Hyperparameter Optimization of Constraint Programming Solvers

Hedieh Haddad[1*], Thibault Falque[2], Pierre Talbot[2], Pascal Bouvry[2]

[1*]Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Esch-sur-Alzette, Luxembourg.
[2]Faculty of Science, Technology and Medicine (FSTM), University of Luxembourg, Esch-sur-Alzette, Luxembourg.

*Corresponding author(s). E-mail(s): hedieh.haddad@uni.lu;
Contributing authors: thibault.falque@uni.lu; pierre.talbot@uni.lu; pascal.bouvry@uni.lu;

Abstract

The performance of constraint programming solvers is highly sensitive to the choice of their hyperparameters. Manually finding the best solver configuration is a difficult, time-consuming task that typically requires expert knowledge. In this paper, we introduce probe and solve algorithm, a novel two-phase framework for automated hyperparameter optimization integrated into the CPMpy library. This approach partitions the available time budget into two phases: a probing phase that explores different sets of hyperparameters using configurable hyperparameter optimization methods, followed by a solving phase where the best configuration found is used to tackle the problem within the remaining time.
We implement and compare two hyperparameter optimization methods within the probe and solve algorithm: Bayesian optimization and Hamming distance search. We evaluate the algorithm on two different constraint programming solvers, ACE and Choco, across 114 combinatorial problem instances, comparing their performance against the solver's default configurations.
Results show that using Bayesian optimization, the algorithm outperforms the solver's default configurations, improving solution quality for ACE in 25.4% of instances and matching the default performance in 57.9%, and for Choco, achieving superior results in 38.6% of instances. It also consistently surpasses Hamming distance search within the same framework, confirming the advantage of model-based exploration over simple local search. Overall, the probe and solve algorithm

offers a practical, resource-aware approach for tuning constraint solvers that yields robust improvements across diverse problem types.

Keywords: Constraint programming, algorithm configuration, Bayesian optimization, hyperparameter optimization, automated algorithm design

1 Introduction

In the field of combinatorial optimization, constraint programming (CP) stands out as a powerful paradigm for solving a wide range of complex problems. It finds applications in diverse domains including scheduling and resource management [1], network design, and logistics [2]. The core principle of CP involves modeling real-world problems as a set of variables, each with a specific domain of possible values, and a set of constraints that must be satisfied [2]. A CP solver then systematically searches for one or more solutions that adhere to all specified constraints.

While CP offers remarkable flexibility, the performance of a solver is heavily dependent on its hyperparameter tuning. The key hyperparameters include propagation levels [3], which determine the extent of constraint filtering and inference; and the search strategy [4], which dictates how the solver explores the solution space. The search strategy is primarily defined by the choice of variable and value selection heuristics, often referred to as branching rules [5]. In addition, the management of computational resources, such as time and memory limits [6, 7], while distinct from algorithmic hyperparameters, plays a critical role in ensuring practical solver performance.

These hyperparameters dictate how the solver explores the search space and manages computational resources. An appropriate choice of hyperparameters can lead to dramatic improvements in solver efficiency and solution quality.

However, manually identifying an optimal set of hyperparameters is a daunting task. The search space of possible hyperparameters is vast, and the ideal configuration often varies significantly from one problem instance to another. This manual tuning process is not only time-consuming but also prone to human error [8].

The general problem of finding a set of hyperparameters that improves solver performance, known as algorithm configuration, has a long history in solving hard combinatorial problems [8]. In recent years, this field has seen rapid progress under the banner of hyperparameter optimization (HPO), largely driven by its widespread adoption and success in machine learning (ML) [9]. Surprisingly, its application within the CP domain remains comparatively limited. The non-trivial task of adapting HPO techniques to the discrete and highly structured search spaces of CP solvers presents a unique research opportunity. These limitations give rise to three main challenges: (i) the combinatorial explosion of possible hyperparameters, (ii) the lack of generalization across problem instances, and (iii) the need for expert knowledge to achieve good performance [10].

For instance, in the ACE solver [11], the possible parameter combinations exceed 650 million, making exhaustive methods such as grid search computationally

infeasible. This complexity underscores the necessity of automated hyperparameter optimization methods capable of efficiently exploring such vast hyperparameter spaces.

To address these challenges, we introduce a framework named the probe and solve algorithm (PSA). The core intuition behind PSA is to automate the search for an optimal or near-optimal set of hyperparameters by splitting a given time budget into two distinct phases. First, a probing phase quickly explores a diverse set of hyperparameters by running the solver for short durations on the specific problem instance, gathering valuable performance data about which strategies work well. Second, the solving phase commits the entire remaining time budget to running the solver with the single best-performing configuration identified during probing. This two-phase approach intelligently manages the trade-off between exploring for good configurations and exploiting the best one found.

Basic HPO methods, such as grid search and random search, offer simple starting points but come with notable drawbacks [8]. Grid search exhaustively evaluates a predefined set of hyperparameters, making it computationally intractable as the number of parameters grows [12]. Random search provides better efficiency but lacks a guided strategy, often failing to concentrate on the most promising regions of the search space [13].

A more sophisticated and efficient alternative is Bayesian optimization (BO), a model-based approach that intelligently navigates the hyperparameter space [14]. BO works by building a probabilistic surrogate model (e.g., a Gaussian process [15]) of the objective function and uses an acquisition function to balance the trade-off between exploration (sampling uncertain, promising new configurations) and exploitation (refining known good configurations) [16]. This makes BO particularly well-suited for optimizing expensive black-box functions, such as the performance of a CP solver.

In this paper, we introduce a novel framework for the automated configuration of constraint solvers that bridges the gap between modern, ML-driven HPO techniques and the structured, combinatorial nature of CP.

The contributions of this paper are as follows:

- We propose PSA, a resource-aware, two-phase framework for HPO of constraint solvers under fixed time budgets (Section 3).
- We provide the first integration of the ACE solver into the CPMpy modeling library, enabling automated configuration of its 150,000+ parameter space (Section 4).
- We conduct a large-scale empirical study comparing PSA with BO and Hamming distance search across 114 XCSP3 benchmark instances using two solvers (ACE and Choco) (Section 4).
- We demonstrate that BO within PSA consistently outperforms both default configurations and Hamming distance search, improving solutions for ACE in 25.4% of instances and Choco in 38.6% (Section 5).

These findings offer a deeper understanding of how to steer adaptive solvers and can inform future research in automated solver design.

The remainder of this paper is organized as follows. Section 2 provides background on constraint programming and hyperparameter optimization methods. Section 3

presents PSA and describes its modular architecture and time management strategies. Section 4 outlines the experimental setup, including benchmark instances, solver configurations, and data collection procedures. Section 5 reports the main empirical results, comparing against the baselines, and analyzing their components. Section 6 interprets these findings and discusses their implications for automated solver configuration. Finally, section 7 concludes the paper and highlights avenues for future research.

2 Background

2.1 Constraint Programming

CP is a declarative paradigm for solving complex combinatorial problems. Its core principle is the separation of problem modeling from the solving algorithm, allowing users to state the problem's logic without specifying the exact steps to find a solution [17].

At its foundation, a CP model is defined as a constraint satisfaction problem (CSP) [18]. A CSP is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where:

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of variables.
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, where each $D(x_i) \subseteq \mathbb{Z}$ is the finite set of possible values for variable x_i.
- $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of constraints that restrict the allowed combinations of values for variables in their scope.

An assignment is a mapping that associates each variable $x_i \in \mathcal{X}$ with one value from its domain $D(x_i)$. A solution to a CSP is an assignment that satisfies all constraints in $\mathcal{C}$. Let $\mathcal{A}$ denote the set of all possible assignments:

$$\mathcal{A} = \{\, A \mid A : \mathcal{X} \to \bigcup_{x_i \in \mathcal{X}} D(x_i),\ A(x_i) \in D(x_i) \text{ for all } x_i \in \mathcal{X} \,\}.$$

Many real-world problems require finding not just *any* solution, but the *best* one according to some criterion. This extends the CSP into a constraint optimization problem (COP). A COP is formally defined as a quadruplet $(\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$, where $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a CSP and f is an *objective function*. This function maps every assignment of variables to an integer value. Formally, if $\mathcal{A}$ is the set of all possible assignments, the signature is $f : \mathcal{A} \to \mathbb{Z}$. The goal of a COP solver is to find a solution that satisfies all the constraints of $\mathcal{C}$ while minimizing (or maximizing) the value of f [17, 19].

