Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-022-09816-z

Online learning of variable ordering heuristics for constraint optimisation problems

Floris Doolaard[1] · **Neil Yorke-Smith**[1]

Accepted: 30 August 2022

Abstract
Solvers for constraint optimisation problems exploit variable and value ordering heuristics. Numerous expert-designed heuristics exist, while recent research learns novel, customised heuristics from past problem instances. This article addresses unseen problems for which no historical data is available. We propose one-shot learning of customised, problem instance-specific heuristics. To do so, we introduce the concept of *deep heuristics*, a data-driven approach to learn extended versions of a given variable ordering heuristic online. First, for a problem instance, an initial online 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 lookahead of classical heuristics. A restart-based search strategy allows for multiple learned models to be acquired and exploited in the solver's optimisation. We demonstrate deep variable ordering heuristics based on the smallest, anti first-fail, and maximum regret heuristics. Results on instances from the MiniZinc benchmark suite show that deep heuristics solve 20% more problem instances while improving on overall runtime for the Open Stacks and Evilshop benchmark problems.

Keywords Variable ordering · Machine learning · Constraint optimisation problem · Gecode · Random forest regression

Mathematics Subject Classification (2010) 90C27 · 68T05

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 [1]. Various variable ordering heuristics have been designed by human experts [2–4]. Recent work also acquires dedicated heuristics using machine learning (ML), or learns which of a given set of heuristic to use [5–8]. However, both classical and learned heuristics are based on the current search node.

✉ Neil Yorke-Smith
n.yorke-smith@tudelft.nl

Floris Doolaard
f.p.doolaard@student.tudelft.nl

[1] STAR Lab, Delft University of Technology, Postbus 5, Delft 2600 AA, Zuid Holland, Netherlands

Published online: 05 October 2022

Further, some ML methods may require significant offline training time before starting search, while others face the familiar ML difficulty of generalising to unseen instances.

This article addresses the situation of online solving of unseen optimisation problems for which no historical data is available for learning. 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 generalising better than classical variable ordering heuristics.

We demonstrate deep heuristics derived from three representative variable ordering heuristics: smallest, anti first-fail, and maximum regret. On the MiniZinc benchmark suite, 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.

In more detail, we implement deep heuristics in the open source Gecode solver [9]. Inspired by Chu and Stuckey [10], given a problem instance, an initial probing phase employs pseudo-random search to gather a variety of variable–value assignments. This data, including features and labels, is then utilised 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 promising solutions.

The article proceed as follows. After brief preliminaries in Sections 2, and 3 introduces and formalises the concept of deep heuristics. Section 4 provides an experimental study on four classes of COPs. Section 5 positions our contribution in the literature. Section 6 concludes and points out future directions.

2 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 containing the possible values for v; and C is a finite set of constraints on what values each variable $v \in V$ may take. One can also add an objective function to a CSP which is maximised or minimised. This turns the CSP into a Constraint Optimisation Problem (COP).

Given a CSP or COP, a *variable ordering heuristic* decides which variable to assign a value to first [11]. A typical heuristic is *first-fail* [2] that selects variable $x \in V$ which is most likely to fail, meaning that by assigning a value to x first the constraints cannot be satisfied. Formally, this is the same as picking the variable with the smallest remaining domain. On the other hand, *anti-first-fail* (AFF) selects the variable with the largest domain, least likely to fail. The *smallest* (SM) heuristic simply chooses the variable with the smallest value in its domain. *Maximum regret* chooses the variable with the largest difference between the two smallest values in its domain.

Many other human-designed heuristics exist for both variable and value ordering, and more recent efforts acquire novel heuristics by machine learning, as discussed in Section 5.

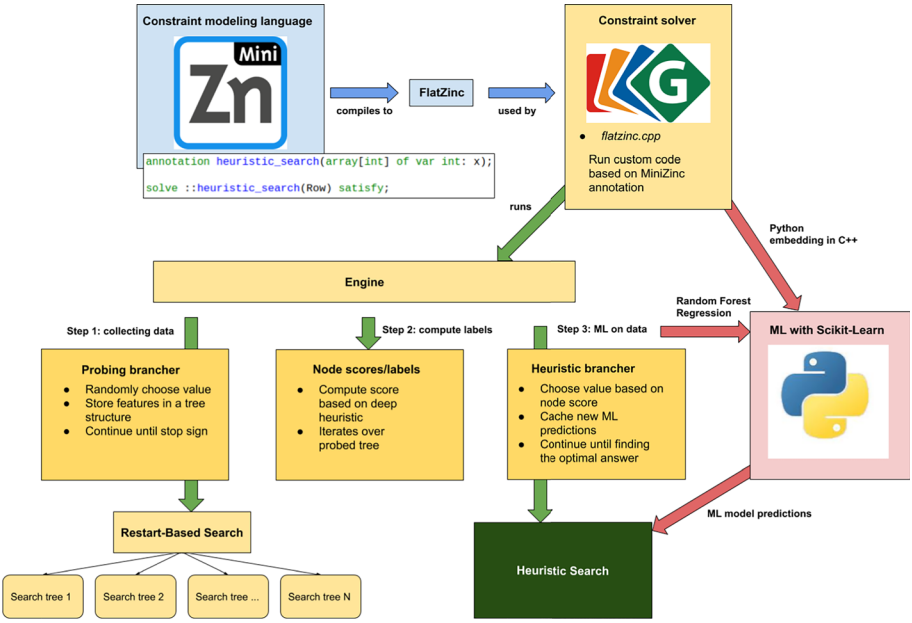


Fig. 1 Probing, learning, and heuristic search phases implemented in Gecod

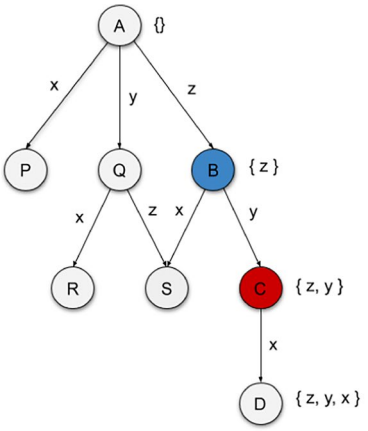
3 Deepification: learning deep heuristics

This section explains how we create deep variable ordering heuristics by defining deep heuristic score functions. We approximate these score functions in an online setting with a supervised ML model, which is used during restart-based search in Gecode. Figure 1 summarises the proposed learning-and-solving framework.

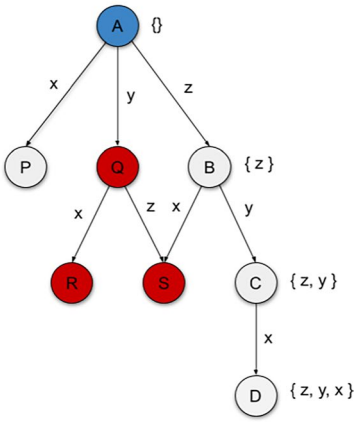
3.1 Deep heuristic score functions

A deep heuristic is built using a *deep heuristic score function* (DHSF) which repeatedly uses a *classical* heuristic score function. That is, 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.

Intuitively, a shallow score looks only at the tip of the iceberg: the idea is that a shallow score is computed from a fast-to-compute and (relatively) easy-to-understand decision. By contrast, a deep score aims to understand more and make a more judicious decision. It looks further into the sub-tree, trying to reveal whether a decision was indeed better than others. For example, suppose the average sub-tree of decision A over the average sub-tree of decision B yields a better score, but decision B had better shallow score: then a myopic shallow decision here may have hidden the more interesting branch.



