Selecting Search Strategy in Constraint Solvers using Bayesian Optimization

1st Hedieh Haddad[1,2], 2nd Pierre Talbot[2], 3rd Pascal Bouvry[1,2]
[1]*Interdisciplinary Centre for Security, Reliability and Trust (SnT), Luxembourg*
[2]*University of Luxembourg, Luxembourg*
hedieh.haddad@uni.lu, pierre.talbot@uni.lu, pascal.bouvry@uni.lu

***Abstract*—In the field of constraint programming, selecting the most effective search strategy for a new problem is a complex task. Despite the existence of numerous autonomous search strategies, the effectiveness of a strategy is highly problem-specific and no single strategy can universally excel. Therefore, for the solver's developers, it is difficult to find a good default strategy working across many problems. For the end-user, it is a daunting task to select the best search strategy, and they will usually rely on the solver's default, missing out better strategies.**

In this paper, we introduce the *probe and solve algorithm* which explores different search strategies in a probing phase, using a portion of the global timeout, and uses the best strategy found to solve the problem. By viewing the search strategy as hyperparameters, we leverage Bayesian optimization, a hyperparameter optimization technique well-known in machine learning but, to the best of our knowledge, not used in constraint programming. A key strength of our approach is to be generic and non-invasive: it can be used on top of any MiniZinc or XCSP3-compatible solvers, without modifying those. Further, *probe and solve* consistently achieved better results in the XCSP3 and MiniZinc competitions than the solver's default search and modern dynamic search strategies: DomWDeg/CACD, FrbaOnDom and PickOnDom, with the ACE and Choco constraint solvers.

***Index Terms*—Constraint programming, search strategies, Bayesian optimization, hyperparameter optimization.**

I. Introduction

Constraint programming (CP) is a computational paradigm that operates with mathematical relations, also known as constraints. This method offers a declarative way to represent a wide array of real-world problems, ranging from scheduling and vehicle routing to biology and musical composition [1].

One of the key advantages of CP is its adaptability. It supports a broad spectrum of constraints, including linear and non-linear, and those over discrete or continuous domains. This versatility sets it apart from other methods like SAT, which focuses on Boolean formulas [2], and linear programming [3], which is designed to solve linear constraints. CP emphasizes defining the problem itself, focusing on the 'what to model' rather than the 'how to solve'. Once the constraint model written, a general-purpose solver is responsible for the actual problem-solving process.

In order to find a solution to a problem, CP solvers use a search process to explore the space of possible solutions. The search process is guided by the domains of the variables and the constraints, and the goal is to find a solution that satisfies all the constraints. To guide the search process, CP solvers use a variety of search strategies to determine the order in which the solver explores the space of possible solutions and can have a significant impact on the performance of the solver.

However, as there is not a universally efficient algorithm for all problems, CP experts often need to customize the solver and devise specific search strategies for each problem to ensure efficiency [4], [5]. In this light, they have implemented some adaptive search strategies to autonomously and dynamically gather valuable information, throughout the search process, enabling more informed decision-making. Among these various search strategies, PickOnDom [6], FrbaOnDom [7], and DomWDeg/CACD [8], [9] stand out as three popular variable selection strategies utilized in constraint programming. Similarly to [6], we have compared these three strategies on the XCSP3 competition of 2023 [10], and the results show that none of these strategies can consistently outperform the others.

In this work, we introduce the *probe and solve algorithm* (PSA) which first probes the problem with different search strategies and then solve the problem with the best strategy found (Section III). By trading solving time for configuration time, we hope to find a better search strategy on average. During the probing phase, we use Bayesian optimization [11] to select the different search strategies to test. Although Bayesian optimization is a standard and efficient hyperparameter optimization (HPO) technique in the field of machine learning [12], [13], we are not aware of its usage in the realm of CP.

A search strategy is usually a combination of a variable and value selection strategies. We keep this simple view as it makes it easy to encode the search strategy into a set of hyperparameters to be optimized by an HPO method. From our experiments, when we considered more parameters—such as combination of strategies, or solver's parameters—the hyperparameters space became overly large and the efficiency of the approach decreased. It is also due to the fact that evaluating each configuration is a time-consuming process as it involves running the solver for some time. Hence, we have kept the hyperparameters space small, containing only the most impactful parameter: the search strategy.

The goal of HPO algorithms is to automatically tune the parameters of an underlying algorithm, but they also contain parameters that can impact the efficiency. A contribution of

This work is partially funded by the joint research programme UL/SnT–ILNAS on Technical Standardisation for Trustworthy ICT, Aerospace, and Construction.

PSA is to be *parameterless*. Indeed, PSA automatically adjusts the time used by each probe to gather valuable feedback and guide the hyperparameter search effectively. Furthemore, by an extensive experimental analysis, we found that probing for 20% of the global timeout is a robust default probing time.

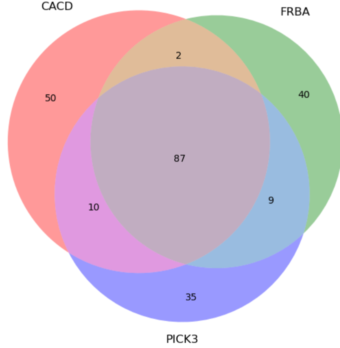


Fig. 1: Comparison of objective values for three different search strategies. Each circle in the diagram represents the instances solved by a specific heuristic. The overlapping areas of the circles indicate instances where the search strategies achieved identical objective values. For instance, CACD outperformed FRBA and PICK3 in 50 instances, matched the result of PICK3 in 10 instances, and all three strategies yielded identical results in 87 instances.