Example: Bounded Knapsack Problem

A classic example of a COP is the bounded knapsack problem [20]. Given a set of items, each with a weight and a value, the goal is to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. A CP model for this can be formulated as follows:

- **Variables:** A set of N integer variables, $\mathcal{X} = \{x_1, x_2, \ldots, x_N\}$, where x_i represents the number of times item i is taken.
- **Domains:** Each variable x_i has the domain $D(x_i) = \{0, 1, \ldots, k_i\}$, where k_i is an upper bound on the quantity of item i.
- **Constraint:** The sum of the weights of the chosen items must not exceed the knapsack's capacity, W_{max}:

$$\sum_{i=1}^{N} w_i \cdot x_i \leq W_{max}$$

- **Objective function:** The goal is to maximize the total value of the items in the knapsack:

$$\text{maximize} \quad f(\mathcal{X}) = \sum_{i=1}^{N} v_i \cdot x_i$$

A solver's task is to find an assignment for $(x_1, \ldots, x_N)$ that satisfies the weight constraint and maximizes the objective function.

2.2 Hyperparameter Optimization Methods

HPO is the process of automating the selection of an optimal set of hyperparameters for a given learning algorithm or solver. Formally, we consider an algorithm $\mathcal{S}$ (CP solver) and a hyperparameters space Λ.

In general, HPO requires defining an objective (or loss) function L that quantifies the performance of the solver for a given set of hyperparameters $\lambda \in \Lambda$. This performance metric is typically empirical, calculated by running the solver $\mathcal{S}$ with configuration λ on a set of problem instances and measuring, for example, the mean runtime or the quality of the solution found [8]. This process results in a performance score, which is a real value.

The objective function for HPO is therefore a black-box function $L : \Lambda \to \mathbb{R}$ [21]. The goal of HPO is to find the configuration λ^* that minimizes this function:

$$\lambda^* = \arg\min_{\lambda \in \Lambda} L(\lambda)$$

It is crucial to distinguish between the COP objective function $f(a)$, which measures the quality of a single problem solution, and the HPO objective function $L(\lambda)$, which measures the performance of a solver configuration.

Example: Applying HPO to the Knapsack Problem

Continuing with the bounded knapsack example, a CP solver must decide which variable to branch on next (e.g., the item with the best value-to-weight ratio) and which value to try first (e.g., the maximum possible quantity). These choices are key hyperparameters that define the search strategy.

- **Hyperparameter space** Λ: The space of all possible search strategies. For instance, a single configuration $\lambda_1 \in \Lambda$ could be a "greedy" strategy:

$$\lambda_1 = \{\text{var\_select: } \texttt{'max\_value\_ratio'}, \text{ val\_select: } \texttt{'indomain\_max'}\}$$

  Another configuration, $\lambda_2 \in \Lambda$, representing a more conservative strategy, could be:

$$\lambda_2 = \{\text{var\_select: } \texttt{'first\_fail'}, \text{ val\_select: } \texttt{'indomain\_min'}\}$$

- **HPO objective function** $L(\lambda)$: To evaluate these strategies on a large knapsack instance, we define $L(\lambda)$ as the best objective value found by the solver within a fixed time limit when using the strategy defined by λ.
- **HPO goal**: The HPO process then automatically tests different configurations like λ_1, λ_2, and many others to find the one, λ^*, that finds the better total value for the knapsack within the given time.

2.2.1 Grid Search

Grid search is a simple yet exhaustive method that evaluates all possible combinations of hyperparameter values within predefined ranges. Although computationally expensive, it provides a useful baseline because it systematically explores the entire hyperparameter space [12].

Let Λ denote the set of hyperparameters space, expressed as the Cartesian product of the individual hyperparameter domains:

$$\Lambda = \mathcal{H}_1 \times \mathcal{H}_2 \times \cdots \times \mathcal{H}_k,$$

where each $\mathcal{H}_i$ represents the discrete set of candidate values for the i-th hyperparameter. A configuration $\lambda = (\lambda_1, \ldots, \lambda_k)$ corresponds to one choice of values, where $\lambda_i \in \mathcal{H}_i$. The total number of configurations is therefore:

$$N = \prod_{i=1}^{k} |\mathcal{H}_i|.$$

The objective function $L(\lambda)$ is evaluated for each configuration $\lambda \in \Lambda$, and the configuration that achieves the lowest value is selected as the best one:

$$\lambda^* = \arg\min_{\lambda \in \Lambda} L(\lambda).$$

While grid search guarantees complete coverage of the hyperparameter space, its computational cost grows exponentially with the number of hyperparameters [8]. For example, if a solver has three hyperparameters and each has ten possible values, the total number of evaluations required is $10^3 = 1{,}000$.

2.2.2 Iterative Search Methods: Random and Local Search

Moving beyond exhaustive methods like grid search, iterative strategies explore the hyperparameter space sequentially. Two fundamental and contrasting approaches are random search and local search, which represent the core principles of global exploration and local exploitation, respectively.

Random search is an uninformed method that focuses purely on exploration. It samples hyperparameters λ at random from the hyperparameter space Λ until a predefined budget is met. The key advantage is its effectiveness in high-dimensional spaces where only a few hyperparameters are critical. Unlike grid search, which systematically evaluates all combinations from predefined discrete grids, random search explores the space more broadly with fewer evaluations, increasing the likelihood of finding near-optimal configurations [13]. However, its major drawback is that it does not learn from past evaluations; every trial is independent, which can lead to inefficiently resampling of unpromising regions [22].

In direct contrast, local search is a method centered on exploitation. It begins with an initial configuration and iteratively moves to an adjacent or neighboring configuration only if it offers improved performance. The concept of a neighborhood is crucial, especially in spaces with categorical hyperparameters [23]. This raises the challenge of how to define a neighbor in a discrete space; the standard approach is to use a distance metric, with the Hamming distance being one of the most fundamental metrics. This is because it is particularly well-suited for categorical data, which has no inherent ordering, by simply counting the number of differing hyperparameter values between two configurations [23]. Its simplicity and efficient computation make it a valuable and straightforward method for defining a neighborhood in these discrete spaces.

2.2.3 Hamming Distance

The Hamming distance is a standard metric that measures dissimilarity by counting the positions at which two configuration vectors differ:

$$d_H(A, B) = \sum_{i=1}^{n} \mathbb{I}(A_i \neq B_i)$$

where $\mathbb{I}(\cdot)$ is the indicator function, which evaluates to 1 if its argument is true and 0 otherwise. The neighborhood of a configuration λ in Λ, denoted $\mathcal{N}(\lambda)$, is defined as:

$$\mathcal{N}(\lambda) = \{\lambda' \in \Lambda \mid d_H(\lambda, \lambda') = 1\}.$$

These methods exemplify the classic exploration–exploitation trade-off. Random search wastes evaluations in unpromising regions (inefficient exploitation), whereas pure local search can easily become trapped in local optima (poor exploration).

These challenges have led to hybrid optimization strategies. For instance, the iterated local search (ILS) metaheuristic [24] is a simple but powerful method. ILS works by repeatedly applying a local search to a solution, then changing it (perturbing)

to escape local optima. By switching between refining a solution locally and making random changes, ILS can effectively explore new and promising areas.

A good example of this is ParamILS, an algorithm that uses the ILS approach [25]. ParamILS searches locally for the best solution, and when it gets stuck, it makes a random change to start searching in a new area. While these iterative methods are better than simple techniques like grid search, they have limits. This shows we need smarter, model-based approaches that intelligently balance exploration and exploitation.

2.2.4 Bayesian Optimization

BO is a powerful, model-based HPO strategy designed to optimize expensive black-box functions. This makes it particularly well suited for tuning computationally costly CP solvers, where each evaluation of a set of hyperparameters can take a significant amount of time. Unlike uninformed methods such as grid search or random search, BO builds a probabilistic surrogate model to approximate the objective function.