(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.

Fig. 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 $\{A,B,C,D\}$

Definition 1 A *heuristic score function* h takes as input the features of a search node based on an arbitrary heuristic ϕ, and outputs a *heuristic score*, based on the objective (i.e., minimise or maximise) of heuristic ϕ.

For example, the heuristic score function for the *smallest* heuristic has as input an unassigned variable, and outputs the lowest value in the domain as the heuristic score. Other features of a search node that a HSF might use are, for example, the sum of the domain sizes of all variables, the value assigned to a variable, or the domain size of a variable.

We take a heuristic score function as one of the inputs to a *deep* heuristic score function:

Definition 2 A *deep heuristic score function* H takes as input a depth parameter d and a heuristic score function h, and outputs a *deep heuristic score* based on the heuristic score function h over a graph fragment.

For example, a DHSF based on *smallest* could return a deep heuristic score by averaging all the collected heuristic scores, as we explain below. Note that a DHSF does not provide a predicted score, rather it is used below as a label to learn a deep heuristic.

Figure 2 provides an illustration of computing a DHSF given a depth value d, for different nodes in a graph with a tree-like shape. The nodes coloured blue represent the current variable under consideration in the search. The edges represent the selection of a currently-unassigned variable from that node. As seen, while a DHSF may give substantial information about many nodes compared to a heuristic score for a single

Table 1 Heuristic score functions

Heuristic score fn.	Output	Objective
$h_{\text{smallest}}(x)$	lowest value in D_x	minimisation
$h_{\text{max-regret}}(x)$	difference between the two smallest values in D_x	maximisation
$h_{\text{anti-ff}}(x)$	size of domain D_x	maximisation

node, computing multiple heuristic scores is a costly operation during search. Hence in the sequel we acquire a learned model to *approximate* a given DHSF and predicts the DHSF's scores based on a set of features.

Given a DHSF – whether the exact function or a learned approximation of it – we can select variables based on its outputs. In the example graph of Fig. 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*.

3.1.1 Three example deep variable ordering heuristics

We demonstrate how to deepify *smallest*, *maximum regret*, and *anti-first-fail* heuristics. The choice to deepify here anti-first-fail (AFF) rather than first-fail is arbitrary. The ambition is to present the potential of the deepification concept through the diversity of these three heuristics. Section 6 discusses deepifying other heuristics.

Specifically, we consider the implementation of these three heuristics in Gecode [9]. We deepify the heuristics given the features that can be retrieved from Gecode. For each of the heuristics, we define the corresponding heuristic score function h (Table 1). The DHSFs are defined recursively as follows:

$$g(x, v, k) = \begin{cases} h_r(x) + \sum_{y \in M} g(y, v', k+1) & \text{if } k < d \\ 0 & \text{if } k = d \end{cases} \tag{1}$$

$$H(x, v, k) = \frac{g(x, v, k)}{nodes - counted} \tag{2}$$

where x is an available variable that is selected in the current search node (i.e., one of the currently unassigned variables); v the value to be assigned for the selected variable x, i.e., $x \leftarrow v$; k the current level in the tree; M the set of variables of children of the current node; $h_r(x)$ the heuristic score for heuristic r by selecting variable x; *nodes-counted* the total nodes for which a heuristic score is computed; and d the depth for which we compute the heuristic in the number of levels of a tree.

For the assignment of values we simply choose the lowest value in the domain for minimisation problems and the highest value for maximisation problems. While value selection heuristics can be used for better performance, in order to isolate the effect of variable ordering heuristics in this article we consistently use the `indomain_min` MiniZinc

annotation (for minimisation problems) for both the Gecode heuristics and their deep versions. Dynamic variable and value ordering are again discussed in Section 6.

3.2 Approximating deep heuristic scores

Since deep heuristic score functions are too expensive to compute at solving time, and moreover require information ahead in the search tree which might be yet unknown from a given search node, we use online supervised learning to approximate DHSFs.

Training samples are needed to fit a ML model, where X_f are the features, y_{score} the labels (scores), and ψ the approximate deep heuristic score function:

$$\psi(X_f) = y_{score} \tag{3}$$

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 an 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 [10] learn value heuristics. The probing creates a *probe tree*, which in our current implementation is then discarded once the information is extracted from it. An idea for the future would be to re-use part of the probe tree in the main solving phase.

The probing phase provides data on different variable assignments and their corresponding label based on the chosen heuristic score function. To provide a maximally-diverse set of assignments, we employ a random variable order and random value assignment during probing. Regardless of which DHSF is to be learned, the following features are saved at each search node: variable domain size D_x, sum of the domain sizes of all variables D_V, assigned value v, value position in the domain v_p, minimum and maximum values in the domain v_{min} and v_{max}, maximum regret as the difference between the two smallest value in the domain r_{min}, maximum regret as the difference between the two largest value in the domain r_{max}.

These features are possible to collect during probing, and also during the actual search. This is important because predictions have to be made based on the features gathered during probing. Features such as 'number of siblings/children' tell us about the structure of the search tree but are not known during search. For example, visiting a node for the first time means the node does not have any children nodes yet, thus making that feature unreliable.

At the end of the probing phase, with the feature data acquired, we compute the labels y_{score}. We thus compute the deep heuristic score for each node in the search tree that was probed, discarding nodes in the probe tree that have children fewer than d levels deep.

Figure 3 gives an example of the computation of labels for a depth value of 2. Let there be a heuristic score s_i for each variable i. The only nodes from which deep heuristic scores can be computed are the green nodes, respecting the depth value. From node A, selecting variable y would yield a deep heuristic score of $(s_y + s_x + s_z)/\textit{nodes-counted} = (2 + 1 + 1)/3 \approx 1.33$ by (2). From node A, selecting variable z, would yield a deep heuristic score of $(s_z + s_x + s_y)/\textit{nodes-counted} = (3 + 1 + 2)/3 = 2$. Finally, from node B, selecting variable y, would yield a deep heuristic score of $(s_y + s_x)/\textit{nodes-counted} = (2 + 1)/2 = 1.5$.

Algorithm 1 shows the naive approach to compute the labels based on the recursive function (2). It could be improved by using a dynamic programming algorithm, iterating from leaves to root and memorising computed values.

Algorithm 1 Computing labels y_{score}.

```
d ← depth value for the heuristic; N ← nodes from the probed tree; h ← height of
the probed tree; s_n ← heuristic score or label for node n;
/* Call the recursive function for each node in the tree
   */
foreach node n in N do
    k ← level in tree of node n;
    s_n ← 0;
    if d ≤ h − k then
        s_n ← COMPUTESCORE(n, d, 0) / nodes-counted;

/* Recursive function with node n, depth d and current
   depth d_c                                                    */
COMPUTESCORE(n, d, d_c)
s_c ← 0;
if d = d_c then
    return s_n
foreach child c of node n do
    s_c ← s_c + COMPUTESCORE(c, d, d_c + 1)
return s_n + s_c;
```

Putting the pieces together, the intuition is summarised as follows: we gather data from a probe tree and use that data to acquire a learned model, which approximates a DHSF. During the main solving phase, if we had the entire search tree in front of us, we would look not only at the current node being explored, but also at its children up to a specified depth. However at solving time, since we do not have the entire tree – indeed, the search is only constructing it incrementally – we exploit whatever information we have at the current node, and also an estimate for the information in its future children, based on what was learned from the probe tree. Next we explain how this main search proceeds.

