Presented at: IJCAI'21 Workshop on Data Science Meets Optimisation, Montreal, Canada, August 2021.

Online Learning of Deeper Variable Ordering Heuristics for Constraint Optimisation Problems

Floris Doolaard , Neil Yorke-Smith*
Delft University of Technology, The Netherlands
f.p.doolaard@student.tudelft.nl, n.yorke-smith@tudelft.nl

Abstract

Solvers for constraint optimisation problems exploit variable and value ordering heuristics. Numerous expert-designed heuristics exist, while recent research uses machine learning to learn novel heuristics. We introduce the concept of *deep heuristics*, a data-driven approach to learn extended versions of a given variable ordering heuristic. First, for a problem instance, an initial probing phase collects data, from which a deep heuristic function is learned. The learned heuristics can look ahead arbitrarily-many levels in the search tree instead of a 'shallow' localised lookup for classical heuristics. We demonstrate deep variable ordering heuristics based on the smallest, anti first-fail, and maximum regret heuristics. The results show that deep heuristics solve 20% more problem instances while improving on overall runtime for the Open Stacks and Evilshop benchmark problems.

1 Introduction

The order in which the variables are chosen can have significant effect on the total runtime of a constraint optimisation problem (COP) solver [Gent *et al.*, 1996]. Various variable ordering heuristics have been designed by human experts [Haralick and Elliott, 1980; Boussemart *et al.*, 2004; Refalo, 2004]. Recent work also acquires dedicated heuristics using machine learning (ML), or learns which of a given set of heuristic to use [Alanazi and Lehre, 2016; Xia and Yap, 2018; Khalil *et al.*, 2017; Chalumeau *et al.*, 2021]. However, both classical and learned heuristics are based on the current search node. Further, some ML methods may require significant offline training time before starting search, while others face the familiar ML difficulty of generalizing to unseen instances.

We address the situation of online solving of unseen optimisation problems. We introduce the concept of *deep heuristics*, a data-driven approach to learn extended versions of a given heuristic. We adopt regression analysis, a simple ML technique which requires little data or training time. The acquired deep variable ordering heuristics are approximation functions that look at multiple levels of a search tree with the aim of generalizing better than classical heuristics.

We demonstrate deep heuristics derived from three representative variable ordering heuristics: smallest, anti first-fail, and maximum regret. On the MiniZinc benchmarks, we empirically compare deep and classical 'shallow' versions of these heuristics. The results indicate that deep heuristics solve 20% more problem instances while also improving on overall runtime for the Open Stacks and Evilshop problems.

As summarised in Figure 1, we implement deep heuristics in the open source Gecode solver [Schulte *et al.*, 2019]. Given a problem instance, an initial probing phase employs pseudo-random search to gather a variety of variable-value assignments. This data is then utilized by the machine learning component to acquire a deep heuristic function. Then second, during solving, given the current search state, the solver can predict scores with the learned model and select the variable with the best predicted score. Third, to leverage the pseudo-random nature of the probing data, a restart-based search strategy allows for multiple ML models to be learned, increasing the chance of finding solutions.

In contrast many works combining combinatorial optimisation with learning, our aim is an online setting where training time is included in the total solving time. Related to our work is the predict-and-optimise paradigm [Mandi *et al.*, 2020] in that we do not directly learn to solve an optimisation problem; hence the quality of the learned function per se is less important than its use to improve the subsequent solving of the combinatorial problem.

The deep heuristics in our work depend on human expert designed variable ordering heuristics. Such heuristics are employed in specific heuristic–problem combinations [Wallace, 2008], while our work is more flexible by learning variable ordering heuristics based on a problem's search tree.

Closer to our work, Chu and Stuckey [2015] use online learning to acquire value heuristics, using partial least squares regression to learn the score function. Our approach differs in that, firstly, we learn variable ordering heuristics and we utilize a more complex score function, utilizing multiple heuristic score functions over multiple nodes. Second, our framework uses a restart-based approach in probing and in search.