The method builds a surrogate model $g(x)$ of the true objective function $L(x)$, and selects the next configuration x_{n+1} to evaluate by optimizing an acquisition function $a(x)$. The role of the acquisition function is to balance the trade-off between exploring uncertain regions of the hyperparameter space and exploiting regions known to have good performance [16]. The most common choice for this surrogate is a Gaussian process (GP) [15], which defines a prior over functions. After observing some data, this is updated to a posterior distribution that models our belief about the objective function's behavior, providing a mean prediction and an uncertainty estimate for any given configuration:

$$g(x) \sim \mathcal{GP}\big(\mu(x), k(x, x')\big)$$

where:

- $\mu(x)$ is the mean function, representing the expected value of $L(x)$.
- $k(x, x')$ is the covariance or kernel function, which measures the similarity between points x and x'. It models the correlation between the function values at those points.

This surrogate model is then used by an acquisition function to intelligently guide the search for the next configuration to evaluate [26, 27]. The role of the acquisition function is to balance the critical exploration (probing regions where the model is highly uncertain, which could potentially hide an even better, undiscovered optimum) and exploitation (focusing on regions that the surrogate model predicts will yield high performance) trade-off [27].

Standard acquisition functions like expected improvement quantify this potential and select the configuration that offers the best balance [16].

The BO process is iterative: after each new configuration is evaluated by running the CP solver, the result is used to update the surrogate model [28]. This allows the search to become progressively more informed, concentrating its evaluations in the most promising areas of the hyperparameter space. By building this explicit model, BO aims to find high-quality solutions with significantly fewer evaluations, a crucial advantage in the CP domain [28, 29].

3 Probe and Solve Algorithm

To systematically optimize a set of hyperparameters for CP solvers, we introduce PSA, a flexible and adaptive framework designed to make effective use of a limited time budget. The core idea of PSA is to partition a global time-limit (T_g) into two distinct phases:

1. A *probing phase*, where a dedicated portion of the time budget is used to explore a wide range of hyperparameters. This is achieved by running many short-lived, time-limited solver runs, each with a different configuration, to gather performance data.
2. A *final solving phase*, where the single best-performing configuration identified during probing is used to solve the problem instance with the entire remaining time budget.

This two-phase approach provides a structured balance between the exploration of the hyperparameter space and the exploitation of the most promising strategy found. The framework is designed to be highly modular, allowing different strategies for time management, hyperparameter selection, and timeout evolution to be composed, enabling a thorough investigation of their combined effects.

The PSA framework follows a two-phase architecture that partitions the global time budget T_g into probing time t_p and solving time t_s:

$$t_p = \rho \cdot T_g, \quad t_s = T_g - t_p, \quad \rho \in [0, 1]$$

With $\rho = 0.2$ as the default value based on empirical analysis, 20% of the total time is dedicated to exploring strategies, while 80% is reserved for actual solving with the best strategy found. This balanced allocation addresses the fundamental trade-off between exploration and exploitation in algorithm configuration [30].

Figure 1 illustrates the PSA architecture. The framework begins with a COP instance and global time budget, proceeds through the probing phase to identify the best configuration λ^*, and concludes with the solving phase using the remaining time.

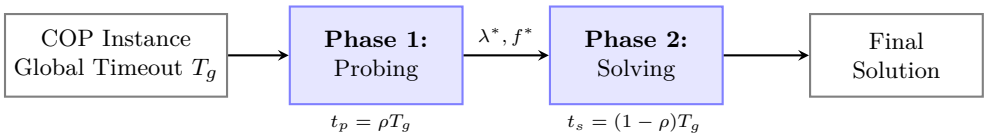


Fig. 1: PSA two-phase architecture with time allocation. The framework partitions the global time budget T_g into probing time t_p and solving time t_s.

3.1 Probing Phase

The goal of the probing phase is to efficiently sample the hyperparameter space to identify a high-quality configuration. The phase consists of a sequence of iterative

trials, hereafter referred to as *rounds*. In each round, a single set of hyperparameters is selected and evaluated by running the solver for a short duration. The execution of the probing phase is detailed in Algorithm 1 and is controlled by these key orthogonal strategy types.

3.1.1 HPO Method

This is the core HPO engine of the PSA, responsible for selecting the next set of hyperparameters to evaluate. To ensure modularity, the framework requires any HPO strategy to implement a specific interface with three key functions:

1. **`NextConfig()`:** Returns the next set of hyperparameters to be evaluated from the search space.
2. **`UpdateModel(params, runtime)`:** Takes the parameters of the last trial and its resulting runtime and objective to update the strategy's internal model. For model-free methods like Hamming distance, this may simply involve updating the incumbent.
3. **`AllocateTime`(T_g):** Partitions the global timeout into probing and solving budgets:

$$\texttt{AllocateTime}(T_g) = (\rho T_g,\ (1-\rho)T_g)$$

   where ρ is the probing ratio (default $\rho = 0.2$). This creates a clear separation: t_p for configuration exploration, t_s for solving with the best configuration.
4. **`GetBestConfig()`:** Returns the best configuration found so far, which is used in the final solving phase.

We implement BO as a core component of PSA. It is a concrete strategy that conforms to this interface. BO is a sophisticated model-based strategy where `NextConfig` uses an acquisition function over a GP model to choose the next point. `UpdateModel` retrains the GP with the new data point.

3.1.2 Global Time Management

This high-level component determines the total time budget allocated to the probing phase. We implement two approaches:

- **Static percentage allocation:** A fixed percentage of T_g is reserved for the entire probing phase. For example, with a T_g of 1800 seconds and a 20% allocation, the probing phase will run for a total of 360 seconds before automatically transitioning to the solving phase. This ensures a predictable and consistent time split across different experiments.
- **Iteration-limited allocation:** The probing phase continues until either the T_g is exhausted or a predefined maximum number of configuration trials (`max_tries`) has been completed. This strategy allows for more flexible exploration, as the total probing time is determined by the cumulative runtime of the trials, making it suitable for scenarios where individual probe runtimes are highly variable.

Algorithm 1 Probing Phase of PSA

Input: COP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$; HPO $\mathcal{H}$; Λ; T_g; τ_0; *TimeInit*; *Evolve*; *stop_type*; L.
Output: $\lambda^\star$, $f^\star$, t_{rem}.

$t_p, t_s \leftarrow \text{AllocateTime}(T_g)$ ▷ Probe/solve budgets
Initialize: $f^\star \leftarrow \infty$, $\lambda^\star \leftarrow$ default, $stag \leftarrow 0$, $t_0 \leftarrow \text{Now}()$

if *TimeInit* = FirstRuntime **then** ▷ Adaptive timeout
    $(sol, rt) \leftarrow \text{Solve}(\text{COP}, \text{default}, t_p)$
    $\tau \leftarrow rt$ ▷ Use runtime
    if *sol* **then** $f^\star \leftarrow f(sol)$; $\lambda^\star \leftarrow$ default
    end if
else ▷ Static timeout
    $\tau \leftarrow \tau_0$
end if

while $\text{Now}() - t_0 < t_p$ **do** ▷ Main loop
    $\lambda \leftarrow \mathcal{H}.\text{NextConfig}()$
    if $\lambda = \varnothing$ **then break**
    end if

    if $\lambda \in \mathcal{M}$ **then** ▷ Check memory array
        $(obj, rt) \leftarrow \mathcal{M}[\lambda]$ ▷ Retrieve cached result
    else
        $(sol, rt) \leftarrow \text{Solve}(\text{COP}, \lambda, \tau)$
        $obj \leftarrow f(sol)$ if *sol* else ∞
        $\mathcal{M}[\lambda] \leftarrow (obj, rt)$ ▷ Store in memory array
    end if

    if $obj < f^\star$ **then** ▷ Better solution found
        $f^\star \leftarrow obj$; $\lambda^\star \leftarrow \lambda$; $stag \leftarrow 0$
    else
        $stag \leftarrow stag + 1$
    end if

    $\mathcal{H}.\text{UpdateModel}(\lambda, obj, rt)$
    $\tau \leftarrow Evolve(\tau)$ ▷ Update timeout

    if *stop_type* = FirstSolution and $obj \neq \infty$ **then break**
    end if
    if *stop_type* = Stagnation and $stag \geq L$ **then break**
    end if
end while

$t_{\text{rem}} \leftarrow T_g - (\text{Now}() - t_0)$
return $(\lambda^\star, f^\star, t_{\text{rem}})$

3.1.3 Round Timeout Initialization

This component determines the timeout for each individual configuration evaluation within the probing phase. This sets the time limit for the first configuration to be tested.