3.3 Using deep heuristics in search

Recall that the three steps of our approach are probing, machine learning and restart-based heuristic search (Fig. 1). We give implementation details of each in turn and explain the search, the final phase.

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 [12].

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. To save runtime, we can cache predictions and re-use them when a variable ordering 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.

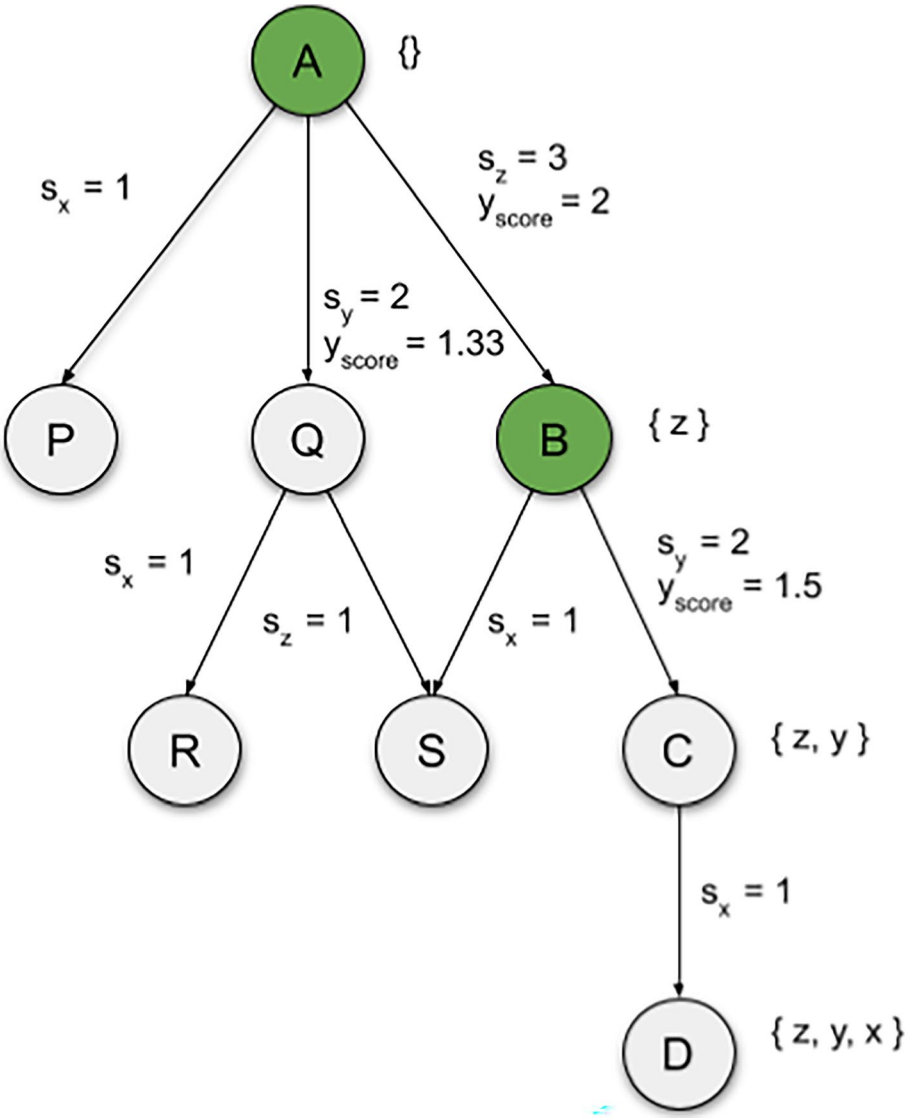


Fig. 3 Computing labels y_{score} in the green nodes for variable choices at least 2 levels deep

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*. As explained earlier, since deep heuristics are not used for value selection, we use the minimum or maximum value in the chosen variable's domain, for minimisation and maximisation problems, respectively.

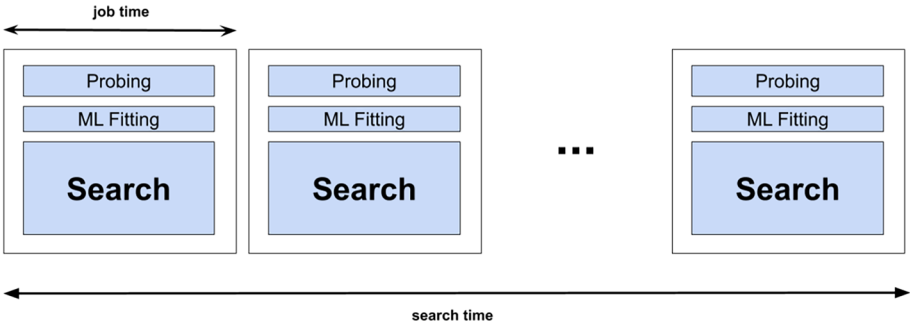


Fig. 4 Restart-based search: independent jobs with a job time repeat until search time is reached

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. The intuition behind the idea is to find a successful deep heuristic search out of multiple jobs, leveraging the stochastic nature of the deepification. Figure 4 shows an overview of the restart-based search.

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 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. Note that within each search job, default Gecode search is used (other than the custom variable ordering). In particular, Gecode's restart-based search ('rbs') is not used. In Section 4 when comparing with classical shallow heuristics, we likewise use Gecode's default search for the latter, and not its 'rbs'.

One could argue to perform longer probing instead of multiple shorter probing phases. However, besides causing a longer fitting time this increases the chance that the solver may be stuck for a long time on a variable ordering. In this case, the solver tries to backtrack, assigning different values to the variables, but fails frequently. However, by restarting search completely we get a different predicted variable ordering due to the pseudo-random probing.

3.3.1 Integration

The framework of Fig. 1 is implemented in C++ inside Gecode, which is invoked from MiniZinc. The deep heuristics can then be used with this MiniZinc annotation:

```
solve :: heuristic_search(q) minimize;
```

where q is a list of decision variables in a minimisation problem. The probing is implemented in C++ in Gecode. To learn the approximate DHSF, from within Gecode we call a Python script which uses the Scikit-Learn library 0.23 to train the ML model. Lastly, as noted, the restart-based search is implemented with custom Gecode branchers.

The source code is available at: https://doi.org/10.4121/17081021.[1]

[1] DOI will be made active by data.4tu.nl after paper acceptance.

4 Experimental study

We conduct an empirical study to investigate how deep heuristics affect total solving time and number of instances solved. The focus is the performance of the Gecode 'shallow' variable ordering heuristics versus their deep versions. Further, we examine varying the probing time limit, the job time limit, and the deepification depth.

4.1 Problem instances

From the MiniZinc benchmarks repository [13] we test deep heuristics on four representative problem classes: Resource Constrained Project Scheduling Problem (RCPSP), Evilshop, Amaze, and Open Stacks; respectively, 138, 11, 13, and 43 instances. These four problem classes are selected based on a number of factors.

First, they provide many instances relative to other problem classes in the repository for us to experiment with. Second, some of the alternative problem classes contain boolean or float decision variables and we have not yet implemented deep heuristics for these variable types; the choice was limited to integer problems. Third, even though Evilshop has much resemblance to the RCPSP problem we selected it due to its different features: it leaves the capacity of resources as a variable, and its tasks use more than half of the possible maximum capacity. Thus, we could observe whether or not there was a significant change of results due to sequential search or a large change in domain values and size. (We observed none of those changes.) From the classes we select problem instances that run for at least a minute in Gecode with the *smallest* variable ordering heuristic, to exclude overly-simple instances.