Glankwamdee and Linderoth [2006] use lookahead branching on grand-child nodes in a mixed integer program (MIP), finding that information from these nodes often re-

*Corresponding author

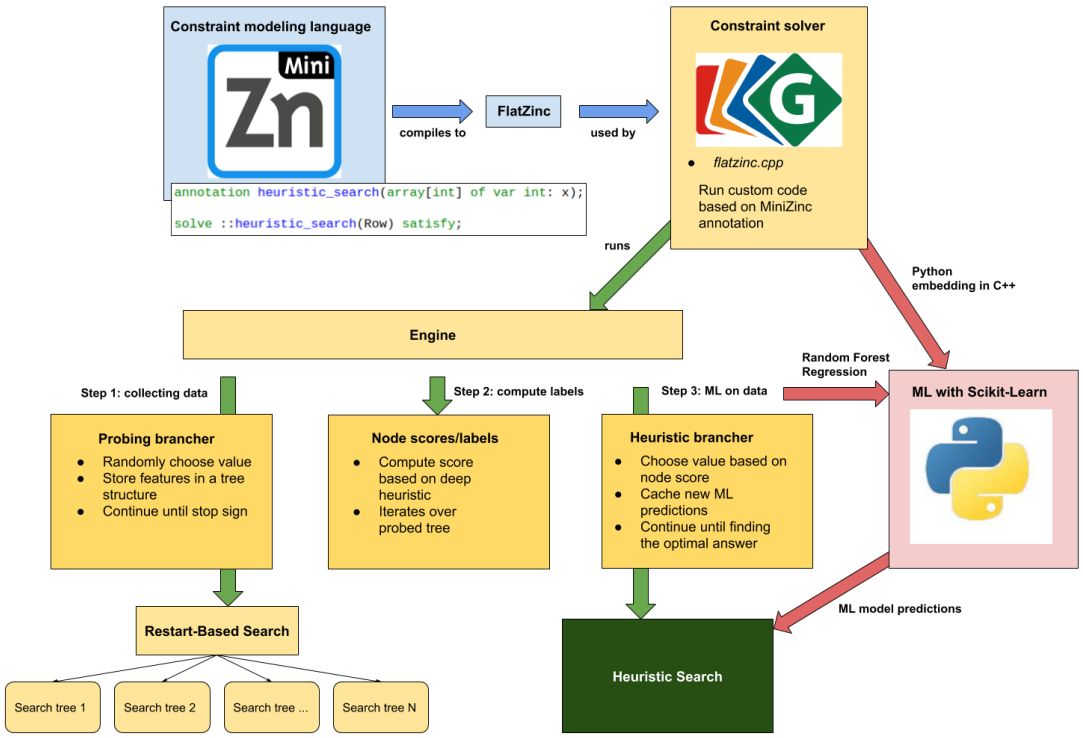


Figure 1: Probing, learning, and heuristic search phases implemented in Gecode.

duces the total size of the search tree and can fix bounds on variables. In our work we use deeper lookaheads, but only during probing since using a lookahead at every node in the search tree is a costly operation. Instead we exploit ML predictions to circumnavigate this cost.

2 Approach

2.1 Preliminaries

We denote a Constraint Satisfaction Problem (CSP) as the tuple (V, D, C), where V is a finite set of decision variables; D is a finite set of domains D_v for each variable $v \in V$, each containing the possible values for v; and C is a finite set of constraints what values each variable $v \in V$ may take. One can also add an objective function to a CSP, turning the CSP into a Constraint Optimization Problem.

Given a CSP, a *variable ordering heuristic* decides which variable to assign a value to first [Rossi *et al.*, 2006]. A typical heuristic is *smallest domain first* [Haralick and Elliott, 1980] (commonly known as *first-fail*) that selects variable $x \in V$ with the smallest domain. On the other hand *anti-first-fail* (AFF) would pick the variable with the largest domain. The *smallest* (SM) heuristic simply chooses the variable with the smallest value in its domain. *Maximum regret* (MR) chooses the variable with the largest difference between the two smallest values in its domain [Loomes and Sugden, 1982].