We evaluate PSA on a standard set of benchmark problems from the XCSP3 competition and the MiniZinc challenge [14]. The results are then compared with the default and most commonly used search strategies to fully leverage each solver's performance (Section V-E). The results demonstrate that the algorithm performs effectively within the XCSP3 framework and ACE solver [15], and MiniZinc with Choco solver [16], as detailed in Section V.

The contributions of this work are as follows:

- **Non-invasive.** PSA is generic and non-invasive as it can be used on top of any MiniZinc or XCSP3-compatible solvers, without modifying those (Section III).
- **Parameterless.** PSA operates without the need for parameter setting or optimization, eliminating the necessity for manual parameter setup.
- **Robustness.** The robust performance of PSA, even with short probing timeouts, is validated through statistical analysis as detailed in Section IV.
- **Efficiency.** PSA demonstrates superior results compared to the default solver strategy or individual dynamic search strategies, highlighting its efficiency (Section V).

II. Background

A. Constraint Programming

A constraint satisfaction problem (CSP) is a tuple $\langle X, D, C\rangle$ where X is a set of variables, D are the domains, and C is the set of constraints. By utilizing a constraint solver, the goal is to find solutions that satisfy these given constraints. An extension is the constraint optimization problem (COP) which is a tuple $\langle X, D, C, obj\rangle$ with $obj \in X$ where the goal is to find the best objective possible (either by minimization or maximization). Without loss of generalities, we suppose minimization problems in our definitions and algorithms. The constraint solver is essentially a backtracking procedure dividing the domains of the variable following a certain search strategy and pruning the domains using a form of logical inference called *propagation* [17]. Commonly used search strategies in constraint programming are a combination of variable and value selection strategies.

Different problem domains and characteristics may require different search strategies to achieve the best performance. Therefore, the design and evaluation of search strategies are ongoing research areas in CP, aiming to provide guidance and tools for effectively solving complex problems [18], [19].

B. Search Strategies

Search strategies are methods to explore the space of possible solutions for a constraint problem. A search strategy consists of two components: a variable selection strategy and a value selection strategy [20]. While it is possible for a search strategy to encompass more than these two components, within the context of this paper, we define a search strategy as the combination of these two elements.

Variable selection strategies determine the order in which variables are chosen for assignment during the search process, such as `input_order` (variables are chosen for assignment in the order they were input into the system) or `first_fail` (the variable with the smallest domain i.e., the fewest possible valid values, is chosen first) [21].

On the other hand, value selection strategies determine the order in which values are considered for assignment to variables, utilizing heuristics like selecting the minimum or maximum value or choosing randomly from the remaining values [22]. These strategies determine how the solver explores the domain of a selected variable. Examples of value choice strategies include selecting the minimum or maximum value, choosing randomly from the remaining values, or using heuristics based on problem-specific knowledge [23].

The choice of heuristics can have a significant impact on the efficiency and effectiveness of the search. Different heuristics may be suitable for different types of problems, constraints, and objective functions. Therefore, finding a good search strategy for a given problem is a challenging task.

C. Hyperparameter Optimization

Hyperparameter optimization (HPO) is the process of finding the optimal values for the hyperparameters of a machine learning algorithm, such as learning rate, regularization, or number of hidden units. Hyperparameters are the parameters that are not learned by the algorithm but are set by the user before training [24]. Choosing the right hyperparameters can have a significant impact on the performance and efficiency of the algorithm, but finding them can be challenging, especially for complex models and problems.

One of the main challenges of HPO is the trade-off between exploration and exploitation, that is, how to balance the search between trying new and potentially better values for the hyperparameters, and using the best values found so far [25]. Exploration can help to avoid getting stuck in local optima and discover new regions of the search space, but it can also waste time and resources on poor values. Exploitation can help to improve the performance and efficiency of the algorithm, but it can also miss out on better values that are not yet evaluated.

Several HPO methods have gained more popularity, including: *grid search*, *random search* [26], *hyper-band optimization* [27], *multi-armed bandit* [28], and *bayesian optimization*. Different HPO methods have different ways of dealing with this trade-off, and there is no one-size-fits-all solution that works for every problem and algorithm [25].

III. Probe and Solve Algorithm

In this section, we introduce the *probe and solve algorithm* (PSA), a universal method designed to pinpoint effective search strategies for tackling constraint problems. The algorithm operates in two phases: the probing phase and the solving phase. The probing phase navigates the landscape of search strategies using an HPO method, while the solving phase applies the most effective strategy to resolve the problem.

We denote S_{var} as the set of variable selection strategies' names (as recognized by the solvers), and S_{val} as the set of value selection strategies' names. The HPO method is tasked with optimizing two arrays of integers, representing S_{var} and S_{val} respectively. We define $HP := \mathcal{P}(S_{var} \times S_{val})$ as the set of all possible combinations of hyperparameters.

A. Algorithm

Let GT be the global timeout allocated to solve a CSP. A portion of this time denoted as PT (probing timeout), is specifically reserved for the probing phase. The value of PT is a crucial parameter in our algorithm, and we have conducted extensive experiments to optimize it within the context of the overall timeout GT (Section V).

The probing phase will run for a portion of GT, which we call probing timeout PT, which is a critical parameter that we experimented with to optimize our algorithm. A probing ratio of 20% strikes the right balance between exploration and exploitation, as shown in Section IV.

During the probing phase, PSA evaluates a variety of search strategies from the set HP. Each strategy is assessed for a limited duration, initially set to 5 seconds, denoted CT (current timeout). This timeframe not only accommodates the computational requirements of simpler problems but also factors in the necessary overhead for transmitting information to the solver and retrieving the solutions.