- **Static initial timeout:** A fixed, small timeout (e.g., 5 seconds) is used. This provides a consistent baseline but may be too short or too long for certain problem instances.
- **First-runtime timeout:** The framework first performs a preliminary run of the solver using its default configuration. This run is allocated an informed time limit which is up to the total available probe budget, and will stop as soon as the first solution is found. The actual runtime observed from this initial run is then used as the timeout for subsequent probes, thereby adapting the timeout to the intrinsic difficulty of the problem instance.

3.1.4 Timeout Evolution Pattern

After the initial round, the timeout for subsequent rounds can evolve to adapt to the performance of previously tested configurations. Our framework manages this through a modular *Timeout Evolution Strategy*, which implements an `Evolve(current_timeout)` function to compute the timeout for the next round based on the value from the current one.

We implement and test three concrete instances of this strategy:

- **Static evolution:** This strategy's `Evolve` function simply returns the same `current_timeout` it was given, keeping the timeout fixed.
- **Geometric evolution:** Here, the $\texttt{Evolve}(t_i)$ function returns the current timeout multiplied by a predefined growth factor, $\beta > 1$. For example, $t_{i+1} = \texttt{Evolve}(t_i) = t_i \times \beta$.
- **Luby sequence evolution:** This strategy's $\texttt{Evolve}(t_i)$ function maintains an internal counter k. It returns $t_{i+1} = t_0 \cdot \text{Luby}(k)$, where t_0 is the initial timeout and $\text{Luby}(k)$ is the k-th value in the sequence (1, 1, 2, 1, 1, 2, 4, ...), incrementing k after each call.

3.1.5 Probing Stop Conditions

The probing phase is allocated a specific time budget (e.g., a percentage of the global time limit or up to a maximum number of tries). The phase will terminate when this budget is exhausted, but it can also be configured to stop early to pivot to the final solving phase more quickly. The defined stopping conditions are:

- **Timeout**: The primary stopping condition is the exhaustion of the allocated probing time budget. The phase will always terminate when its time is up, ensuring that the final solving phase receives its planned portion of the global time limit.
- **First solution found:** The probing phase can be configured to halt as soon as any configuration finds the first feasible solution. This is particularly useful in scenarios where quickly finding any valid solution is more critical than extensive exploration for the absolute best configuration.

- **Stagnation:** The process can terminate if the best-found solution has not improved after a predefined number of rounds. This prevents the framework from wasting time on continued exploration if the HPO strategy appears to have converged or is no longer making progress.

3.1.6 General Improvements

To enhance the efficiency and scalability of our approach, we incorporate a memory-based mechanism:

- **Memory array:** A data structure is implemented to store previously evaluated configurations and their corresponding results. When a configuration is revisited, the solver retrieves the stored results instead of performing redundant evaluations. This not only accelerates the optimization process, but also avoids redundant computation, particularly for large-scale problem instances.

3.2 Solving Phase

Once the probing phase concludes, PSA transitions to the solving phase, which utilizes the best-found configuration to solve the instance with the remaining time budget.

Algorithm 2 Solving Phase of PSA

Input: COP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$; HPO $\mathcal{H}$; $\lambda^\star$; $f^\star$; t_{rem}.
Output: sol_{final}, $f(sol_{\text{final}})$.

if $f^\star = \infty$ **then** ▷ No solution found in probing
    $\lambda_{\text{final}} \leftarrow \mathcal{H}$.GetBestConfig()
    $(sol_{\text{final}}, rt) \leftarrow \text{Solve}(\text{COP}, \lambda_{\text{final}}, t_{\text{rem}})$
else ▷ Incumbent exists; try to improve
    $\mathcal{C}' \leftarrow \mathcal{C} \cup \{f(\mathcal{X}) < f^\star\}$ ▷ Add cut constraint
    $(sol_{\text{final}}, rt) \leftarrow \text{Solve}((\mathcal{X}, \mathcal{D}, \mathcal{C}', f), \lambda^\star, t_{\text{rem}})$
end if

return $(sol_{\text{final}}, f(sol_{\text{final}}))$

The solving phase operates as follows:

1. The single best-performing set of hyperparameters ($\lambda^\star$) identified during the probing phase is retrieved.
2. If any solution was found during probing (i.e., $f^\star < \infty$), a new objective cut constraint is added to the model: $\mathcal{C}' \leftarrow \mathcal{C} \cup \{f(\mathcal{X}) < f^\star\}$. This constraint forces the solver to search only for solutions that are strictly better than the current best, implementing an optimization loop that focuses the search on improvement rather than re-finding known solutions.
3. The solver is executed with $\lambda^\star$ and the (potentially updated) constraint model $\mathcal{C}'$, using the entire remaining time budget (T_g − probing elapsed time).

3.3 Complete PSA Algorithm Specification

The integrated PSA algorithm, presented in Algorithm 3, combines the probing and solving phases into a unified framework. This algorithm operationalizes the two-phase tuning strategy by integrating the modular components for HPO and time management described previously.

Algorithm 3 Complete PSA Framework

Input: COP instance $(\mathcal{X}, \mathcal{D}, \mathcal{C}, f)$; global time limit T_g; probing fraction ρ; HPO method $\mathcal{H}$; hyperparameter space Λ.
Output: Final solution sol_{final} and its objective value $f(sol_{final})$.

$t_p \leftarrow \rho \times T_g$ ▷ Calculate probing budget
$t_s \leftarrow T_g - t_p$ ▷ Calculate solving budget

Phase 1: Probing
$(\lambda^\star, f^\star) \leftarrow \text{ProbingPhase}((\mathcal{X}, \mathcal{D}, \mathcal{C}, f), \Lambda, t_p, \mathcal{H})$ ▷ Execute Algorithm 1

Phase 2: Solving
$(sol_{final}, f_{final}) \leftarrow \text{SolvingPhase}((\mathcal{X}, \mathcal{D}, \mathcal{C}, f), \lambda^\star, f^\star, t_s)$ ▷ Execute Algorithm 2

return (sol_{final}, f_{final})

This integrated approach provides several key properties:

- **Time-awareness:** The framework strictly respects the global timeout T_g, ensuring practical applicability.
- **Adaptability:** The adaptive timeout mechanism in the probing phase adjusts to problem difficulty.
- **Robustness:** Fallback mechanisms ensure a solution is returned even if individual phases fail.
- **Modularity:** Components (HPO methods, time allocation strategies) can be easily swapped.

4 Experimental Setup

This section details the environment, problem instances, and the specific configurations employed for our large-scale benchmarking study. We begin by describing the two constraint solvers used as underlying engines, followed by an outline of the problem instances and the comprehensive benchmarking methodology.

To robustly evaluate the PSA framework, we selected two distinct constraint solvers: ACE and Choco. Their varying complexities and feature sets provide a broad and representative testbed for our tuning approach.

Table 1: The Tunable Hyperparameter Space of the ACE Solver.

Parameter	Description	# Options
`varh`	Variable Selection Heuristic	10
`valh`	Value Selection Heuristic	10
`saf`	Stay Array Focus	2
`pc1`	Preserve Unary Constraints	2
`toh`	Convert to Hybrid Tables	2
`lc`	Last Conflict Weighting	4
`negative`	Algorithm for Negative Table Constraints	3
`sc2`	Structure for Binary Table Constraints	4
`sc3`	Structure for Ternary Table Constraints	4
Total Combinatorial Configurations		**153,600**

4.1 The ACE Solver

ACE is a modern constraint solver, implemented in Java, designed to tackle combinatorial problems involving integer and Boolean variables. Its extensive capabilities comply with the XCSP3-core standard, a widely recognized format for constraint problems [31]. ACE offers a rich library of constraints, including advanced table constraints (ordinary, starred, and hybrid), and a wide array of global constraints such as `AllDifferent`, `Cardinality` [32], `Count`, `Element`, and `Cumulative` [1, 17, 33]. Additionally, it features various built-in search heuristics and is optimized for mono-criterion optimization. This comprehensive feature set is particularly valuable for PSA, as it contributes to a vast, complex, and highly influential hyperparameter space, making it an ideal candidate for showcasing the benefits of automated tuning.