HSF	Output
$h_{\text{smallest},x}$	lowest value in D_x
$h_{\text{max-regret},x}$	difference between two smallest values in D_x
$h_{\text{anti-ff},x}$	size of domain D_x

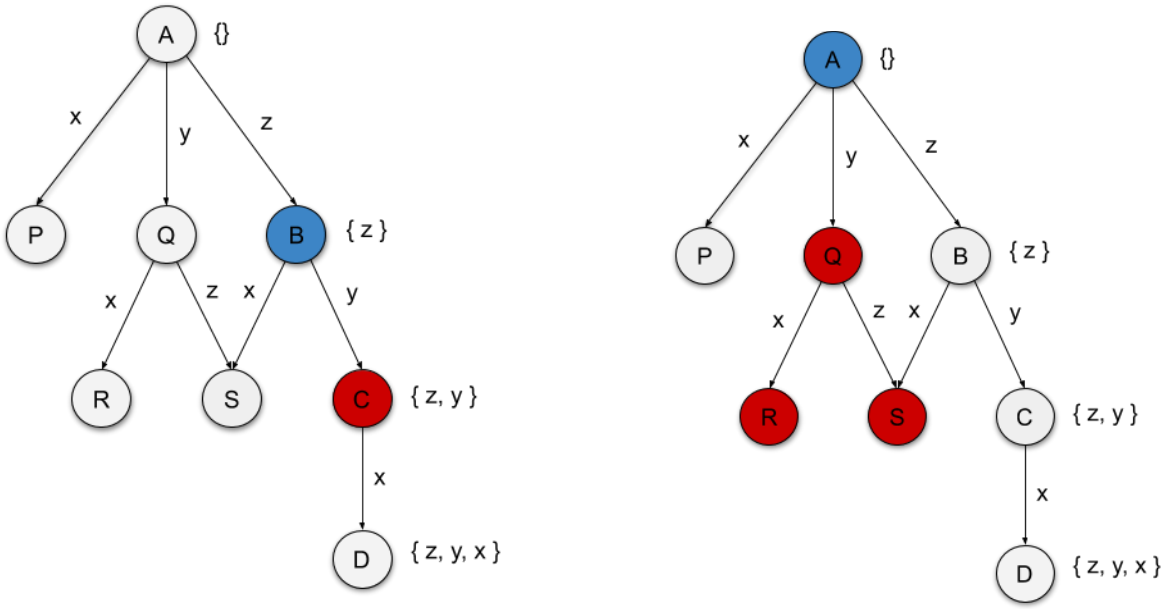
Table 1: Heuristic score functions on a domain D_x

2.2 Deep heuristic score functions

A deep heuristic is built using a *deep heuristic score function* (DHSF) which repeatedly uses a 'classical' heuristic score function (HSF) such as in Table 1. As depicted in Figure 2, for a graph with tree-like shape where nodes represent variables and edges are the next variable selection options, a DHSF looks at multiple levels of the tree, iterating over multiple nodes. For instance, a DHSF based on *smallest* could return a *deep heuristic score* by averaging all the collected heuristic scores, as we explain below.

Given a DHSF, we can select variables based on its outputs. In the example graph of Figure 2, consider node A. We compute a deep heuristic score for each of the variable selection options x, y, and z by inputting the set of features for node A. We then compare the scores and select the variable with the highest or lowest score depending on the kind of heuristic we want to use. For example, if we use the *smallest* heuristic then we use the minimum score of the DHSF. We call the heuristic that selects a variable this way *Deep Smallest*.

Since DHSF can be expensive to compute exactly, we use supervised learning to approximate deep heuristic score func-



(a) Computing a deep heuristic score from node B for the selection of variable y with a depth level of 1. This is the same as using a classical heuristic.

(b) Computing a deep heuristic score from node A for the selection of variable y with a depth level of 2. Nodes R and S would not be considered by most classical heuristics.

Figure 2: Examples of computing a deep heuristic score with a total of three variables. The nodes represent the current variable ordering and the edges a variable selection. A solution is found through the path $\{A, B, C, D\}$.

tions. In our setting historical solving data is not available: we begin with an unseen problem instance and assume that no probing or other search has been done before solving commences. To overcome the lack of training samples, we create a instance-specific dataset online, through a probing phase over the COP. The probing phase acts as a short pre-search to gather features at every search node of the search tree, akin to how Chu and Stuckey [2015] learn value heuristics.