RCPSP The RCPSP[2] is an NP-hard problem [14] which consists of J activities each having a process time p_j for $j \in J$. Once an activity is started it cannot be stopped and due to technological requirements there is a precedence relation between activities. An activity requires an amount of resources, for example a machine or vehicle, before it can start. These resources are renewable as their full capacity becomes available again after a period [15]. The objective is to find and optimal schedule with respect to the earliest end time of the schedule. The tasks' resource requirements may not exceed the resource capacities and each precedence relation must be met.

Specific to Minizinc and CP, this problem is modelled with a constant discrete capacity over time and tasks with a constant discrete duration and resource requirements.

The solver will use the starting times of the activities as decision variables.

Evilshop The Evilshop[3] problem is a variant of the classic job-shop scheduling problem where the capacity of resources has been left as a variable and tasks use more than half of the possible maximum capacity. Besides that, start times are scaled up.

The solver will use the starting times of the jobs as decision variables.

[2] https://github.com/MiniZinc/minizinc-benchmarks/blob/master/rcpsp/rcpsp.mzn
[3] https://github.com/MiniZinc/minizinc-benchmarks/blob/master/evilshop/evilshop.mzn

Amaze Amaze[4] is a game in which we have been given a grid containing pairs of natural numbers where the goal is to connect the pairs: 1 to 1, 2 to 2, etc. Lines can only be drawn horizontally or vertically and they may never cross.

The solver will use each cell of the grid as a decision variable to insert natural numbers.

Open Stacks In the Open Stacks[5] problem, also called Minimising the Maximum Number of Open Stacks (MOSP), there are customers who can order products. Only one product at a time can be manufactured in batches. The customers can order multiple products and a stack to save the products for the customers is opened once manufacturing for one of their products has begun. Once all ordered products for a customer are manufactured, the products can be sent and the stack is freed. The aim is to determine the sequence of manufacturing products such that we minimise the maximum amount of open stacks that are needed which is the maximum number of customers whose products are being manufactured simultaneously [16].

The solver uses the schedule of manufacturing products as decision variables.

4.2 Parameter selection

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. The number of decision variables of the instances is at least 30, which means that $d = 25$ ensures we gather data for at least the first 5 levels of the search tree.

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 limit to 1 million nodes and for Open Stacks to 2 minutes, as the collection of 1 million nodes of information on Open Stacks takes an excessive time. Probing time for RCPSP, Evilshop, and Amaze is usually within 1 minute. The last column of Table 4 shows the average dataset size that is used after probing has finished.

4.3 Evaluation metrics

We use three metrics to assess quality of deep heuristics. The first is to compare *total runtime* versus Gecode's standard variable ordering heuristics. Note that a comparison in total number of search nodes cannot be made due to a limitation in Gecode: when a job is cancelled in the restart-based search, Gecode does not show the statistics with the number of nodes. Second, we record the *number of instances that are solved* by each heuristic.

The third metric is the *quality of the fitted ML model*. We seek to measure this with the coefficient of determination R^2. Further, since the order of the predicted values for selecting variables is more important than predicting the exact true value, we use Spearman's rank correlation metric r_s to see the relationship between the predicted data set and true test set. Thirdly, we use the true rankings of predicted values. R^2 and Spearman's rank

[4] https://github.com/MiniZinc/minizinc-benchmarks/blob/master/amaze/amaze.mzn

[5] https://github.com/MiniZinc/minizinc-benchmarks/blob/master/open_stacks/open_stacks_01.mzn

correlation are also used to evaluate the top 10 predicted values (rather than all predicted values) to get a better idea of the quality of early choices. These are denoted as R^2@10 and Spearman@10. These metrics are used on a withheld 20% of the dataset to prevent bias; the other 80% is used for training.

Since our restart-based search is sequential and does not carry over any information between searches, the final results reported below in each case come from the deep heuristic search in one search job: the last job.

4.4 Evaluating deepified heuristics

First, Fig. 5 shows the total runtime of the *smallest* (SM), *maximum regret* (MR), and *anti-first-fail* (AFF) heuristics versus the deep versions, *deep smallest* (DS), *deep maximum regret* (DR), and *deep anti-first-fail* (DAFF). Averaged over all instances, DS uses 7.7% less total runtime than SM, DAFF 26.1% less than AFF, while DR uses 4.4% more than MR.

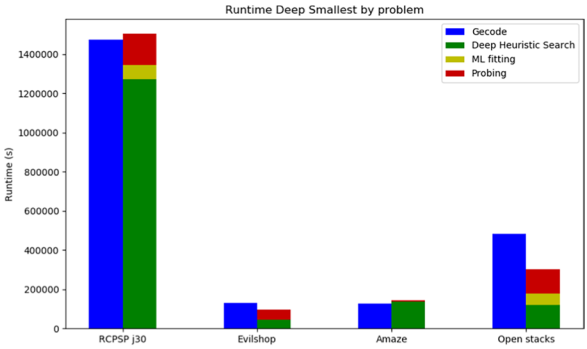
Figure 6 compares the classical heuristics with each other and the deep heuristics with each other by showing their average runtime. We observe that AFF performs worse than the other heuristics, and also that AFF mostly timed-out. We also observe that MR on average outperforms the other heuristics for each problem where as DR does not. Below we compare the deep heuristics with each other in detail.

Figures 7 and 8 and Table 2 compare the average runtime. It is evident that AFF performs worse than DAFF. Figure 7 includes timeouts, which means that we do not know for how long these instances would have run given unlimited runtime. These runs are assigned a time of twice the timeout, i.e., 8 hours. Figure 8 shows the same comparison without timed-out instances. On most problems deep heuristics are outperformed in average runtime. Figure 8b misses two bars for the Amaze problem because there are no instances where neither AFF nor DAFF time outs.

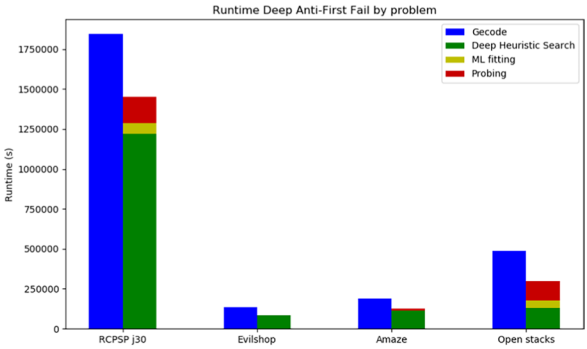
Second, we examine the the percentage of instances solved per problem, in Table 3. The deep heuristics outperform their classical counterparts as they are able to solve more problems within 4 hours. Only in the Amaze problem deep smallest (DS) and deep regret (DR), and Evilshop DS is outperformed. Drilling down, Fig. 9 shows the number of instances in which heuristics outperform their counterpart. Notable is that DS 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 Fig. 10. It can be said that the solutions have significantly less runtime than the classical heuristics. This can be partly explained by the fact that search jobs have a maximum runtime of 15 minutes: the searches that timed out without finding a solution only had jobs that ran for more than 15 minutes. Recall that within each search job, restarting search is not employed; likewise for the runs with shallow heuristics.

