) = 400$ and $\texttt{time}(p, s_1) = 800$, the difference between the scores of $s_0$ and $s_1$ would be the same

[5] `sunny-cp` was not eligible for prizes, since it contains solvers developed by the MZC organizers.

[6] Gecode-free, the sequential version of Gecode-par, in the open category would be ranked $12^{\text{th}}$.

[7] Opturion CPX [32] is based on CPX, but *it is not* CPX. Moreover the constituent solvers of the portfolio may be obsolete w.r.t. the version submitted to MZC 2014.

[8] As pointed out also in [39], the good results of a portfolio solver on a competition can be attributed to the fact that it could be over-trained.

even if the absolute time difference is 1 second in the first case, 400 seconds in the second. In our opinion, this scoring methodology could overestimate small time differences in case of easy instances, as well as underrate big time differences in case of medium and hard instances. A possible workaround to this problem is to assign each solver a score in $[0, 1]$ that, in case of indistinguishable answer between two solvers, is linearly proportional to the solving time difference. We then propose a modified metric `MZC-MOD` that differs from the MZC score since, when the answers of $s_0$ and $s_1$ are indistinguishable, it gives to the solvers a reward $\texttt{MZC-MOD}(p, s_i) = 0.5 + \frac{\texttt{time}(p,s_{1-i}) - \texttt{time}(p,s_i)}{2\,T}$ where $T$ is the timeout (i.e., 900 seconds for MZC). With this new metric, according to the previous examples, if $\texttt{time}(p, s_0) = 1$ and $\texttt{time}(p, s_1) = 2$, then the score difference is minimal since $\texttt{MZC-MOD}(p, s_0) = 0.5 + 1/2 * 900 = 0.501$ and $\texttt{MZC-MOD}(p, s_1) = 0.5 - 1/2 * 900 = 0.499$. In contrast, if $\texttt{time}(p, s_0) = 400$ and $\texttt{time}(p, s_1) = 800$ then the difference is proportionally higher: $\texttt{MZC-MOD}(p, s_0) = 0.722$ and $\texttt{MZC-MOD}(p, s_1) = 0.278$.

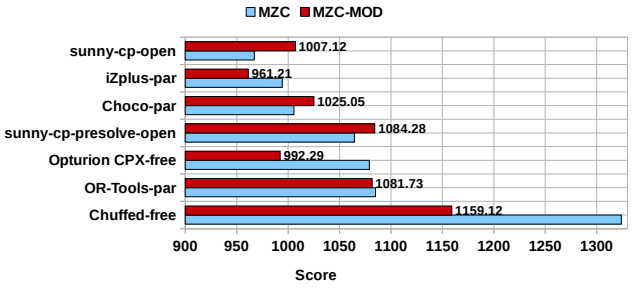


Figure 3: Score differences between MZC score and `MZC-MOD`.

Figure 3 depicts the effects of using `MZC-MOD` in place of MZC score w.r.t. the first seven positions of the ranking. The classification with the new score is much more compact and reflects the lower impact of the solving time difference in case of identical answers. Chuffed is firmly in the lead, but if compared to the original score loses about 165 points. Except for `sunny-cp` and Choco, all other approaches have a deterioration of performance. In particular, sunny-cp-presolve-open gains 19.82 points surpassing OR-Tools-par in the 2nd position. Even more noticeable is the score improvement of sunny-cp-open, that earns about 40 points and 2 positions in the ranking.

# 5. RELATED WORK

As already mentioned, the only other portfolio solver able to process MiniZinc files that has attended a MiniZinc Challenge is based on Numberjack [18]. It is a parallel solver that, differently from `sunny-cp`, launches concurrently all its constituent solvers, without making any solver selection and without extracting any feature. In this work we do not report a direct comparison with this solver since it is a parallel one and because it achieved rather poor results in MiniZinc Challenge 2013, perhaps due to some issues in parsing the FlatZinc language. Indeed, in the MZC 2013 open track Numberjack scored just 382.06 points while Chuffed scored 1034.63 points.