The algorithm attempts to solve the problem within this timeout. If the solver successfully finds the objective within this timeframe, it continues to run the next experiment in the probing phase using this timeout. However, if the solver fails to find any objective result within the timeout, the approach adapts by extending the defined timeout. This increment is determined by a geometric coefficient, set to 1.2, which multiplies the current timeout CT, thereby increasing it.

The timeout continues increasing until an objective is found or PT is reached, using the calculation of elapsed time, denoted by ET. This adaptive approach ensures that the algorithm can effectively handle problems of various complexities while maintaining efficiency. Specifically, in scenarios where a solution cannot be found within 5 seconds.

Choosing the probing timeout too large will reduce the number of combinations we can test and the effectiveness of the HPO method. But choosing it too small will prevent us from getting any solution and the comparison between two runs will be harder. In our experiments, we compare two runs using the objective value found and in the case of ties, the time it took to reach that objective value. At the end of the probing phase, we obtain a rank of the search strategies tested, and we select the best one for the solving phase.

We now define more explicitly our approach in Algorithm 1.

Algorithm 1 Probe and Solve Algorithm (PSA)

function PSA($\langle X, D, C, obj\rangle, hpo, HP, GT, PT$)
  Initialize ET to 0 seconds
  Initialize $best_obj$ to ∞
  while $ET < PT$ **do**
    $psolve \leftarrow \lambda s.solve(\langle X, D, C, obj\rangle, s, CT)$
    $ranking, obj \leftarrow hpo(HP, psolve)$
    $ET \leftarrow ET + CT$
    if $obj \neq \infty$ **then**
      $min(obj, best_obj)$
    else
      $CT \leftarrow CT \times Geometric_Coefficient$
    end if
  end while
  if $best_obj = \infty$ **then**
    return $solve(\langle X, D, C, obj\rangle, ranking[0], GT - PT)$
  else
    return $min(best_obj, solve(\langle X, D, C \wedge obj < best_obj, obj\rangle, ranking[0], GT - PT))$
  end if
end function

The algorithm accepts a COP $\langle X, D, C, obj\rangle$, a hyperparameter function hpo, the set of available search strategies HP, and two timeouts: GT and PT.

The HPO method, denoted as $hpo(HP, psolve)$, takes the set of search strategies and an evaluation function. It returns the ranking of the best search strategies and the best objective found so far.

We assume a function $solve(\langle X, D, C, obj\rangle, s, T)$ that optimizes a given COP using the search strategy $s \in HP$ under the timeout T. The solving phase operates for the remainder of the global timeout and employs the best search strategy identified in the probing phase to solve the constraint problem.

IV. Validating the Robustness in Shorter Timeouts

In our study, we employed two statistical methods, Spearman's rank correlation [29] and Kendall's tau [30], to analyze the results of the probing phase. The aim was to determine whether the algorithm could identify effective search strategies based on the rankings observed at both short and global timeouts, and to understand how these rankings at shorter timeouts correlate with those obtained at the global timeout.

We randomly selected three instances and ran them 10 times, each time recording the intermediary results and ranking the search strategies. For each run, we separated the ranking of the results after different timeouts: 5%, 10%, 20%, 50%, and 100% of the global timeout. We then examined whether the rankings of the results, evaluated based on the objective function were correlated. This allowed us to investigate whether the behavior of the search strategies remained consistent across different runs and for the specific problem.

The correlation coefficient ranges from -1 to 1, where 1 indicates a perfect positive correlation, -1 indicates a perfect negative correlation, and 0 indicates no correlation. In our context, a high positive correlation suggests that the rankings obtained from a specific run with a specific timeout are similar to those obtained from another run with the same timeout.

This implies that if we run the approach with a fraction of the global timeout and it yields a ranking of the search strategies, the behavior of the search strategies tends to mirror the behavior observed with the same instance and with the same fraction of the global timeout but in a different run. This holds true even though the seed for the Bayesian optimization varies and PSA provides a different set and combination of search strategies each time. Therefore, if a superior search strategy is identified during a specific run, we can be reasonably confident that the same strategy would also be deemed superior when evaluated over another run.

For instance, in *KidneyExchange-4-081*, the table shows an average correlation of 0.83 for all runs at first 5% of the global timeout to each other. This suggests that in all 10 runs of this specific instance, the behavior of search strategies in the first 5% of the global timeout were mostly the same as each other and they were correlated in 83% of the time.

The result of ranking correlations can be seen in Table I.

V. Experiments

In this section, we evaluate the performance of the *PSA* on a set of benchmark problems from different domains.

A. Experimental Setup

The experiments presented, were carried out using a high-performance computing facility. The computing time equates to approximately 2500 node-hours. The technical specifications of a cluster compute node are: 2xAMD Epyc ROME 7H12 @ 2.6 GHz [64c/280W] processor with 256 GB RAM.

Our research relies on the data from the fifth international XCSP3 constraint solver competition held in 2023 [15]. Specifically, we focus on all the COP problems derived from this competition, amounting to a total of 250 instances, covering various types of constraints and objective functions. We use ACE [15] as our main solver and Bayesian optimization as the main HPO method to generate and evaluate different combinations of search strategies as hyperparameters. For each problem, we align our global timeout setting of 1200 seconds with the one used by the XCSP3 Challenge.

Our aim is to demonstrate that PSA yields promising results. Therefore, we have chosen to set the probing phase timeout to different percentages of the global timeout, namely: 5%, 10%, 20%, 50%, and 100%. This allows to compare the efficiency and applicability of this timeout and to determine whether a shorter or longer probing phase yields better results.