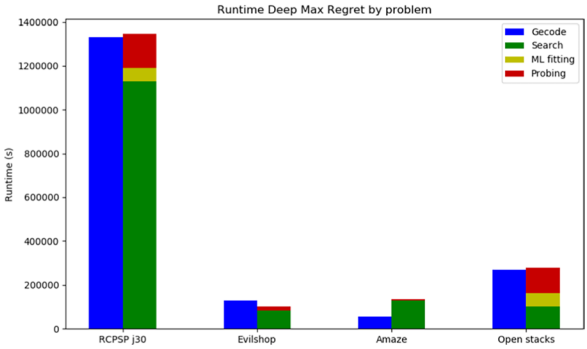
Third, Table 4 shows the average quality of the fitted model for deep heuristics. Note that the values 'err' denote values that have not been recorded due to the dataset being likely too small. Rankings for each dataset were computed by looking at what variable the heuristics would select first and giving it rank 1, etc. Then we compute the *predicted* selection of variables for the corresponding deep heuristic and rank its order of



(a) smallest (SM) versus deep smallest (DS)

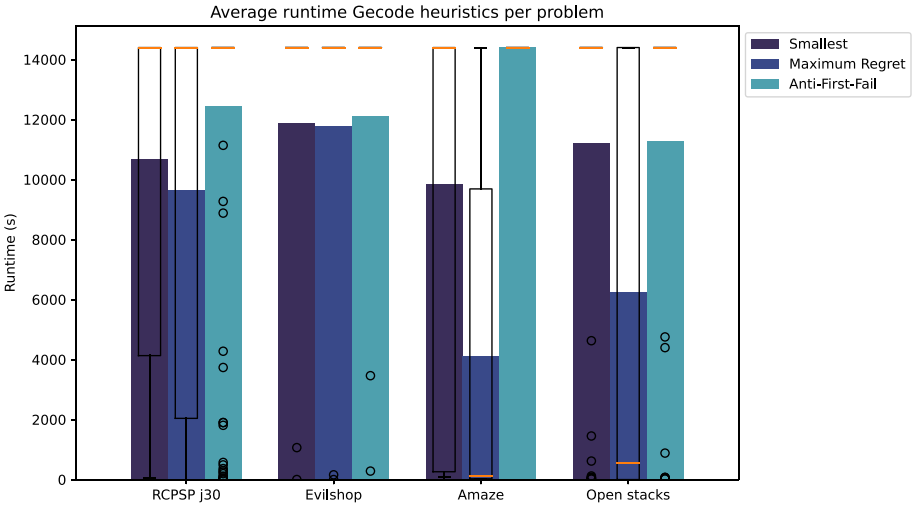


(b) anti-first-fail (AFF) versus deep anti-first-fail (DAFF)

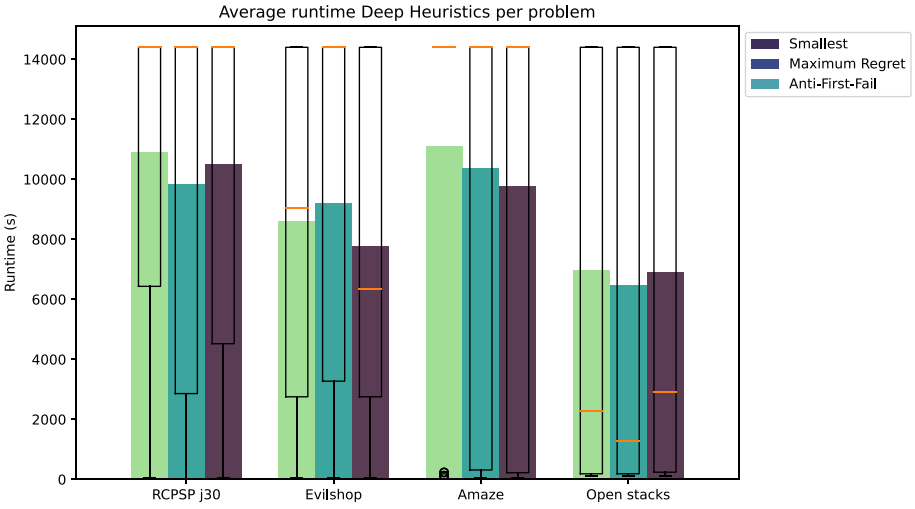


(c) maximum regret (MR) versus deep maximum regret (DR)

Fig. 5 Total runtime divided into probing, ML, and search



(a) Gecode heuristics

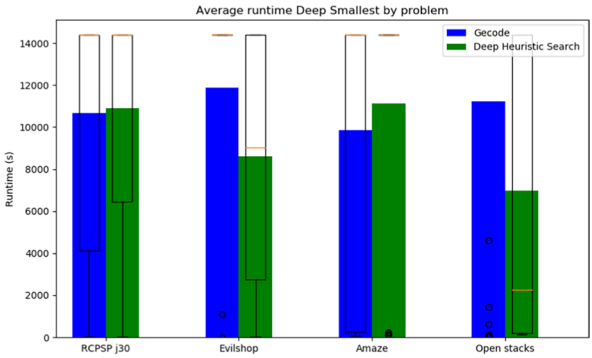


(b) Deep heuristics

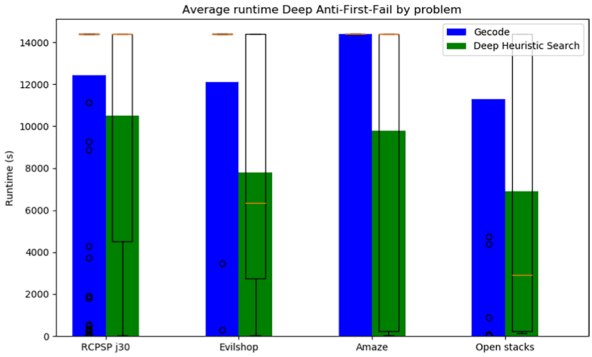
Fig. 6 Comparison of average runtime between heuristics

selection, similarly. Finally we compare the two rankings, by showing how the top three predicted rankings compare to the actual ranking in the dataset. As explained earlier, we report the R^2 and Spearman's rankings, both for the whole ranking and for the top 10.

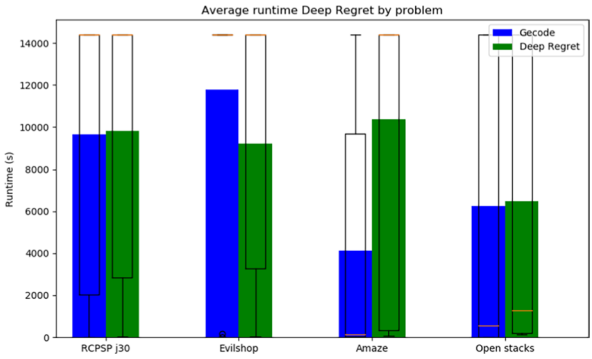
Fig. 7 Comparison of average runtime including timed-out instances