Other related solvers are CPHydra [31] and Proteus [21]. However, these solvers can handle only CSP instances (encoded in XCSP format).

For a comprehensive survey on portfolio approaches applied to SAT, planning, and QBF problems we refer the interested reader to [25], for CSP to [1], for ASP to [12]. Concerning the application of portfolio techniques to optimization problems we refer instead the reader to [3, 5, 16, 22, 38, 41]. Among the various proposal, certainly worth mentioning are the approaches that turned out to be effective in different solving challenges. For instance, the SAT portfolio solvers 3S [23], SATZilla [43], and CSHC [27] respectively won gold medals in the 2011, 2012, and 2013 editions of the SAT competition. CPHydra [31] was the winner of the 2008 International Constraint Solver Competition while the ASP portfolio solver claspfolio [19] won in different tracks in the 2009 and 2011 ASP competitions.

# 6. CONCLUSIONS

In this paper we described `sunny-cp`, a sequential portfolio solver able to solve both satisfaction and optimization problems defined in MiniZinc language. `sunny-cp` is aimed to provide a flexible, configurable, and usable CP portfolio solver that can be set up and executed by the end users just like an usual, single CP solver.

Since we already verified in former evaluations the effectiveness of `sunny-cp` when validated on heterogeneous and large sets of test instances, in this work we analyzed the results obtained by two versions of `sunny-cp` in the MiniZinc Challenge 2014, the reference competition for evaluating CP solvers. Despite the grading mechanism of the challenge may penalize a sequential portfolio solver, `sunny-cp` turned out to be competitive even when compared to parallel solvers, and it was sometimes even able to outperform state-of-the-art constraint solvers. In this regard, we also proposed and evaluated an alternative scoring system that, in case of indistinguishable quality of the solutions, takes into account the solving time difference in a more linear way.

The improvement of the state of the art proposed in this paper is not related to the introduction of a new approach, but concerns the presentation of a new tool that could serve as a baseline for future developments. Indeed, a major weakness of portfolio solvers is that their use is actually confined to annual competitions. In practice, they are often very difficult to use and compare. We therefore hope that `sunny-cp` can take a step forward in encouraging and disseminating the actual use of CP portfolio solvers.

As a future work we would like to extend the `sunny-cp` framework by adding new solvers and functionalities. There is plenty of lines of research that can be explored: e.g., the selection strategies of [24, 27, 31], the impact of using different distance metrics and features [26], the robustness towards the $k$ parameter of $k$-NN algorithm [4], the automated tuning of different solvers configurations [24], the use of benchmark generation techniques [20], the evaluation of different ranking methods [7, 14]. Certainly one of the most promising directions for further research is the extension of `sunny-cp` to a parallel environment, where multiple cores can be used to launch more than one constituent solver in parallel. In this setting it may be important to devise and evaluate cooperative strategies between the constituent solvers, by exploiting and sharing tokens of knowledge like the no-goods generated by the lazy solvers of the portfolio.

## Acknowledgments

We would like to thank the staff of the Optimization Research Group of NICTA (National ICT of Australia) for allowing us to use Chuffed and G12/Gurobi solvers, as well as for granting us the computational resources needed for building and testing `sunny-cp`.

## 7. REFERENCES

[1] R. Amadini, M. Gabbrielli, and J. Mauro. An Empirical Evaluation of Portfolios Approaches for Solving CSPs. In *CPAIOR*, 2013.

[2] R. Amadini, M. Gabbrielli, and J. Mauro. An Enhanced Features Extractor for a Portfolio of Constraint Solvers. In *SAC*, 2014.

[3] R. Amadini, M. Gabbrielli, and J. Mauro. Portfolio Approaches for Constraint Optimization Problems. In *LION*, 2014.

[4] R. Amadini, M. Gabbrielli, and J. Mauro. SUNNY: a Lazy Portfolio Approach for Constraint Solving. In *ICLP*, 2014.