B. Implementation

Our approach is implemented in Python, leveraging several libraries for optimization and CP. The key libraries used include Skopt [31] for HPO, Minizinc and PyCSP3 library for CP.

The search strategies of the solvers are described declaratively in a JSON file. This allows us to easily modify and experiment with different set of available strategies without changing the core code.

For instance, a simplified example of the ACE solver within the XCSP3 looks like this:

```
"XCSP3": {
 "Search-Strategy": {
  "varh_values":
   ["PickOnDom", "FrbaOnDom", "WdegOnDom"],
  "valh_values":
   ["First", "Median"]}}
```

In this example, `PickOnDom`, `FrbaOnDom` and `WdegOnDom` are the variable selection strategies, while `First` and `Median` are the value selection strategy.

C. XCSP3 Benchmark with ACE Solver

In this subsection, we evaluate the performance of PSA on the XCSP3 problems. XCSP3 is an XML-based format designed to represent instances of combinatorial constrained problems from the perspective of CP. It is an intermediate integrated format that can represent each instance separately while preserving its structure [10].

We compare the outcomes of PSA with four baselines. These baselines include three popular variable selection strategies: PICK3 (PickOnDom which is set with a pick degree of three and linked with variables when constraint propagation concludes with a conflict, as it was identified as the most effective degree [6]), DomWDeg/CACD, FrbaOnDom and the solver's default strategy. Performance is evaluated based on the objective value and the solving time. For the three popular variable selection strategies, the value selection strategy was left to the default of the solver as done in [6]. For the default search strategy of solvers, we do not specify any variable or value selection strategy, leaving this decision to the solver.

Instance	5%		10%		20%		50%	
	Spearman	Kendall Tau	Spearman	Kendall Tau	Spearman	Kendall Tau	Spearman	Kendall Tau
CarpetCutting-test05	0.94	0.83	0.96	0.92	0.91	0.84	0.89	0.81
GeneralizedMKP-OR05x100-75-1	0.99	0.99	0.99	0.99	0.92	0.84	0.91	0.80
RIP-25-0-j120-01-01	-0.33	-0.33	0.88	0.79	0.92	0.82	0.97	0.89
KidneyExchange-4-081	0.83	0.83	0.87	0.84	0.90	0.82	0.93	0.82

TABLE I: Average percentage of ranking correlation for the same percentage of the global timeout in 10 different runs - (5, 10, 20, 50)% of global timeout

First, we aimed to observe the impact of these different combinations of search strategies on the solver's performance. We strive to identify the best search strategy using PSA. This is done by exploring all the variable selection strategies and value selection strategies offered by the ACE solver itself.

The solver provides an array of strategies for both variable selection and value selection, as shown below:

- **Variable selection strategies:**

  {RunRobin, Wdeg, Memory, PickOnDom, FrOnDom, WdegOnDom, ProcOnDom, Regret, FrbaOnDom, Ddeg}

- **Value selection strategies:**

  {Dist, OccsR, Median, AsgsFp, Bivs2, First, AsgsFm, Last, Robin, RunRobin, Bivs, InternDist, Occs, FlrsE}

This allows to thoroughly evaluate the effectiveness of different search strategies in improving the solver's performance.

Fig. 2 illustrates the performance comparison between PSA, with a probing timeout ratio of 0.2, and the four baseline strategies. Each bar in the figure represents a specific baseline. The height of the bars indicates the percentage of instances where the performance of PSA either surpassed, matched, or was surpassed by the baseline. This figure also provides insight into the distribution of percentages across different models where PSA outperformed the baselines at this specific ratio.

In Fig. 2a, PSA performs better in 29.49% of the models than the default solver, and both yield equal results in 44.87% of the models, indicating that neither strategy consistently outperforms the other across all scenarios.

In the second step, we aim to evaluate the effectiveness of dynamic search strategies in comparison to static ones. Consequently, we select search strategies that are simpler to implement and do not require collecting statistics of propagators. The results obtained using a simple subset of variable and value selection strategies with a probing timeout ratio of 0.2, can be seen in Fig. 2b.

Variable and value selection strategies are shown below:

Variable selection strategies:

{Deg, Rand, Lexico, Srand}

Value selection strategies:

{Vals, First, Last, RunRobin, Robin, Srand, Rand}

The results indicate that PSA method surpasses the baseline methods in approximately 12% to 20% of the instances. However, the baseline methods demonstrate superior performance, outperforming PSA in around 50% of the instances, considering the fact that the baselines utilize dynamic methods to calculate and update weights. This could be interpreted as an indication that dynamic approaches surpass static approaches in this context. This superior performance of dynamic approaches could be attributed to their ability to adapt to changing conditions during the search process. Unlike static strategies, which use a fixed method throughout, dynamic strategies can adjust their methods based on the current state of the search. This adaptability allows them to navigate the search space more effectively, potentially avoiding local optima and finding better solutions.

Given that static search strategies appear to be less effective than dynamic ones, we employed our algorithm on a subset of three dynamic variable selection strategies. This was done to identify which subset of search strategies could yield enhanced performance. Consequently, we selected the three baselines, namely: PICK3, CACD, and FRBA as a subset of our variable selection strategies. The results of this approach with a probing timeout ratio of 0.2, can be observed in Fig. 2c.