The chosen hyperparameter space for ACE, summarized in Table 1, consists of 9 distinct dimensions that govern the solver's core search behavior. These include heuristics for variable and value selection (`varh`, `valh`), the information of table constraints (`sc2`, `sc3`), and other categorical parameters that guide branching, learning, and constraint handling. The combination of these chosen options results in a combinatorial search space of over 150 thousand unique configurations. This scale makes exhaustive tuning methods like grid search computationally infeasible and underscores the significant challenge of manual tuning. Consequently, ACE serves as an excellent testbed for demonstrating PSA's ability to effectively navigate such a hyperparameter space.

A practical contribution of this work is the integration of the previously unsupported ACE solver into the CPMpy library [34]. Within this newly integrated environment, we apply our HPO strategies to tune ACE across various problem instances. The framework's success is assessed based on both the quality of the final objective value obtained and the total time required to find that solution. Thus, utilizing ACE not only provides a challenging and relevant testbed for our tuning framework but also enhances the capabilities of the broader CPMpy ecosystem for future research.

Table 2: The Tunable Hyperparameter Space of the Choco Solver.

Parameter	Description	# Options
Lc	Level of Consistency (Level of propagations)	2
Restarts	Restart policy (e.g., LUBY, GEOMETRIC, NONE)	12
Valh	Value Selection Heuristic	6
Varh	Variable Selection Heuristic	19
Flush	Propagation queue flush threshold	5
Total Combinatorial Configurations		**136,800**

4.2 The Choco Solver

We also utilize the Choco solver (v4.10.6), a widely recognized open-source CP library implemented in Java. Choco provides a rich collection of heuristics and propagation levels, making it a strong alternative for benchmarking the PSA framework [35].

The hyperparameter space for Choco, which was derived from a JSON file detailing its available hyperparameters, is presented in Table 2. This space includes key components such as restart policies (`restarts`), variable and value selection heuristics (`varh`, `valh`), consistency levels (`lc`), and propagation queue flushing thresholds (`flush`).

4.3 Benchmarking Methodology

Our benchmarking approach involved four distinct types of experimental runs for both the ACE and Choco solvers:

Default Solver Baselines

For each of the 114 instances, we first solve the instances with the default version of both ACE and Choco solvers. These runs served as a foundational baseline, reflecting the out-of-the-box performance of each solver without any hyperparameter tuning. Each default run adhered to the same system environment and the 1800-second global time limit.

PSA Framework Configurations

The core of our extensive benchmarking involves the PSA framework, which employs two distinct HPO methods: BO and Hamming distance search [36]. Both methods are implemented within the PSA framework, allowing a direct comparison of their effectiveness. We conduct a full-factorial experiment to systematically evaluate the impact of each modular component of the PSA framework. This design allows us to not only identify the best overall configuration but also to analyze the independent contribution of each strategic dimension to performance. A total of 24 unique PSA configurations for each HPO method are generated by combining all possibilities from the following strategic dimensions:

- **Probing Time Allocation:** 2 options (*Static*, *Dynamic*)
- **Round Timeout Initialization:** 2 options (*Static*, *First-Runtime*)
- **Timeout Evolution Pattern:** 3 options (*Static*, *Geometric*, *Luby*)

- **Probing Stop Condition:** 3 options (*First Solution*, *Timeout*, *Stagnation*)

Some combinations of these settings are inherently incompatible. In particular, when the probing stop condition is set to *First Solution*, the probing phase terminates as soon as the first solution is found. In this case, the probing loop executes only a single round, which makes any form of round-timeout evolution irrelevant. For this reason, whenever the stop condition is *First Solution*, only the *static* timeout evolution pattern is meaningful and permitted. Accordingly, these 24 configurations are derived from two batches:

- Round-timeout static: $2\times3\times3 = 18$ configurations
- Round-timeout First-Runtime: $2\times1\times3 = 6$ configurations

These 24 PSA configurations for each HPO method are evaluated per instance for both the ACE and Choco solvers.

Champion Configuration Evaluation

Following the exploratory phase of the PSA framework, the single best-performing configuration (the *champion configuration*) for each combination of solver and HPO method is identified based on the aggregated results. Subsequently, these identified champion configurations are then run separately on all 114 instances. These dedicated runs provide a direct and robust measure of their performance for comparative analysis against the baselines.

4.4 Execution Environment and Data Collection

To ensure fairness and reproducibility, all experiments were conducted under strictly controlled conditions.

- **Global time limit:** Each individual experimental run is allocated a maximum wall-clock time of 1800 seconds (30 minutes), aligning with the standard duration used in the XCSP3 competition from which our benchmarks are drawn.
- **Computational resources:** The full suite of experiments was executed on the high-performance computing (HPC) facilities, with the technical specifications of a cluster compute node being 2xAMD Epyc ROME 7H12 @ 2.6 GHz [64c/280W] processor with 256 GB RAM.
- **Total computational effort:** The complete experiment, encompassing 6156 individual runs (114 instances $\times$ (24 PSA configurations $\times$ 2 HPO methods + 1 default solver baseline configuration + 1 PSA champion for each HPO method) $\times$ 2 solvers), represents a total of 3078 compute-hours.
- **Data logging and reproducibility:** For each run, detailed performance data is logged to structured CSV files:
  - **Objective value:** The best objective value found by the solver within the time limit.
  - **Solver status:** The final exit status reported by the solver (e.g., `OPTIMUM FOUND`, `SATISFIABLE`, `TIMEOUT`, `ERROR`).
  - **Runtime:** The total wall-clock time elapsed for the run.

Table 3: PSA Component Strategy Frequencies for Choco Solver

Component	Strategy	BO Wins	Hamming Wins
Global Time	Percent	53.40%	42.98%
	Dynamic	46.60%	57.02%
Round Timeout	Static	76.70%	81.58%
	First-Runtime	23.30%	18.42%
Timeout Evolution	DynamicGeometric	31.07%	28.07%
	DynamicLuby	31.07%	20.18%
	Static	37.86%	51.75%
Stop Condition	Timeout	36.89%	38.6%
	Stagnation	35.92%	27.19%
	FirstSolution	27.18%	34.21%

  - **Configuration details:** All command-line flags used for each run are logged to ensure full reproducibility and allow detailed performance analysis.

- **Parameter Settings for PSA Components:** For strategies utilizing the *Static Percentage Allocation*, the probing budget was set to 20% of T_g, a choice that tries to balance exploration with a substantial budget for the final solve. The *Static* initial round timeout was set to a baseline of *5* seconds. For the *Geometric* evolution pattern, a growth factor of $\beta = 1.5$ is used. These values are chosen as reasonable, standard defaults to avoid overfitting the framework to this specific benchmark set.

5 Results and Analysis

This section presents the comprehensive results of our large-scale benchmarking experiments. Our primary objective is to identify the most effective automated strategies for configuring a solver's hyperparameters within the PSA framework and to derive key principles for efficiently utilizing a limited time budget. We organize our findings into three main stages: for each solver, starting with Choco and then proceeding to ACE, first, we provide an aggregate analysis of the performance of PSA's internal component strategies, examining general trends across all problem instances. Second, based on these aggregate insights, we synthesize and present the champion configurations. Third, we then conduct detailed solver-specific performance comparisons, evaluating how these identified champion configurations perform against baseline configurations. For clarity and focus, the numerical results and pie charts presented in this section derive from a single, representative random seed.

We present the aggregated results in four comprehensive tables. Table 3 shows the frequency of winning PSA component strategies for the Choco solver with both BO and Hamming HPO methods. Table 4 provides the equivalent analysis for ACE. Based on these frequencies, we derive champion configurations for each solver-HPO combination in Table 5. Finally, Table 6 summarizes the performance comparisons between PSA-tuned and default configurations.