2.3 Using deep heuristics in search

Recall that the three steps of our approach are probing, machine learning and restart-based heuristic search (Figure 1). Probing gathers data by solving the COP using random variable and value orderings. A cutoff bound restarts search after a specified number of failures. A new first variable is selected after each restart, in order to gain maximal data at the top of the search tree [Ortiz-Bayliss *et al.*, 2018].

The learning must operate online, and so should be relatively fast in training and fast in prediction. We considered support vector regression and stochastic gradient descent, and settled on random forest regression. Even with a linear model, the ML latency means it is not tractable to make predictions for every feature combination. In future work, we can cache predictions and re-use them when a feature combination has been seen before. The downside to this is that a value change also leads to a prediction change in the model and a cached prediction would result in a different outcome.

Heuristic search is implemented upon Gecode 6.2.0 using custom *branchers* which make choices on which variable and value to pick next. For each currently-unassigned variable, a heuristic score is predicted through the ML model within this brancher. A variable is then selected depending on the chosen deep heuristic: e.g., the lowest predicted heuristic score for Deep Smallest and Deep Max Regret and the highest predicted score for Deep Anti-First Fail. Since deep heuristics are not used for value selection, we use the minimum or maximum value in the chosen variable's domain, for minimization and maximization problems, respectively.

Lastly, to obtain a wider variety of variable orderings and values we exploit a restart-based process which allows for multiple search jobs within the total search time given. A *job* consists of probing, ML fitting and heuristic search. First we specify a job time. Whenever the search does not finish within the job time then the job is halted and sequentially the next job started. Data gathered by probing is not transferred between jobs, as initial experiments found it greatly increases the time to fit a random forest regression model. Jobs restart sequentially until the overall search time limit is reached.

The framework of Figure 1 is implemented in C++ inside Gecode; a Python script is called from C++ to learn the DHSF, using Scikit-Learn. Gecode with a deepified variable ordering heuristic can be invoked from MiniZinc by e.g., `solve :: heuristic_search(q) minimize;` where q are decision variables.

3 Empirical Results

We conduct an empirical study to investigate how deep heuristics affect total solving time and number of instances solved. We test deep heuristics on four representative prob-

Comparison	Total time no timeouts		Total time with timeouts		Job solution time with timeouts	
	p-value	signif.?	p-value	signif.?	p-value	signif.?
DSM vs SM: RCPSP	0.12	No	0.68	No	$1.02 \cdot 10^{-13}$	Yes
DMR vs MR: RCPSP	0.014	Yes	0.75	No	$6.76 \cdot 10^{-14}$	Yes
DAFF vs AFF: RCPSP	0.45	No	$4.6 \cdot 10^{-4}$	Yes	$2.8 \cdot 10^{-23}$	Yes
DSM vs SM: Evilshop	0.19	No	0.08	No	$4.5 \cdot 10^{-5}$	Yes
DMR vs MR: Evilshop	0.15	No	0.21	No	$2.2 \cdot 10^{-3}$	Yes
DAFF vs AFF: Evilshop	0.13	No	0.028	Yes	$1.2 \cdot 10^{-5}$	Yes
DSM vs SM: Amaze	–	No	0.53	No	0.12	No
DMR vs MR: Amaze	0.45	No	0.004	Yes	0.62	No
DAFF vs AFF: Amaze	–	No	0.018	Yes	$2.8 \cdot 10^{-21}$	Yes
DSM vs SM: OS	0.33	No	$3.8 \cdot 10^{-4}$	Yes	$4.86 \cdot 10^{-9}$	Yes
DMR vs MR: OS	0.016	Yes	0.85	No	0.14	No
DAFF vs AFF: OS	0.14	No	$1.7 \cdot 10^{-4}$	Yes	$1.2 \cdot 10^{-8}$	Yes

Table 2: T-tests of average (arithmetic mean) total runtime, without and with timeouts, and average job solution runtime with timeouts. p-value with $\alpha = 0.05$. '–' denotes p-value cannot be computed because of too few instances.