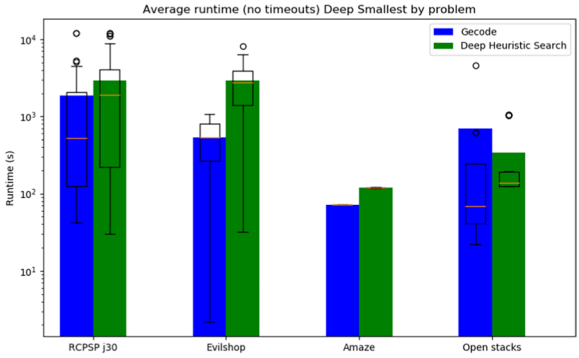
(a) SM versus DS



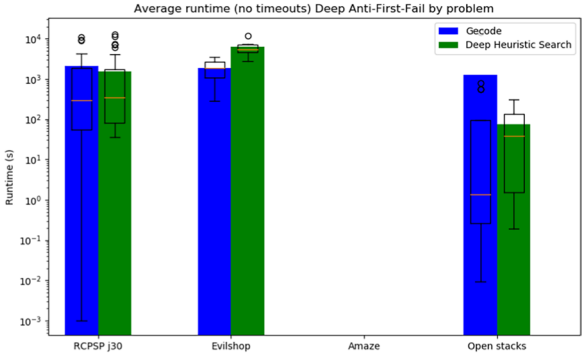
(b) AFF versus DAFF



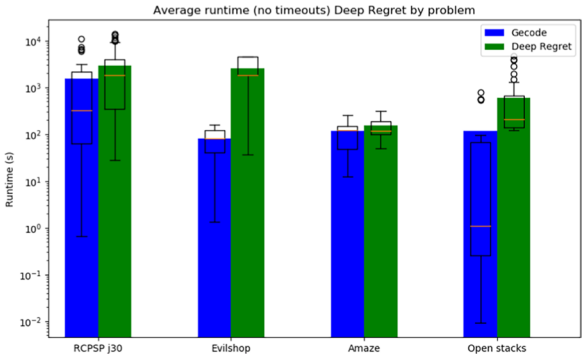
(c) MR versus DR



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Fig. 8 Comparison of average runtime without timed-out instances

Table 2 T-tests of average 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

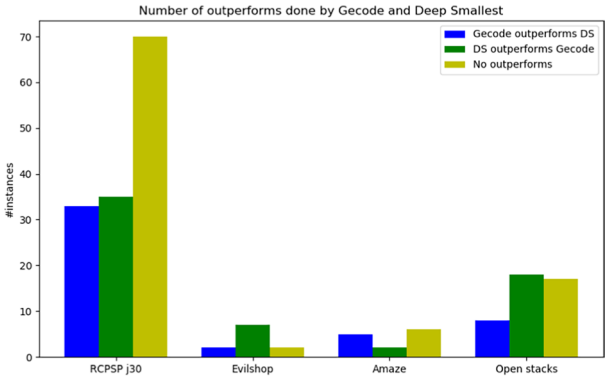
Comparison	Total time		Total time		Job solution time	
	no timeouts		with timeouts		with timeouts	
	p-value	signif.?	p-value	signif.?	p-value	signif.?
DS vs SM: RCPSP	0.12	No	0.68	No	$1.02 \cdot 10^{-13}$	Yes
DR vs MR: RCPSP	0.014	Yes	0.75	No	$6.76 \cdot 10^{-14}$	Yes
DAFF vs AFF: RCPSP	0.45	No	$4.6 \cdot 10^{-4}$	Yes	$2.8 \cdot 10^{-23}$	Yes
DS vs SM: Evilshop	0.19	No	0.08	No	$4.5 \cdot 10^{-5}$	Yes
DR 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
DS vs SM: Amaze	–	No	0.53	No	0.12	No
DR 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
DS vs SM: OS	0.33	No	$3.8 \cdot 10^{-4}$	Yes	$4.86 \cdot 10^{-9}$	Yes
DR 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 3 Percentage of total instances solved by heuristics

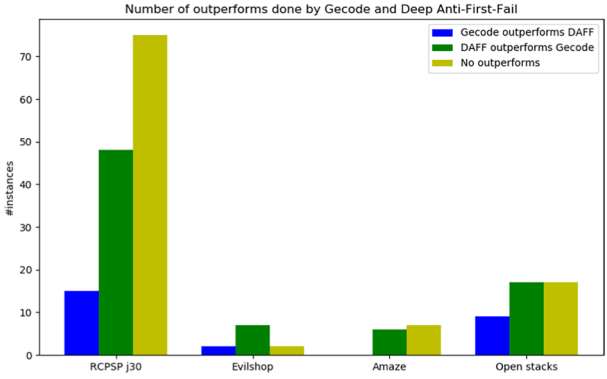
Heuristic	RCPSP j30	Evilshop	Amaze	Open stacks
Gecode smallest	29.7%	54.6%	38.5%	23.3%
Deep smallest	31.2%	18.2%	23.1%	54.3%
Gecode anti-first fail	15.9%	18.2%	18.2%	23.3%
Deep anti-first fail	35.4%	63.6%	63.6%	56.6%
Gecode max regret	37.7%	18.2%	76.9%	57.4%
Deep max regret	41.6%	48.5%	28.2%	58.1%

The R^2 values in Table 4 are high; similarly, the Spearman's rank correlations are mostly around 0. However the true ranking is based on only 20% of the dataset, which means that data for early variable assignment in the total solution could be located in the other 80%. The table also shows the true ranks of predicted values.[6] In general we see that the first three predicted ranks are close to the true first rank based on the average size of the dataset. For example, for DS on RCPSP the 1st predicted rank has as true 1st rank 706 which is $706/41730 \cdot 100 = 1.7\%$ from the top rank. This suggests that the DHSFs are performing better than either R^2 or Spearman can capture (Tables 5 and 6).

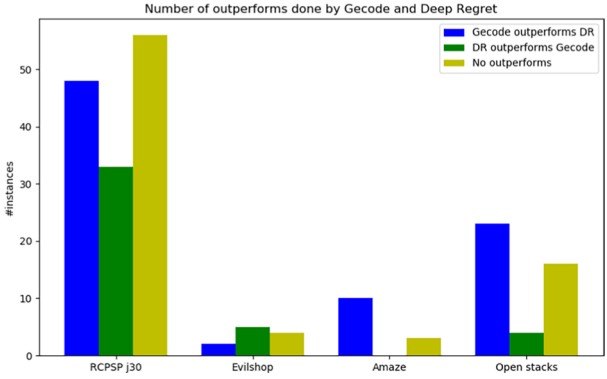
[6] For DR on Evilshop, metrics could not be recorded due to an unknown implementation reason.



(a) SM versus DS



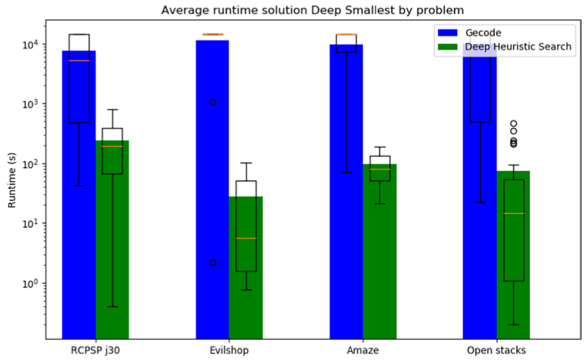
(b) AFF versus DAFF



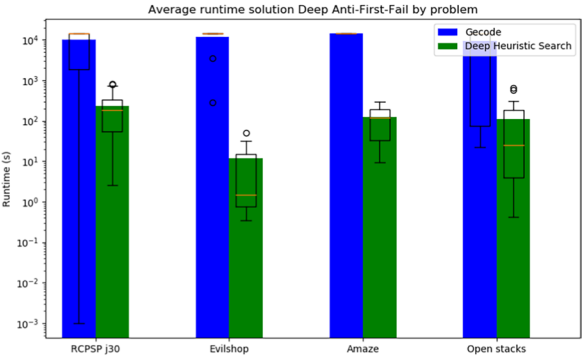
(c) MR versus DR

Fig. 9 Number of times that heuristics outperform each other

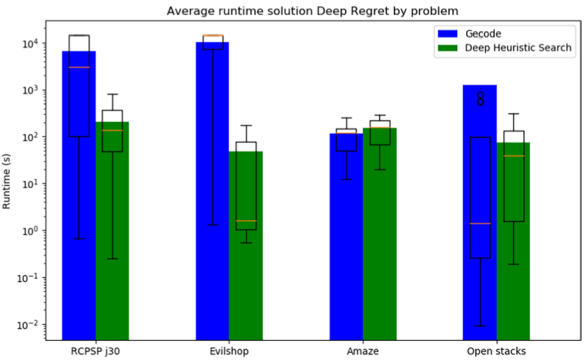
Fig. 10 Comparison of average runtime: deep heuristic solutions