Table 4: PSA Component Strategy Frequencies for ACE Solver

Component	Strategy	BO Wins	Hamming Wins
Global Time	Percent	50.44%	48.25%
	Dynamic	49.56%	51.75%
Round Timeout	Static	77.18%	86.84%
	First-Runtime	22.82%	13.16%
Timeout Evolution	DynamicGeometric	48.73%	27.19%
	DynamicLuby	25.17%	49.12%
	Static	26.10%	23.68%
Stop Condition	Timeout	35.59%	29.82%
	Stagnation	33.94%	35.96%
	FirstSolution	30.46%	34.21%

Table 5: Champion Configurations for Both Solvers

Component	Choco Champion (BO / Hamming)	ACE Champion (BO / Hamming)
Global Time Management	Percent / Dynamic	Percent / Dynamic
Round Timeout Initialization	Static / Static	Static / Static
Timeout Evolution Pattern	DynamicGeometric / Static	DynamicGeometric / DynamicLuby
Probing Stop Condition	Timeout / Timeout	Timeout / Stagnation

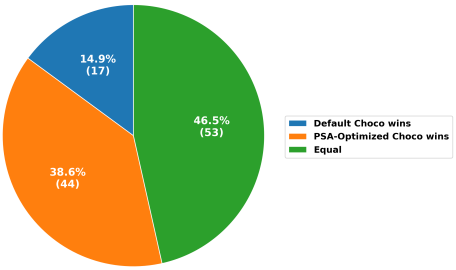
5.1 Choco Solver Results

Analysis of PSA Component Strategies for Choco

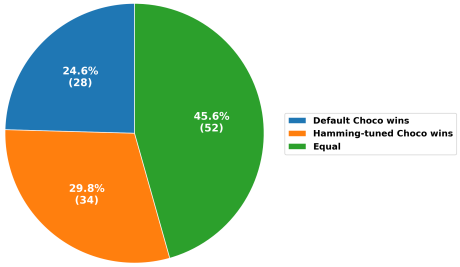
Table 3 reveals distinct patterns in effective PSA strategies for the Choco solver. For global time management, BO favors the *percent* strategy (53.40%), while Hamming prefers *dynamic* allocation (57.02%). Both HPO methods strongly favor *static* round timeout initialization (76.70% for BO, 81.58% for Hamming). Timeout evolution patterns differ significantly: with BO, all three strategies are nearly equally effective (37.86% static, 31.07% each for dynamic variants), whereas with Hamming, *static* evolution dominates (51.75%). For stop conditions, *timeout* is most frequent with both methods, though Hamming shows stronger preference for *firstSolution* (34.21%) compared to BO (27.18%).

Synthesizing Champion Configurations for Choco

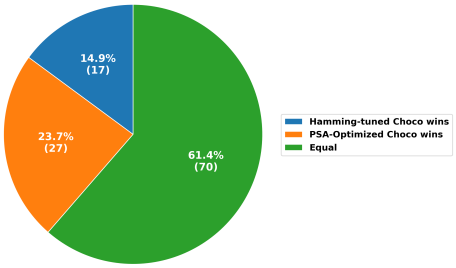
Based on the aggregated analysis in Table 3, we derive two champion configurations for Choco, one optimized for BO and another for Hamming. These configurations balance component effectiveness with overall strategy coherence and are summarized in Table 5. For BO, the champion combines *percent* global time, *static* timeout initialization, *dynamicGeometric* evolution, and *timeout* stop condition. For Hamming, the configuration uses *dynamic* global time, *static* timeout initialization, *static* evolution, and *timeout* stop condition.



(a) PSA-BO Choco vs. Default Choco



(b) PSA-Hamming Choco vs. Default Choco



(c) PSA-BO Choco vs. PSA-Hamming Choco

Fig. 2: Pairwise Performance Comparison for Choco Solver Approaches. (a) *PSA-BO Choco* versus *default Choco*. (b) *PSA-Hamming Choco* versus *default Choco*. (c) *PSA-BO Choco* versus *PSA-Hamming Choco*.

Choco Performance Comparisons

To evaluate PSA's effectiveness for Choco, we compare three approaches: default settings (*default Choco*), PSA with BO (*PSA-BO Choco*), and PSA with Hamming (*PSA-Hamming Choco*). Results are summarized in Table 6 and visualized in Figure 2.

PSA-BO Choco vs. Default Choco: PSA-BO Choco performs better on 38.6% of instances, while default Choco is superior on only 14.9%, with ties on 46.5% (Figure 2a). This demonstrates BO's advantage over default settings.

PSA-Hamming Choco vs. Default Choco: Default Choco outperforms PSA-Hamming on 24.6% of instances, while PSA-Hamming wins on 29.8%, with ties on 45.6% (Figure 2b). This indicates PSA Hamming-based tuning generally outperforms Choco's default configurations.

PSA-BO Choco vs. PSA-Hamming Choco: PSA-BO clearly outperforms PSA-Hamming, winning on 23.7% of instances versus only 14.9% for Hamming, with ties on 61.4% (Figure 2c). This validates BO's superiority over the simpler Hamming approach.

Table 6: Performance Comparison Summary (Percentage of Instances)

Comparison	Winner A	Tie	Winner B
Choco Results			
PSA-BO Choco vs Default	38.6%	46.5%	14.9%
PSA-Hamming Choco vs Default	29.8%	45.6%	24.6%
PSA-BO vs PSA-Hamming Choco	23.7%	61.4%	14.9%
ACE Results			
PSA-BO ACE vs Default	25.4%	57.9%	16.7%
PSA-Hamming ACE vs Default	13.2%	68.4%	18.4%
PSA-BO vs PSA-Hamming ACE	24.6%	64.9%	10.5%

5.2 ACE Solver Results

Analysis of PSA Component Strategies for ACE

Table 4 shows PSA strategy effectiveness for ACE. Global time management shows minimal preference: BO slightly favors *percent* (50.44%), Hamming slightly favors *dynamic* (51.75%). Both strongly prefer *static* timeout initialization (77.18% BO, 86.84% Hamming). Timeout evolution differs dramatically: BO strongly prefers *dynamicGeometric* (48.73%), while Hamming favors *dynamicLuby* (49.12%). Stop conditions show balanced distributions, with *timeout* most frequent for BO (35.59%) and *stagnation* for Hamming (35.96%).

Synthesizing Champion Configurations for ACE

Based on Table 4, we derive champion configurations for ACE (Table 5). For BO: *percent* global time, *static* timeout initialization, *dynamicGeometric* evolution, and *timeout* stop condition. For Hamming: *dynamic* global time, *static* timeout initialization, *dynamicLuby* evolution, and *stagnation* stop condition.

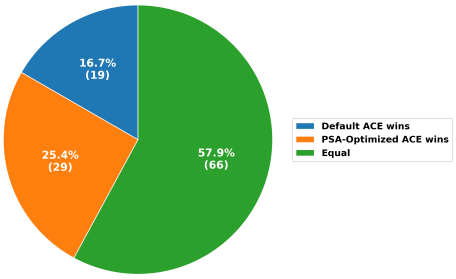
Performance Comparison of ACE Configurations

We compare default ACE with PSA-BO ACE and PSA-Hamming ACE. Results are in Table 6 and Figure 3.

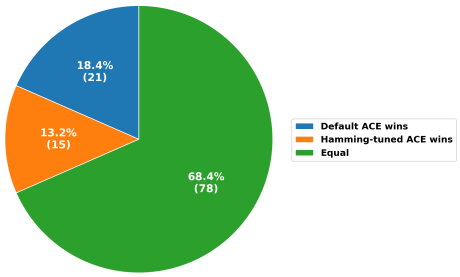
PSA-BO ACE vs. Default ACE: PSA-BO ACE wins on 25.4% of instances versus 16.7% for default, with ties on 57.9% (Figure 3a). BO provides clear advantage over default settings.

PSA-Hamming ACE vs. Default ACE: Default ACE outperforms PSA-Hamming on 18.4% of instances, while PSA-Hamming wins on 13.2%, with ties on 68.4% (Figure 3b). Hamming-based tuning generally underperforms defaults.