Heuristic	RCPSP	Evilshop	Amaze	Stacks
Gecode Smallest	29.7%	54.6%	38.5%	23.3%
Deep Smallest	31.2%	18.2%	23.1%	54.3%
Gecode AFF	15.9%	18.2%	18.2%	23.3%
Deep AFF	35.4%	63.6%	63.6%	56.6%
Gecode MR	37.7%	18.2%	76.9%	57.4%
Deep MR	41.6%	48.5%	28.2%	58.1%

Table 3: Percentage of total instances solved by heuristics

lem classes from the MiniZinc benchmarks: Resource Constrained Project Scheduling Problem (RCPSP), Evilshop, Amaze, and Open Stacks; 138, 11, 13, and 43 instances.

All the instances are run for a maximum time of 4 hours. We set the job time to be 15 minutes for the deep heuristics, allowing at most 16 jobs in total. We compare the solvers with and without instances that time out. For all problems we select a depth value of 25.

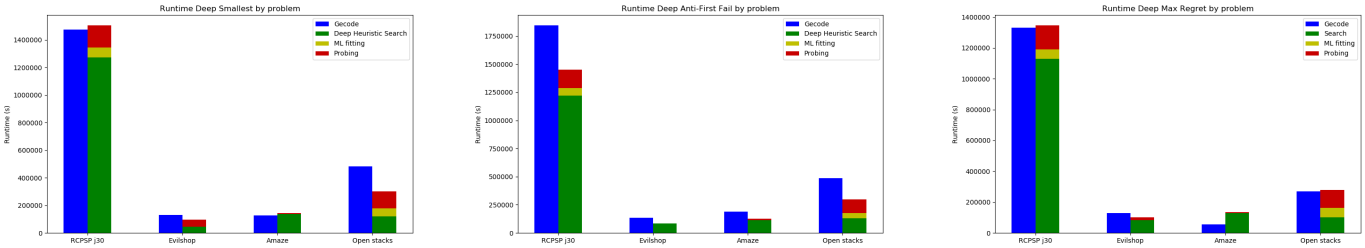
Selecting how long we should probe can be very dependent on the problem: as the complexity of the problem changes, for instance multiple decision variables or more constraints per variable, it may take a variable amount of time to collect data. For RCPSP, Evilshop, and Amaze we set the probing time to 1 million nodes and for Open Stacks 2 minutes as the collection of 1 million nodes of information takes a very long time. Probing time for RCPSP, Evilshop, and Amaze is usually within the minute.

First, Figure 3 shows the total runtime of the SM, MR, and AFF heuristics versus the deep versions, DSM, DMR, and DAFF. Averaged over all instances, DSM uses 7.7% less total runtime than SM, DAFF 26.1% less than AFF, while DMR uses 4.4% more than MR. Figure 4 compares the classical heuristics with each other and the deep heuristics with each other by showing their average runtime. We observe that AFF performs worse than the other heuristics, and also that AFF mostly timed-out. We also observe that MR on average outperforms the other heuristics for each problem where as DMR does not. Figure 5 and 6 and Table 2 compare the average runtime. It is evident that AFF runs worse that DAFF. Figure 5 includes timeouts, which means that we do not know for how long these instances would have run given unlimited runtime. Hence Figure 6 shows the same comparison without timed-out instances. On most problems deep heuristics are outperformed in average runtime. Thus we can reason that classical heuristics perform better than the deep heuristics when they solve instances within 4 hours. Figure 6b misses two bars for the Amaze problem because there are no instances where neither AFF nor DAFF timeouts.