[5] R. Amadini and P. J. Stuckey. Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers. In *CP*, 2014.

[6] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette. PeneLoPe, a Parallel Clause-Freezer Solver. In *SAT Challenge*, 2012.

[7] D. L. Berre and L. Simon. Preface. *JSAT*, 2006.

[8] Y. Chevaleyre, U. Endriss, J. Lang, and N. Maudet. A Short Introduction to Computational Social Choice. In *SOFSEM*, 2007.

[9] Third International CSP Solver Competition 2009. `http://www.cril.univ-artois.fr/CPAI09`.

[10] B. de Cat, B. Bogaerts, J. Devriendt, and M. Denecker. Model Expansion in the Presence of Function Symbols Using Constraint Programming. In *ICTAI*, 2013.

[11] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2Nd Edition)*. 2000.

[12] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. T. Schneider, and S. Ziller. A Portfolio Solver for Answer Set Programming: Preliminary Report. In *LPNMR*, 2011.

[13] Gecode website. `http://www.gecode.org`.

[14] A. V. Gelder. Careful Ranking of Multiple Solvers with Timeouts and Ties. In *SAT*, 2011.

[15] C. P. Gomes and B. Selman. Algorithm portfolios. *Artif. Intell.*, 2001.

[16] H. Guo and W. H. Hsu. A machine learning approach to algorithm selection for NP-hard optimization problems: a case study on the MPE problem. *Annals OR*, 2007.

[17] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 2009.

[18] E. Hebrard, E. O'Mahony, and B. O'Sullivan. Constraint Programming and Combinatorial Optimisation in Numberjack. In *CPAIOR*, 2010.

[19] H. Hoos, M. T. Lindauer, and T. Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *TPLP*, 2014.

[20] H. H. Hoos, B. Kaufmann, T. Schaub, and M. Schneider. Robust Benchmark Set Selection for Boolean Constraint Solvers. In *LION*, 2013.

[21] B. Hurley, L. Kotthoff, Y. Malitsky, and B. O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *CPAIOR*, 2014.

[22] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown. Algorithm Runtime Prediction: The State of the Art. *CoRR*, 2012.

[23] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm Selection and Scheduling. In *CP*, 2011.

[24] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. ISAC - Instance-Specific Algorithm Configuration. In *ECAI*, 2010.

[25] L. Kotthoff. Algorithm Selection for Combinatorial Search Problems: A Survey. *CoRR*, 2012.

[26] C. Kroer and Y. Malitsky. Feature filtering for instance-specific algorithm configuration. In *ICTAI*, 2011.

[27] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. In *IJCAI*, 2013.

[28] MiniZinc website. `http://www.minizinc.org`.

[29] mzn2feat-1.0 source code. `http://www.cs.unibo.it/~amadini/mzn2feat-1.0.tar.bz2`.

[30] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP*, 2007.

[31] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *AICS 08*, 2009.

[32] Opturion website. `http://www.opturion.com/cpx.html`.

[33] J. R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 1976.

[34] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. 2006.

[35] O. Roussel. ppfolio website. http://www.cril.univ-artois.fr/~roussel/ppfolio/.

[36] O. Roussel and C. Lecoutre. XML Representation of Constraint Networks: Format XCSP 2.1. *CoRR*, 2009.

[37] SAT Competition 2013 web site. http://satcompetition.org/2013/.

[38] K. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 2008.

[39] P. J. Stuckey, R. Becket, and J. Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 2010.

[40] Source code of sunny-cp. `https://github.com/jacopoMauro/sunny-cp/tree/mznc14`.

[41] O. Telelis and P. Stamatopoulos. Combinatorial Optimization through Statistical Instance-Based Learning. In *ICTAI*, 2001.

[42] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. In *CP*, 2007.

[43] L. Xu, F. Hutter, J. Shen, H. Hoos, and K. Leyton-Brown. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. SAT Challenge, 2012.