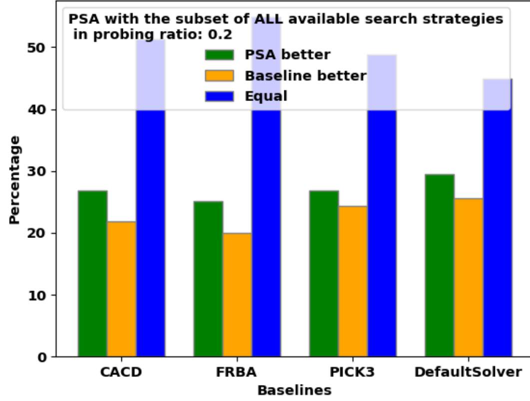
In Fig. 2c, at a ratio of 0.2, PSA performed better in 23.98% of the models than FRBA, while this baseline performed better in 18.70% of the models. The results were equal in 57.32% of the models. This pattern is observed across different ratios and heuristics, with some variations, and PSA demonstrated a competitive performance against the baselines although PSA takes some time to probe.

D. Minizinc Benchmark with Choco Solver

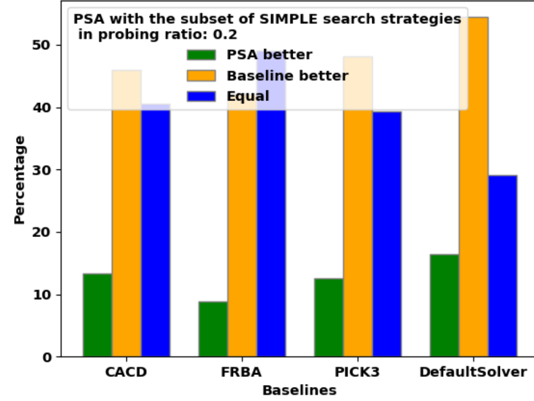
In this subsection, we concentrate on the efficacy of PSA when applied to the MiniZinc benchmark. MiniZinc is an open-source constraint modeling language. The MiniZinc models are subsequently compiled into FlatZinc, a solver input language comprehensible by a broad spectrum of solvers [14]. For this study, the utilized solver is Choco.

We compare the outcomes of our algorithm with the default search strategy, which is the combination of (`DomWDeg`, `Indomain_Min`) [32]. This combination is the preferred choice for the solver's default in the Choco solver.

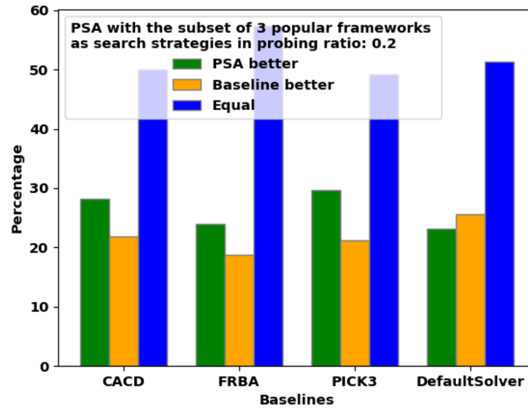
The results of PSA on the MiniZinc benchmark problems, shown in Fig. 3, demonstrate its performance across a variety of problems. The results are more competitive than the solver's default strategy. When the ratio is set at 0.5, PSA outperforms the default search strategy, achieving better results in 28.42% of the instances. This is the highest percentage of instances where PSA performs better among all the ratios tested. Gen-



(a) All available S_{var} and S_{val} in comparison with the baselines.



(b) Simple set of S_{var} and S_{val} in comparison with the baselines.



(c) S_{var}={$PICK3$, $CACD$, $FRBA$} and all available S_{val} in comparison with the baselines.

Fig. 2: Comparative performance of PSA and baselines: an examination of variable and value selection strategies with probing timeout ratio 0.2. The comparison is conducted across different baselines and is categorized into three sections: (a) comprehensive analysis using the set of all variable/value selection strategies (b) analysis using the set of simple variable/value selection strategies (c) analysis using the set of three popular frameworks as variable selection strategies with the set of all value selection strategies.

erally, PSA has demonstrated its potential by outperforming the default search strategy and baselines in certain scenarios.

The process of using MiniZinc involves compiling the models into FlatZinc [33] and then sending them to the solver. This process is inherently time-consuming as it involves both compilation and communication overheads. To address this, we have decided to compile the model to FlatZinc once and use it for the entire process of the experiment rather than in each run, significantly reducing the time and computational resources required. This change has contributed to the improved performance of PSA in the MiniZinc framework.

E. Detailed Analysis and PSA Performance

The comprehensive results derive from the various bar charts and tables and are shown in Table II. This includes a subset of variable and value selection strategies, both static and dynamic search strategies. These reveal that dynamic strategies tend to yield superior results. This can be attributed to several factors such as their ability to adapt to changing conditions, their capacity to learn from past decisions, and their potential to explore the search space more effectively.

We conduct an array of tests using different probing ratios, specifically 5%, 10%, 20%, 50%, and 100% of the global timeout. The results indicate that a probing ratio of 20% generally outperforms the others for our specific set of problems. Additionally, we introduce two new columns in our results table: *Fallback to Default* and *Same Search Strategy*. The *Fallback to Default* column quantifies the instances where PSA is unable to determine a search strategy within the given timeframe, leading to the application of the default search strategy during the solving phase. The *Same Search Strategy*

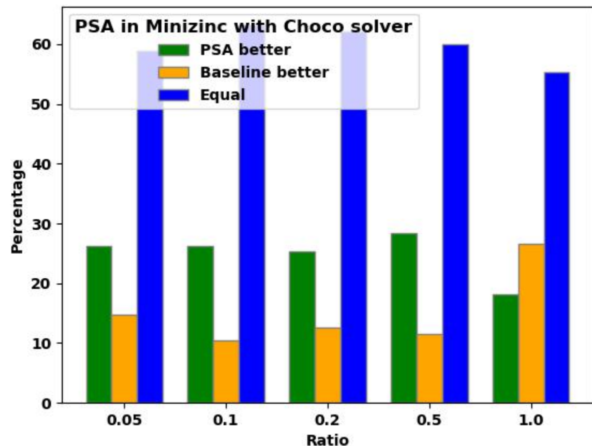


Fig. 3: Performance comparison of PSA and solver's default across different ratios in Minizinc framework with Choco solver.

column denotes the instances where PSA identifies a search strategy identical to the solver's default search strategy after the probing phase.