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Table 4 Average quality of deep heuristics presented through R^2, Spearman's rank correlation, and predicted rankings. 'err' denotes value not recorded

Heuristic and Problem	R^2	R^2@10	Spear.	Spear. @10	1st pred. value	2nd pred. value	3rd pred. value	dataset size
DS RCPSP	0.19	−31787	0.40	0.03	706	818	822	41730
DS Evilshop	0.31	−71.74	0.14	0.39	14	15	17	272
DS Amaze	−1.68	−35904	−0.02	0.16	13	13	16	385
DS Open Stacks	0.05	−28957	0.29	0.02	10361	9474	10258	143417
DAFF Fail RCPSP	0.39	−13701	0.57	0.03	1812	1687	1814	41426
DAFF Evilshop	−0.01	−109.17	0.25	0.07	22	22	24	274
DAFF Amaze	−0.32	−7148	−0.04	0.15	16	18	21	1198
DAFF Open Stacks	−0.02	−35837	0.21	0.05	3994	4274	4271	94234
DR RCPSP	−0.12	0.00	0.01	0.00	802	785	724	41449
DR Evilshop	err	err	err	err	err	err	err	270
DR Amaze	−0.16	−0.05	NaN	0.10	34	36	33	1230
DR Open Stacks	−0.12	0.00	0.09	0.02	4741	5120	5010	138506

Table 5 Significance of mean comparison average runtime with timeouts, comparing the p-value to $\alpha = 0.05$

Mean average runtime (with timeouts)	p-value	Significant difference
DS vs SM: RCPSP	0.68	No
DR vs MR: RCPSP	0.75	No
DAFF vs AFF: RCPSP	$4.6 \cdot 10^{-4}$	Yes
DS vs SM: Evilshop	0.08	No
DR vs MR: Evilshop	0.21	No
DAFF vs AFF: Evilshop	0.028	Yes
DS vs SM: Amaze	0.53	No
DR vs MR: Amaze	0.004	Yes
DAFF vs AFF: Amaze	0.018	Yes
DS vs SM: OS	$3.8 \cdot 10^{-4}$	Yes
DR vs MR: OS	0.85	No
DAFF vs AFF: OS	$1.7 \cdot 10^{-4}$	Yes

4.5 Exploring probing meta-parameters

We next focus on a preliminary exploration of three meta-parameters of the deepification process: probing time, job time, and deepification depth.

First, halving probing time leads to increased runtime for most problems, as shown in Fig. 11, with a respective total increase of 9.1%, 15.2%, and 13.7% for DS, DAFF, and DR. We can see that the total probing time is sometimes larger even though probing time was halved: this could be the case because it took more search jobs to find a solution which means more probing phases were initiated (Fig. 12).

Second, Table 7 presents the number of instances solved when we halve the job time from 900 to 450 seconds. Comparing with Table 3, it can be noticed that halving the job time leads to a considerable decrease of the number of instances that are solved. That means that we do

Table 6 Significance of mean comparison average solution runtime with timeouts, comparing the p-value to $\alpha = 0.05$

Mean average solution runtime	p-value	Significant difference
DS vs SM: RCPSP	$1.02 \cdot 10^{-13}$	Yes
DR vs MR: RCPSP	$6.76 \cdot 10^{-14}$	Yes
DAFF vs AFF: RCPSP	$2.8 \cdot 10^{-23}$	Yes
DS vs SM: Evilshop	$4.5 \cdot 10^{-5}$	Yes
DR vs MR: Evilshop	$2.2 \cdot 10^{-3}$	Yes
DAFF vs AFF: Evilshop	$1.2 \cdot 10^{-5}$	Yes
DS vs SM: Amaze	0.12	No
DR vs MR: Amaze	0.62	No
DAFF vs AFF: Amaze	$2.8 \cdot 10^{-21}$	Yes
DS vs SM: OS	$4.86 \cdot 10^{-9}$	Yes
DR vs MR: OS	0.14	No
DAFF vs AFF: OS	$1.2 \cdot 10^{-8}$	Yes

not profit from the increase in the number of search jobs that can be performed: there is simply not enough time to solve certain data instances within 450 seconds.

Figure 13 shows the number of times classical heuristics and deep heuristics outperform one another. Notable is that SM outperforms DS in more instances but DAFF and DR show an improvement in number of outperforms. There is a slight improvement for the Open Stacks problem and no improvement for the Amaze and Evilshop problem.

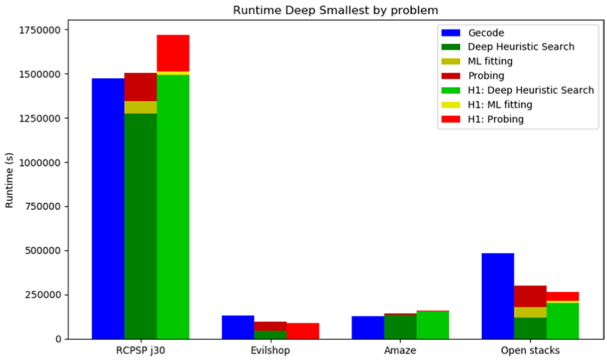
Third, halving the DHSF depth results in an overall increase of 1.3% average runtime and 21.4% decrease in average solution runtime (Fig. 14). Through inspection we see that deep heuristics with a lower depth value perform worse on RCPSP and Open Stacks. On the other hand, Evilshop only sees an improvement with DR while Amaze has an overall lower average runtime. This improvement can be a specific benefit to the Amaze problem because of the increased amount of data gathered. Looking closely at DR on RCPSP, we see a large increase in average runtime. Overall, reducing depth does not seem to positively affect runtime, possibly indicating that deep heuristics need a sufficient depth to find promising solutions (Fig. 15).

5 Related Work

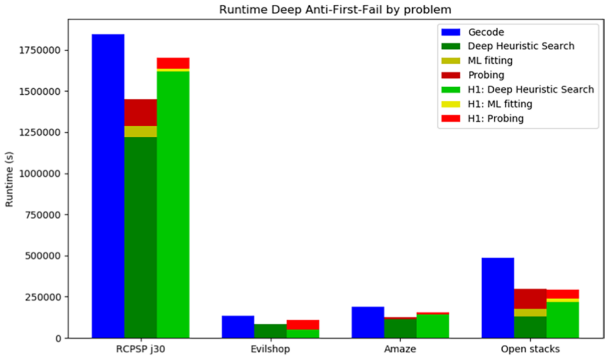
There is much recent interest in combining combinatorial optimisation with reinforcement learning (e.g., [17–20]) and deep neural networks (e.g., [21, 22]). In contrast to most works, our aim is an online setting where training time is included in the total solving time.