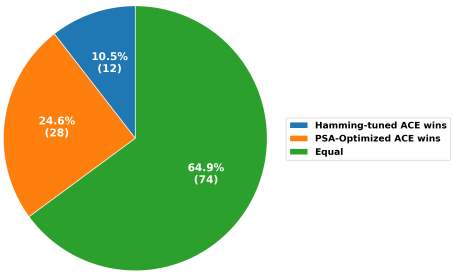
PSA-BO ACE vs. PSA-Hamming ACE: PSA-BO dominates, winning on 24.6% of instances versus 10.5% for Hamming, with ties on 64.9% (Figure 3c). BO's model-based approach proves significantly more effective.



(a) PSA-BO ACE vs. Default ACE



(b) PSA-Hamming ACE vs. Default ACE



(c) PSA-BO ACE vs. PSA-Hamming ACE

Fig. 3: Pairwise Performance Comparison for ACE Solver Approaches. (a) *PSA-BO ACE* versus *default ACE*. (b) *PSA-Hamming ACE* versus *default ACE*. (c) *PSA-BO ACE* versus *PSA-Hamming ACE*.

6 Discussion and Future Research

Our experiments provide strong evidence that PSA is an effective strategy for optimizing constraint solvers under a fixed time budget. By utilizing BO to intelligently manage computational resources, PSA consistently identifies configurations that outperform baseline methods. This section interprets these findings, discusses their practical implications for users, and outlines promising avenues for future research.

The empirical results highlight several key trends. First, PSA demonstrates a clear advantage over the Hamming distance baseline. This validates the use of a model-based approach; the computational overhead of building a surrogate model in BO is justified by its ability to navigate complex search spaces far more efficiently than simple local search.

Furthermore, our analysis of the most successful configurations reveals that they typically combine two synergistic elements:

1. **Static Initialization:** A short, fixed time limit for the initial run acts as a gatekeeper, rapidly filtering out poor configurations without wasting significant time.

2. **Adaptive Exploration:** Following the initial filter, a dynamic evolution strategy increases the timeout. This ensures that if a configuration shows promise, the solver is granted sufficient time to deepen the search.

This combination effectively balances the exploration-exploitation trade-off, minimizing risk while maximizing the probability of finding a high-quality solution. Consequently, the champion configurations we identified serve as robust generalists. While they may not be the theoretical optimum for every single instance, they provide a reliable, high-performance baseline across a wide variety of problem types. This robustness is particularly valuable for users who cannot afford the computational cost of tuning a solver separately for every new problem encounter.

These findings translate into three practical benefits for the CP community:

- **Democratization of Solving:** PSA automates the complex task of hyperparameter tuning. This pushes CP closer to the ideal model and solve paradigm, allowing users to focus on defining their problems rather than mastering the intricacies of solver configuration.
- **Standardized Benchmarking:** The framework offers a principled methodology for comparing solvers under fixed time budgets, providing a scientific alternative to ad-hoc manual tuning.
- **Solver Agnosticism:** The modular nature of PSA allows it to be easily applied to other solvers or algorithms, as demonstrated by its successful deployment on both ACE and Choco.

Despite these successes, there remain opportunities for further development. First, while our study focused on ACE and Choco, future work should validate the generalizability of PSA on other major solvers, such as OR-Tools. Second, the current implementation uses fixed meta-parameters (e.g., a 20% probing budget). A promising direction is the development of a *self-tuning* system that automatically adapts these parameters based on the problem size or complexity. Finally, rather than relying on a single static champion, future iterations could dynamically select a strategy class—switching between *static* and *dynamic* timeout evolution—based on the specific characteristics of the input problem.

7 Conclusion

This paper introduced PSA, a resource-aware, two-phase framework for automated HPO of CP solvers. PSA allocates a portion of the overall time budget to a probing phase, where BO guides a fast exploration of the configuration space, and uses the remaining time to solve the problem with the most promising configuration. The framework is fully integrated into CPMpy and includes the first integration of the ACE solver into the library.

Our evaluation across 114 benchmark instances shows that PSA provides consistent benefits for two solvers with fundamentally different architectures. For ACE, PSA with BO provided clear benefits over the default configuration: it achieved better results in 25.4% of the instances (29 cases), while the default configuration was better in only 16.7% (19 cases); both performed equally on 57.9% (66 cases). For Choco,

PSA with BO improved performance in 38.6% of instances and matched the baseline in 46.5%, while the default configuration was better in only 14.9% of cases.

Across both solvers, PSA with BO consistently outperformed the Hamming distance baseline. For ACE, PSA-BO won 24.6% of comparisons against PSA-Hamming, while PSA-Hamming won only 10.5%. For Choco, PSA-BO won 23.7% of comparisons against PSA-Hamming, while PSA-Hamming won 14.9%. These results demonstrate the advantage of model-guided BO over simpler local-perturbation strategies.

Beyond numerical improvements, the results highlight an important insight: the effectiveness of a tuning strategy depends strongly on the type of problem. Harder optimization problems benefit from dynamic timeout evolution, which allows PSA to progressively allocate more time to promising configurations, while easier satisfaction-oriented instances benefit from evaluating many configurations quickly under static time budgets. This indicates that PSA not only improves solver performance but also exposes useful structure in how hyperparameters interact with different classes of problems.

Overall, PSA contributes a practical and automated approach for improving the performance of constraint solvers without requiring expert knowledge or manual parameter tuning. Future work will focus on making PSA more adaptive by incorporating instance features, supporting multi-objective configuration, and extending the framework toward general automated solver design.

Acknowledgment

Funding

This work is partially funded by the joint research program UL/SnT—ILNAS on Technical Standardization for Trustworthy ICT, Aerospace, and Construction, and supported by the Luxembourg National Research Fund (FNR)—COMOC Project, ref. C21/IS/16101289.

Data Availability

The benchmark instances used in this study are publicly available as part of the XCSP24 benchmark suite at: https://www.cril.univ-artois.fr/XCSP24/.

Code Availability

The implementation of the Probe and Solve Algorithm (PSA) integrated into CPMpy is available at: https://github.com/Hedieh-Haddad/cpmpy/tree/psa.

References

[1] Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems. Springer, Amsterdam (2001)

[2] Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming [Book]. ISBN: 9780080463803 (2006). https://www.oreilly.com/library/view/handbook-of-constraint/9780444527264/ Accessed 2024-06-18

[3] Bessiere, C.: Chapter 3 - Constraint Propagation. In: Rossi, F., Beek, P., Walsh, T. (eds.) Foundations of Artificial Intelligence. Handbook of Constraint Programming, vol. 2, pp. 29–83. Elsevier, Amsterdam (2006). https://doi.org/10.1016/S1574-6526(06)80007-6 . https://www.sciencedirect.com/science/article/pii/S1574652606800076 Accessed 2025-10-15

[4] Pesant, G., Gendreau, M.: View of local search in constraint programming. In: Principles and Practice of Constraint Programming - CP'96, vol. 1118, pp. 353–366. Springer, Cambridge, US (1996). https://doi.org/10.1007/3-540-61551-2 . https://doi.org/10.1007/3-540-61551-2 Accessed 2025-10-15

[5] Marty, T., Boisvert, L., François, T., Tessier, P., Gautier, L., Rousseau, L.-M., Cappart, Q.: Learning and fine-tuning a generic value-selection heuristic inside a constraint programming solver. Constraints (2024) https://doi.org/10.1007/s10601-024-09377-4 . Accessed 2024-12-13

[6] Aglin, G., Nijssen, S., Schaus, P.: Learning Optimal Decision Trees Under Memory Constraints. In: Amini, M.-R., Canu, S., Fischer, A., Guns, T., Kralj Novak, P., Tsoumakas, G. (eds.) Machine Learning and Knowledge Discovery in Databases vol. 13717, pp. 393–409. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-26419-1 . Series Title: Lecture Notes in Computer Science. https://link.springer.com/10.1007/978-3-031-26419-1 Accessed 2026-01-04

[7] O'Sullivan, B.: Opportunities and Challenges for Constraint Programming. Proceedings of the AAAI Conference on Artificial Intelligence **26**(1), 2148–2152 (2021) https://doi.org/10.1609/aaai.v26i1.8450 . Accessed 2026-01-04