VI. Related Work

The application of artificial intelligence techniques to constraint solving has not been well explored in research. Some studies overview the potential of machine learning in optimizing constraint solving processes [34].

Recently, a new constraint programming solver, SeaPearl, was introduced [35] which supports machine learning routines to learn branching decisions using reinforcement learning. This represents a significant step forward in the integration of machine learning and constraint programming.

In another study [36], a novel algorithmic framework is introduced, combining multi-armed bandits and restarts to optimize variable selection strategies in constraint-based applications. This approach is designed to be autonomous, adjusting solver parameters to efficiently handle instances without manual tuning, also an adaptive variant of Successive Halving that exploits Luby's universal restart sequence is presented [37].

In another study [38] they introduce the concept of deep heuristics, a data-driven approach to learn extended versions of a given variable selection strategy. It demonstrates that deep heuristics, which can look ahead arbitrarily-many levels in the search tree, solve 20% more problem instances while improving on overall runtime for certain benchmark problems.

Despite the growing interest in using machine learning for optimizing constraint solvers and search strategies, there is much to explore, especially in applying hyperparameter optimization techniques to constraint programming. Our work focuses on this area, offering a black-box method that requires no modifications to the solver, enhancing its flexibility and applicability across different constraint programming scenarios.

VII. Conclusion

In this research, we explored the use of hyperparameter optimization in determining effective search strategies for constraint programming solvers. Our aim was to develop a simple, universal method that enables users, particularly those new to constraint problem modeling, to swiftly distinguish between efficient and inefficient search strategies. This is especially beneficial when there is a lack of initial insight into the potential effectiveness of different search strategies.

Our results indicate that PSA can surpass ACE and Choco's default strategies and recent dynamic strategies. It is noteworthy that PSA produced performance metrics that were comparable to those of the solver's default search strategy and the baselines. This is encouraging as it suggests potential for further improvement and better results in future studies. Furthermore, when using the Choco solver in Minizinc, it consistently produced results that were predominantly superior to the solver's default search strategy.

The result also showed that there is a strong correlation between results obtained within shorter timeouts and those achieved in full timeouts. This implies that the same ranking of results can likely be obtained during the probing phase, thereby validating our approach.

There were instances where no combination of variable and value strategies could surpass the solver's default. However, our approach still managed to produce promising results.

This research contributes to the ongoing development of more effective tools and techniques for solving complex constraint problems across various domains. It serves as a foundation for the development of more advanced and efficient constraint problem-solving methodologies. We are eager to continue refining our approach and anticipate achieving even better results in future studies.

TABLE II: Comprehensive results for all the possible ratios in comparison with the baselines.

Method		Baselines	Results (%)			Additional Results (%)	
			PSA Better	Baselines Better	Equal Results	Fallback to default	Same search strategy
XCSP3 with ACE solver All available S_{var} and S_{val}	0.05	CACD	23.11	**24.37**	52.52	25.60	0.81
		FRBA	20.73	**25.20**	54.07		
		PICK3	21.95	**26.02**	52.03		
		Default	19.23	**30.77**	50.00		
	0.1	CACD	23.85	**25.94**	50.21	16.19	0.80
		FRBA	**25.10**	21.86	53.04		
		PICK3	**25.91**	24.29	49.80		
		Default	**27.85**	24.05	48.10		
	0.2	CACD	**26.89**	21.85	51.26	13.0	1.21
		FRBA	**25.20**	19.92	54.88		
		PICK3	**26.83**	24.39	48.78		
		Default	**29.49**	25.64	44.87		
	0.5	CACD	**25.63**	23.95	50.42	8.82	1.56
		FRBA	**23.98**	21.14	54.88		
		PICK3	**26.02**	24.39	49.59		
		Default	**28.21**	24.36	47.44		
	1.0	CACD	2.94	**26.47**	70.59	8.53	2.40
		FRBA	3.25	**25.20**	71.54		
		PICK3	2.85	**26.42**	70.73		
		Default	2.56	**25.64**	71.79		
XCSP3 with ACE solver Simple set of S_{var} and S_{val}	0.05	CACD	13.08	**47.26**	39.66	28.16	0.00
		FRBA	11.02	**42.04**	46.94		
		PICK3	15.51	**47.35**	37.14		
		Default	14.29	**51.95**	33.77		
	0.1	CACD	15.90	**43.51**	40.59	23.88	0.00
		FRBA	11.34	**40.49**	48.18		
		PICK3	15.79	**44.53**	39.68		
		Default	16.46	**49.37**	34.18		
	0.2	CACD	13.39	**46.03**	40.59	18.62	0.00
		FRBA	8.91	**42.11**	48.99		
		PICK3	12.55	**48.18**	39.27		
		Default	16.46	**54.43**	29.11		
	0.5	CACD	12.61	**50.42**	36.97	16.26	0.00
		FRBA	9.35	**44.31**	46.34		
		PICK3	10.57	**53.25**	36.18		
		Default	17.95	**56.41**	25.64		
	1.0	CACD	0.00	**9.62**	90.38	15.38	0.00
		FRBA	0.00	**9.72**	90.28		
		PICK3	0.00	**9.72**	90.28		
		Default	0.00	**13.92**	86.08		
XCSP3 with ACE solver S_{var}={PICK3, CACD, FRBA} All available S_{val}	0.05	CACD	**25.52**	19.67	54.81	22.26	0.40
		FRBA	**23.89**	18.22	57.89		
		PICK3	**26.32**	23.08	50.61		
		Default	20.25	**27.85**	51.90		
	0.1	CACD	**27.31**	20.59	52.10	18.29	0.81
		FRBA	**27.24**	16.67	56.10		
		PICK3	**28.86**	22.76	48.37		
		Default	**19.23**	30.77	50.00		
	0.2	CACD	**28.15**	21.85	50.00	15.04	8.13
		FRBA	**23.98**	18.70	57.32		
		PICK3	**29.67**	21.14	49.19		
		Default	**23.08**	25.64	51.28		
	0.5	CACD	**25.52**	24.69	49.79	8.90	2.42
		FRBA	**23.89**	22.27	53.85		
		PICK3	**26.32**	25.10	48.58		
		Default	26.58	**27.85**	45.57		
	1.0	CACD	3.77	**28.03**	68.20	8.50	2.83
		FRBA	2.83	**26.72**	70.45		
		PICK3	2.83	**27.94**	69.23		
		Default	1.27	**30.38**	68.35		
Minizinc with Choco solver	0.05	Default	**26.32**	14.74	58.95	36.84	0.00
	0.1	Default	**26.32**	10.53	63.16	31.58	1.05
	0.2	Default	**25.26**	12.63	62.11	22.11	0.00
	0.5	Default	**28.42**	11.58	60.00	23.68	1.05
	1.0	Default	18.09	**26.60**	55.32	18.30	1.06