Second, we examine the the percentage of instances solved per problem, in Table 3. The deep heuristics outperform their classical counterparts as they are able to solve more problems within 4 hours. Only in the Amaze problem DSM and DMR and Evilshop DSM is outperformed. Drilling down, Figure 8 shows the number of instances in which heuristics outperform their counterpart. Notable is that DSM and DAFF outperform SM and AFF in more instances, while many RCPSP problems cannot be solved within the time limit by either heuristic. Overall MR works better than SM and AFF on RCPSP and Open Stacks. If a deep heuristic search completes within 4 hours then one of the search jobs in the framework succeeded. We denote these successes as 'solutions'. The solutions are independent of other search jobs and hence we compare their runtimes in Figure 7. It can be said that the solutions have significantly less runtime than the classical heuristics which can be partly explained by the fact that search jobs have a maximum runtime of 15 minutes.

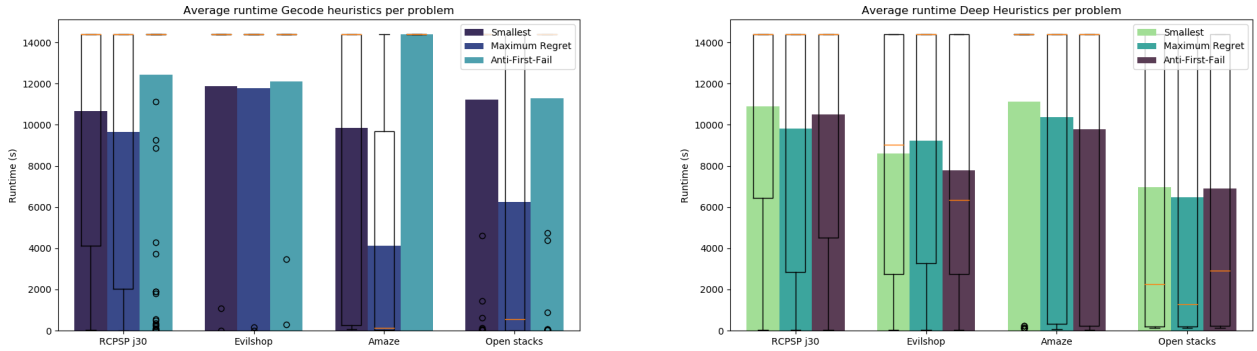
4 Discussion and Future Work

This paper addressed the problem of one-shot learning of search heuristics for constraint optimisation problems. We proposed to learn extended versions of existing variable ordering heuristics, through a deepification process. The learning uses a probing phase to gather data coupled with a fast regression approach. We demonstrated deep heuristics based



(a) SM versus DSM (b) AFF versus DAFF (c) MR versus DMR

Figure 3: Total runtime divided into probing, ML, and search



(a) Gecode heuristics (b) Deep heuristics

Figure 4: Comparison of average runtime between heuristics

on three different heuristics: smallest, anti first-fail, and maximum regret. Compared to the classical 'shallow' heuristics, we find that deep heuristics solve 20% more problem instances across a subset of the MiniZinc benchmarks, while improving on overall runtime for two problem classes.

Further experiments are warranted to assess the contribution of each the parts of our approach. In particular, recognising the stochasticity inherent in a learning-based approach, we use restarts with the deep heuristics – but not with their shallow counterparts.

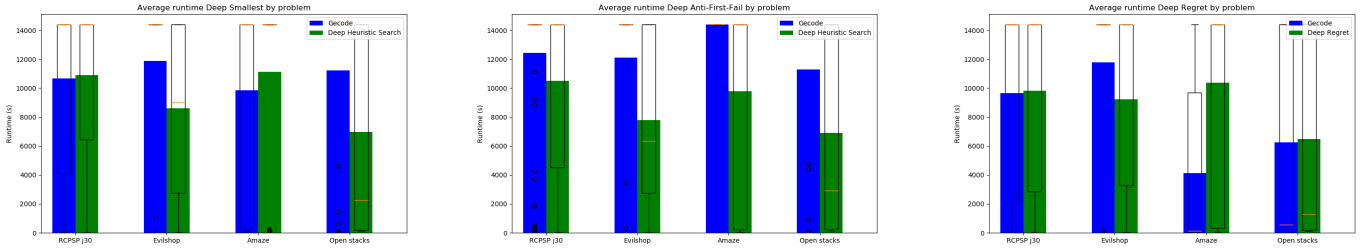
The result that, overall, deep heuristics solve more instances – albeit with increased average runtime on some problem–heuristic combinations and reduced on other combinations – suggests development of the approach in a number of possible directions. First, our framework is implemented in Gecode. To deepify heuristics such as *domwdeg*, Gecode needs additional instrumentation: during probing we cannot record, e.g., search node successes, failures, and added or removed constraints. Further engineering is needed also to parallelise the probing, which would improve performance, since Gecode cannot handle our simultaneous node addition to the same search tree.