[8] Bischl, B., Binder, M., Lang, M., Pielok, T., Richter, J., Coors, S., Thomas, J., Ullmann, T., Becker, M., Boulesteix, A.-L., Deng, D., Lindauer, M.: Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges. arXiv. arXiv:2107.05847 [stat] (2021). https://doi.org/10.48550/arXiv.2107.05847 . http://arxiv.org/abs/2107.05847 Accessed 2025-05-20

[9] Wu, J., Chen, X.-Y., Zhang, H., Xiong, L.-D., Lei, H., Deng, S.-H.: Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization. Journal of Electronic Science and Technology **17**(1), 26–40 (2019) https://doi.org/10.11989/JEST.1674-862X.80904120 . Accessed 2025-04-16

[10] Hutter, F., Kotthoff, L., Vanschoren, J. (eds.): Automated Machine Learning: Methods, Systems, Challenges. The Springer Series on Challenges in Machine Learning. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05318-5 . http://link.springer.com/10.1007/978-3-030-05318-5 Accessed 2025-10-15

[11] Lecoutre, C.: ACE, a generic constraint solver. arXiv. arXiv:2302.05405 [cs] (2024). https://doi.org/10.48550/arXiv.2302.05405 . http://arxiv.org/abs/2302.05405 Accessed 2025-04-16

[12] Liashchynskyi, P., Liashchynskyi, P.: Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS. arXiv. arXiv:1912.06059 [cs, stat] (2019). https://doi.org/10.48550/arXiv.1912.06059 . http://arxiv.org/abs/1912.06059 Accessed 2024-01-13

[13] Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. J. Mach. Learn. Res. **13**(null), 281–305 (2012)

[14] Ungredda, J., Branke, J.: Bayesian Optimisation for Constrained Problems. arXiv. arXiv:2105.13245 [cs, stat] (2021). https://doi.org/10.48550/arXiv.2105.13245 . http://arxiv.org/abs/2105.13245 Accessed 2024-07-04

[15] Rasmussen, C.E.: Gaussian Processes in Machine Learning. In: Bousquet, O., Luxburg, U., Rätsch, G. (eds.) Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures, pp. 63–71. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28650-9_4 . https://doi.org/10.1007/978-3-540-28650-9 Accessed 2025-05-20

[16] Candelieri, A.: Mastering the exploration-exploitation trade-off in Bayesian Optimization. arXiv. arXiv:2305.08624 [cs, math] (2023). https://doi.org/10.48550/arXiv.2305.08624 . http://arxiv.org/abs/2305.08624 Accessed 2024-06-27

[17] Lecoutre, C.: Constraint Networks: Techniques and Algorithms. Wiley, Hoboken, NJ (2009). https://doi.org/10.1002/9780470611821

[18] Apt, K.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003). https://doi.org/10.1017/CBO9780511615320 . https://www.cambridge.org/core/books/principles-of-constraint-programming/C008FB32571F66C3EE0EEEBDE1F98A7D Accessed 2025-01-17

[19] Hooker, J.N., Van Hoeve, W.-J.: Constraint programming and operations research. Constraints **23**(2), 172–195 (2018) https://doi.org/10.1007/s10601-017-9280-3 . Accessed 2025-01-17

[20] Cacchiani, V., Iori, M., Locatelli, A., Martello, S.: Knapsack problems — An overview of recent advances. Part II: Multiple, multidimensional, and quadratic knapsack problems. Computers & Operations Research **143**, 105693 (2022) https://doi.org/10.1016/j.cor.2021.105693 . Accessed 2026-01-01

[21] Feurer, M., Hutter, F.: Hyperparameter Optimization. In: Hutter, F., Kotthoff, L., Vanschoren, J. (eds.) Automated Machine Learning: Methods, Systems, Challenges. The Springer Series on Challenges in Machine Learning, pp.

3–33. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05318-5_1 . https://doi.org/10.1007/978-3-030-05318-5 Accessed 2024-01-13

[22] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B.: Algorithms for Hyper-Parameter Optimization. In: Advances in Neural Information Processing Systems, vol. 24. Curran Associates, Inc., Cambridge, US (2011). https://proceedings.neurips.cc/paper-files/paper/2011/hash/86e8f7ab32cfd12577bc2619bc635690-Abstract.html Accessed 2025-03-28

[23] Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential Model-Based Optimization for General Algorithm Configuration. In: Coello, C.A.C. (ed.) Learning and Intelligent Optimization, pp. 507–523. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25566-3_40

[24] Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2. AAAI'07, pp. 1152–1157. AAAI Press, Vancouver, British Columbia, Canada (2007)

[25] Hutter, F., Stuetzle, T., Leyton-Brown, K., Hoos, H.H.: ParamILS: An Automatic Algorithm Configuration Framework. Journal of Artificial Intelligence Research **36**, 267–306 (2009) https://doi.org/10.1613/jair.2861 . arXiv:1401.3492 [cs]. Accessed 2025-01-21

[26] Kushner, H.J.: A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise. Journal of Basic Engineering **86**(1), 97–106 (1964) https://doi.org/10.1115/1.3653121 . Accessed 2025-05-20

[27] Gan, W., Ji, Z., Liang, Y.: Acquisition Functions in Bayesian Optimization. In: 2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE), pp. 129–135 (2021). https://doi.org/10.1109/ICBASE53849.2021.00032 . https://ieeexplore.ieee.org/document/9696089 Accessed 2025-05-20

[28] Braußе, F., Khasidashvili, Z., Korovin, K.: Combining Constraint Solving and Bayesian Techniques for System Optimization. In: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, pp. 1788–1794. International Joint Conferences on Artificial Intelligence Organization, Vienna, Austria (2022). https://doi.org/10.24963/ijcai.2022/249 . https://www.ijcai.org/proceedings/2022/249 Accessed 2024-11-25

[29] Amadini, R., Gabbrielli, M., Mauro, J.: Portfolio approaches for constraint optimization problems. Annals of Mathematics and Artificial Intelligence **76**(1), 229–246 (2016) https://doi.org/10.1007/s10472-015-9459-5 . Accessed 2025-04-16

[30] Haddad, H., Talbot, P., Bouvry, P.: Selecting Search Strategy in Constraint Solvers using Bayesian Optimization. In: 2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI), pp. 764–773 (2024). https://doi.org/10.1109/ICTAI62512.2024.00113 . https://ieeexplore.ieee.org/abstract/document/10849432 Accessed 2025-08-01

[31] Boussemart, F., Lecoutre, C., Audemard, G., Piette, C.: XCSP3-core: A Format for Representing Constraint Satisfaction/Optimization Problems. arXiv. arXiv:2009.00514 [cs] (2024). https://doi.org/10.48550/arXiv.2009.00514 . http://arxiv.org/abs/2009.00514 Accessed 2025-04-16

[32] Régin, J.-C.: Generalized Arc Consistency for Global Cardinality Constraint. AAAI Press, Portland, Oregon, USA (1996). Journal Abbreviation: Proceedings AAAI'96 Pages: 215 Publication Title: Proceedings AAAI'96

[33] Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of global constraints. In: Proceedings of the 19th National Conference on Artifical Intelligence. AAAI'04, pp. 112–117. AAAI Press, San Jose, California (2004)

[34] CPMpy, D.T.: CPMpy: Constraint Programming and Modeling in Python — CPMpy 0.9.24 documentation (2025). https://cpmpy.readthedocs.io/en/latest/ Accessed 2025-10-16

[35] Prud'homme, C., Fages, J.-G.: Choco-solver: A Java library for constraint programming. Issue: 78 Pages: 4708 Publication Title: Journal of Open Source Software Volume: 7 original-date: 2011-11-04 (2022). https://doi.org/10.21105/joss.04708 . https://github.com/chocoteam/choco-solver Accessed 2024-06-27

[36] Bleukx, I., Berden, S., Coenen, L., Decleyre, N., Guns, T.: Model-Based Algorithm Configuration with Adaptive Capping and Prior Distributions. In: Schaus, P. (ed.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research vol. 13292, pp. 64–73. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-08011-1 . Series Title: Lecture Notes in Computer Science. https://link.springer.com/10.1007/978-3-031-08011-1 Accessed 2025-03-28