References

[1] F. Rossi, P. v. Beek, and T. Walsh, "Handbook of Constraint Programming [Book]," Aug. 2006, iSBN: 9780080463803. [Online]. Available: https://www.oreilly.com/library/view/handbook-of-constraint/9780444527264/

[2] A. Biere, M. Heule, H. Van Maaren, and T. Walsh, "Handbook of Satisfiability: Second Edition," ser. Frontiers in Artificial Intelligence and Applications, vol. 336. IOS Press, Feb. 2021. [Online]. Available: http://ebooks.iospress.nl/doi/10.3233/FAIA336

[3] J. M. Bernd Gärtner, *Understanding and Using Linear Programming*, ser. Universitext. Berlin, Heidelberg: Springer, 2007. [Online]. Available: http://doi.org/10.1007/978-3-540-30717-4

[4] H. Simonis and B. O'Sullivan, "Search Strategies for Rectangle Packing," in *Principles and Practice of Constraint Programming*, P. J. Stuckey, Ed. Berlin, Heidelberg: Springer, 2008, pp. 52–66. [Online]. Available: https://doi.org/10.1007/978-3-540-85958-1_4

[5] E. Teppan, G. Friedrich, and A. Falkner, "QuickPup: A Heuristic Backtracking Algorithm for the Partner Units Configuration Problem," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 26, no. 2, pp. 2329–2334, Jul. 2012. [Online]. Available: https://doi.org/10.1609/aaai.v26i2.18979

[6] G. Audemard, C. Lecoutre, and C. Prud'homme, "Guiding Backtrack Search by Tracking Variables During Constraint Propagation." Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2023.9

[7] H. Li, M. Yin, and Z. Li, "Failure Based Variable Ordering Heuristics for Solving CSPs (Short Paper)," in *LIPIcs, Volume 210, CP 2021*, vol. 210, 2021. [Online]. Available: https://doi.org/10.4230/LIPICS.CP.2021.9

[8] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," Jan. 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:17864847

[9] W. Hugues, C. Lecoutre, A. Paparrizou, and S. Tabary, "Refining Constraint Weighting," Nov. 2019, pp. 71–77. [Online]. Available: https://doi.org/10.1109/ICTAI.2019.00019

[10] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette, "XCSP3: An Integrated Format for Benchmarking Combinatorial Constrained Problems," Nov. 2022, arXiv:1611.03398 [cs]. [Online]. Available: http://arxiv.org/abs/1611.03398

[11] J. Ungredda and J. Branke, "Bayesian Optimisation for Constrained Problems," May 2021, arXiv:2105.13245 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2105.13245

[12] T. Agrawal, "Bayesian Optimization," in *Hyperparameter Optimization in Machine Learning: Make Your Machine Learning and Deep Learning Models More Efficient*, T. Agrawal, Ed. Berkeley, CA: Apress, 2021, pp. 81–108. [Online]. Available: https://doi.org/10.1007/978-1-4842-6579-6_4

[13] A. Tanay, *Hyperparameter Optimization in Machine Learning: Make Your Machine Learning and Deep Learning Models More Efficient*. Berkeley, CA: Apress, 2021. [Online]. Available: https://doi.org/10.1007/978-1-4842-6579-6

[14] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer, "The MiniZinc Challenge 2008–2013," *AI Magazine*, vol. 35, no. 2, pp. 55–60, Jun. 2014, number: 2. [Online]. Available: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2539

[15] C. Lecoutre, "ACE, a generic constraint solver," Jan. 2023, arXiv:2302.05405 [cs]. [Online]. Available: http://arxiv.org/abs/2302.05405