Second, selecting meta-parameters for the deepification process. If the problem class is known, the meta-parameters could be tuned after selecting the right heuristic for the problem; possibly, ML can learn meta-parameters settings given meta-data about the COP instance. Third, we explored the use of random search in probing. The use of specific probing heuristic(s) is interesting. Fourth, use of the score from the DHSF might be improved by adding exploration and exploitation attributes. For instance, a discount factor would give more weight to earlier choices adhering to the principle of making good early decisions [Ortiz-Bayliss *et al.*, 2018].

This paper showed how to deepify variable ordering heuristics, and our approach can be readily applied to value ordering. Further, the learned deep heuristic function can learn both variable and value orderings (compare Cox *et al.* [2019]). Another interesting direction is extending the restart-based search mechanism. Currently, new data is gathered at every restart without saving data from the previous search jobs. One could allow the total dataset from probing to grow over time. The challenge is the volume of data gathered and efficiently learning over it; stochastic gradient descent may be an interesting option.

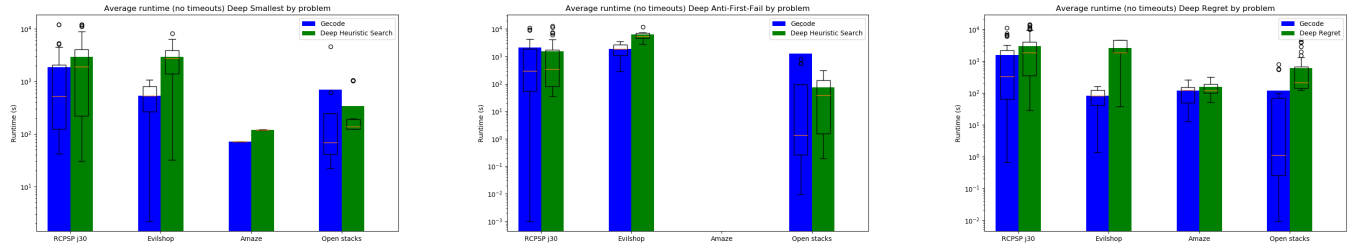
Acknowledgements

Thanks to the reviewers for their comments. We thank C. Schulte, who unfortunately passed away last year, and G. Tack for assistance with Gecode and MiniZinc. This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant number 952215.



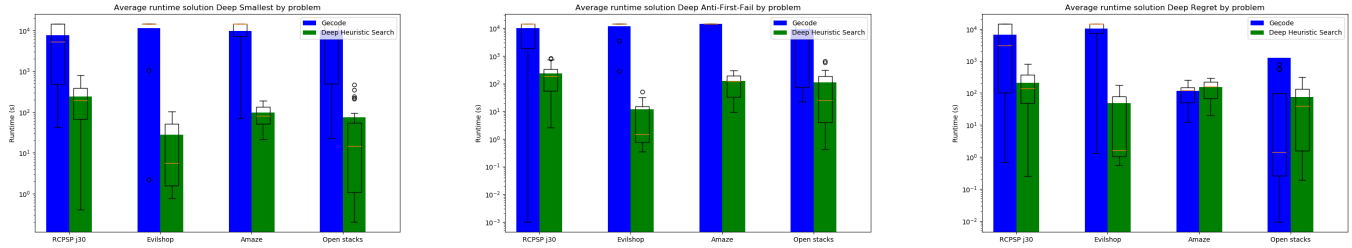
(a) SM versus DSM (b) AFF versus DAFF (c) MR versus DMR

Figure 5: Comparison of average runtime including timed-out instances



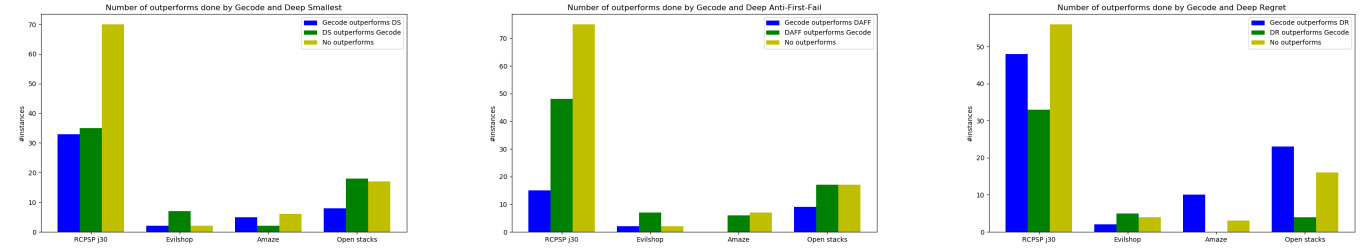
(a) SM versus DSM (b) AFF versus DAFF (c) MR versus DMR

Figure 6: Comparison of average runtime without timed-out instances



(a) SM versus DSM (b) AFF versus DAFF (c) MR versus DMR

Figure 7: Comparison of average runtime: deep heuristic solutions



(a) SM versus DSM (b) AFF versus DAFF (c) MR versus DMR

Figure 8: Number of times that heuristics outperform each other

References

Fawaz Alanazi and Per Kristian Lehre. Limits to learning in reinforcement learning hyper-heuristics. In *Proceedings of 16th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP'16)*, pages 170–185, 2016.

Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 146–150, 2004.

Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. SeaPearl: A constraint programming solver guided by reinforcement learning. *CoRR*, abs/2102.09193, 2021.

Geoffrey Chu and Peter J. Stuckey. Learning value heuristics for constraint programming. In *Proceedings of the 12th International Conference on the Integration of AI and OR Techniques in Constraint Programming (CPAIOR'15)*, page 108–123, 2015.

James L. Cox, Stephen Lucci, and Tayfun Pay. Effects of dynamic variable–value ordering heuristics on the search space of sudoku modeled as a constraint satisfaction problem. *Inteligencia Artificial*, 22(63):1–15, 2019.

Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, pages 179–193, 1996.

Wasu Glankwamdee and Jeff Linderoth. Lookahead branching for mixed integer programming. Technical report, Lehigh University, October 2006.

Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263—313, 1980.

Elias B. Khalil, Bistra Dilkina, George L. Nemhauser, Shabbir Ahmed, and Yufen Shao. Learning to run heuristics in tree search. In *Proceedings of 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, pages 659—666, 2017.

Graham Loomes and Robert Sugden. Regret theory: An alternative theory of rational choice under uncertainty. *The Economic Journal*, 92(368), 1982.

Jayanta Mandi, Emir Demirovic, Peter J. Stuckey, and Tias Guns. Smart predict-and-optimize for hard combinatorial optimization problems. In *Proceedings of 34th AAAI Conference on Artificial Intelligence (AAAI'20)*, pages 1603–1610, 2020.

José Carlos Ortiz-Bayliss, Iván Amaya, Santiago Enrique Conant-Pablos, and Hugo Terashima-Marín. Exploring the impact of early decisions in variable ordering for constraint satisfaction problems. *Computational Intelligence and Neuroscience*, 2018:6103726:1–6103726:14, 2018.

Philippe Refalo. Impact-based search strategies for constraint programming. In *Proceedings of the 10th International Conference on the Principles and Practice of Constraint Programming (CP'04)*, pages 557–571, 2004.

Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode 6.2.0.* 2019.

Richard J. Wallace. Determining the principles underlying performance variation in CSP heuristics. *International Journal of Artificial Intelligence Tools*, 17(5):857–880, 2008.

Wei Xia and Roland H. C. Yap. Learning robust search strategies using a bandit-based approach. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, pages 6657–6665, 2018.