[16] C. Prud'homme and J.-G. Fages, "Choco-solver: A Java library for constraint programming," 2022, issue: 78 Pages: 4708 Publication Title: Journal of Open Source Software Volume: 7 original-date: 2011-11-04T09:09:18Z. [Online]. Available: https://github.com/chocoteam/choco-solver

[17] K. Apt, "Constraint propagation algorithms," in *Principles of Constraint Programming*. Cambridge: Cambridge University Press, 2003, pp. 254–298. [Online]. Available: https://doi.org/10.1017/CBO9780511615320.007

[18] P. Talbot, "Spacetime Programming: A Synchronous Language for Composable Search Strategies," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–16. [Online]. Available: https://doi.org/10.1145/3354166.3354183

[19] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. J. Stuckey, "Search combinators," *Springer, Berlin, Heidelberg*, vol. 18, no. 2, pp. 269–305, 2013. [Online]. Available: http://link.springer.com/article/10.1007/s10601-012-9137-8

[20] A. Palmieri and G. Perez, "Objective as a Feature for Robust Search Strategies," in *Principles and Practice of Constraint Programming*, J. Hooker, Ed. Cham: Springer International Publishing, 2018, pp. 328–344. [Online]. Available: https://www.doi.org/10.1007/978-3-319-98334-9_22

[21] D. Grimes and R. J. Wallace, "Sampling Strategies and Variable Selection in Weighted Degree Heuristics," *Lecture Notes in Computer Science*, pp. 831–838, Jan. 2007. [Online]. Available: https://www.academia.edu/55629464/Sampling_strategies_and_variable_selection_in_constraint_satisfac_tion_search

[22] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence*, vol. 14, no. 3, pp. 263–313, Oct. 1980. [Online]. Available: https://doi.org/10.1016/0004-3702(80)90051-X

[23] P. Refalo, "Impact-Based Search Strategies for Constraint Programming," in *Principles and Practice of Constraint Programming – CP 2004*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and M. Wallace, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, vol. 3258, pp. 557–571, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-30201-8_41

[24] M. Wistuba, N. Schilling, and L. Schmidt-Thieme, "Hyperparameter Search Space Pruning – A New Component for Sequential Model-Based Hyperparameter Optimization," in *Machine Learning and Knowledge Discovery in Databases*, ser. Lecture Notes in Computer Science, A. Appice, P. P. Rodrigues, V. Santos Costa, J. Gama, A. Jorge, and C. Soares, Eds. Cham: Springer International Publishing, 2015, pp. 104–119. [Online]. Available: https://www.doi.org/10.1007/978-3-319-23525-7_7

[25] M. Feurer and F. Hutter, "Hyperparameter Optimization," in *Automated Machine Learning: Methods, Systems, Challenges*, ser. The Springer Series on Challenges in Machine Learning, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds. Cham: Springer International Publishing, 2019, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-030-05318-5_1

[26] P. Liashchynskyi and P. Liashchynskyi, "Grid Search, Random Search, Genetic Algorithm: A Big Comparison for NAS," Dec. 2019, arXiv:1912.06059 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1912.06059

[27] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization," Jun. 2018, arXiv:1603.06560 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1603.06560

[28] A. Slivkins, "Introduction to Multi-Armed Bandits," Apr. 2024, arXiv:1904.07272 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1904.07272

[29] Y. Dodge, "Spearman Rank Correlation Coefficient," in *The Concise Encyclopedia of Statistics*. New York, NY: Springer, 2008, pp. 502–505. [Online]. Available: https://doi.org/10.1007/978-0-387-32833-1_379

[30] L. Puka, "Kendall's Tau," in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Berlin, Heidelberg: Springer, 2011, pp. 713–715. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_324

[31] "scikit-optimize: sequential model-based optimization in Python — scikit-optimize 0.9.0 documentation." [Online]. Available: https://scikit-optimize.github.io/dev/

[32] "The MiniZinc Handbook — The MiniZinc Handbook 2.3.0." [Online]. Available: https://docs.minizinc.dev/en/2.3.0/index.html

[33] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a Standard CP Modelling Language," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed. Berlin, Heidelberg: Springer, 2007, pp. 529–543. [Online]. Available: https://www.doi.org/10.1007/978-3-540-74970-7_38

[34] A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V.-M. Le, K. Pilsl, M. Enzelsberger, and T. N. T. Tran, "An overview of machine learning techniques in constraint solving," *Journal of Intelligent Information Systems*, vol. 58, no. 1, pp. 91–118, Feb. 2022. [Online]. Available: https://doi.org/10.1007/s10844-021-00666-5

[35] F. Chalumeau, I. Coulon, Q. Cappart, and L.-M. Rousseau, "SeaPearl: A Constraint Programming Solver guided by Reinforcement Learning," Apr. 2021, arXiv:2102.09193 [cs]. [Online]. Available: http://arxiv.org/abs/2102.09193

[36] H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou, and S. Tabary, "Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts," *Santiago de Compostela*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221714020

[37] F. Koriche, C. Lecoutre, A. Paparrizou, and H. Wattez, "Best Heuristic Identification for Constraint Satisfaction," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*. Vienna, Austria: International Joint Conferences on Artificial Intelligence Organization, Jul. 2022, pp. 1859–1865. [Online]. Available: https://www.ijcai.org/proceedings/2022/258

[38] F. Doolaard and N. Yorke-Smith, "Online Learning of Deeper Variable Ordering Heuristics for Constraint Optimisation," Oct. 2022. [Online]. Available: https://doi.org/10.1007/s10472-022-09816-z