Perhaps the most interesting connection is with the work of Rousseau and colleagues [8, 18], who combine reinforcement learning (RL) and constraint programming. The approach is to use a dynamic programming model as a common intermediary. RL acquires a value selection heuristic, which is used in the COP search. The authors note the importance of caching to reduce the number of invocations to the ML component at runtime.

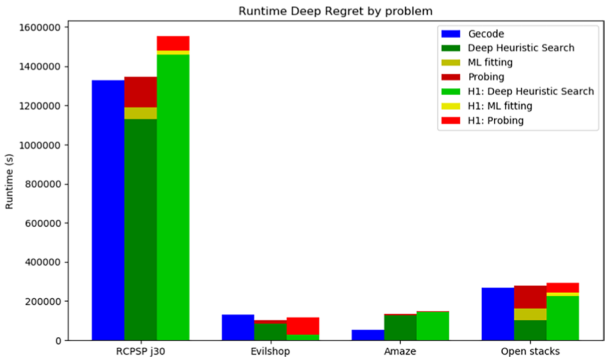
Related to our work in another way is the predict-and-optimise paradigm [23] in that we do not directly learn to solve an optimisation problem; hence the quality of the learned function per se is less important than its use to improve the subsequent solving



(a) SM versus DS



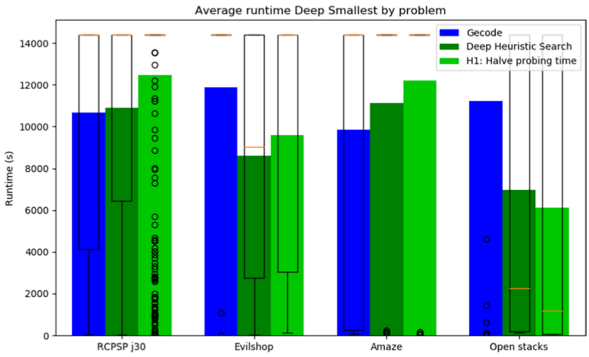
(b) AFF versus DAFF



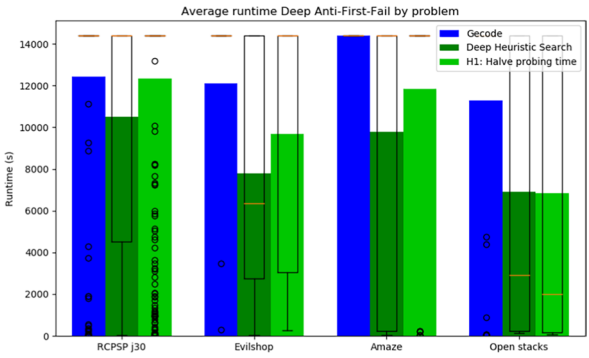
(c) MR versus DR

Fig. 11 Reduced probing time: Total runtime

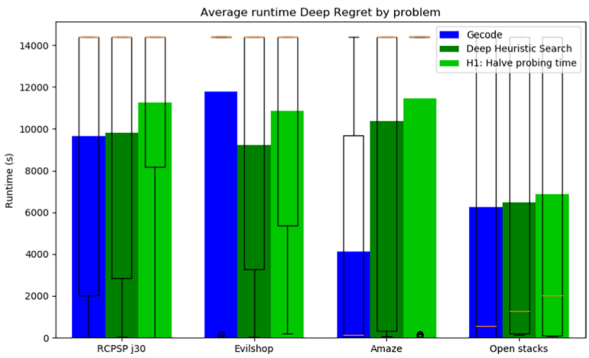
Fig. 12 Hypothesis 1: Average runtime of classical heuristics versus deep heuristics



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Table 7 Reduced job time: Percentage of total instances solved

Heuristic	RCPSP j30	Evilshop	Amaze	Open Stacks
Deep smallest	26.6%	9.1%	20.5%	56.6%
Deep anti-first fail	34.8%	15.4%	28.2%	44.2%
Deep max regret	35.5%	15.2%	25.6%	49.6%

of the combinatorial problem.[7] Another connection is with efforts to estimate the size of a branch-and-bound search tree and estimate the percentage of the search process complete in an anytime fashion [24].

The deep heuristics in our work depend on human expert designed variable ordering heuristics. Haralick and Elliott [2] designed the *first-fail* heuristic, ordering variables on smallest domain first. Boussemart et al. [3] designed *domwdeg*, ordering variables on domain size divided by weighted degree. Refalo [4] designed the *impact*-based search strategy where we order variables or values based on how much a variable's domain size would decrease. Petrovic and Epstein [25] learn which among a set of heuristics to use. These and many other human expert heuristics and meta-heuristics are employed in specific heuristic–problem combinations – and are well studied [26] – while our work is more flexible by learning variable ordering heuristics based on a problem's search tree.

Frost and Dechter [27] develop a value ordering heuristic, *look-ahead value ordering* (LVO), based on the number of times a value of the current variable conflicts with some value of a future variable. The authors do not consider use of machine learning to approximate and accelerate the lookahead.

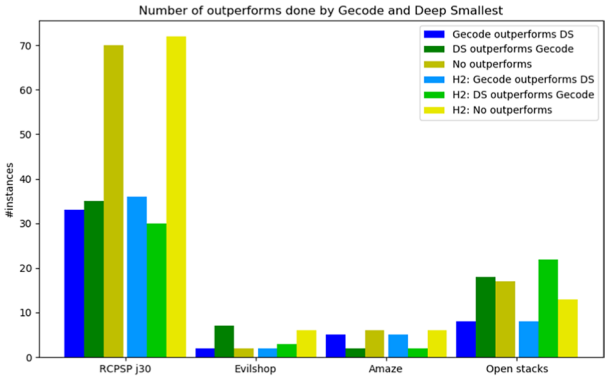
Closer to our work, Chu and Stuckey [10] use online learning to acquire value ordering heuristics, using partial least squares regression to learn the score function. Our approach differs in that, firstly, we learn variable ordering heuristics and we utilise a more complex score function, utilising multiple heuristic score functions over multiple nodes. Second, our framework uses a restart-based approach in probing and in search.

Glankwamdee and Linderoth [28] used lookahead branching on grand-child nodes in a mixed integer program (MIP), finding that information from these nodes often reduces 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.

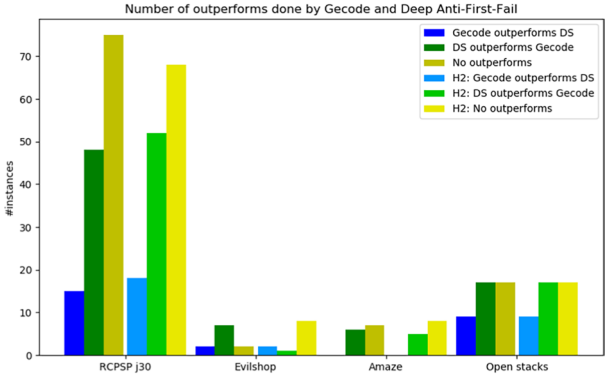
Other works make use of ML to select heuristics. For instance, Xia and Yaap [6] use a multi-armed bandit reinforcement learning approach and experiment with this on non-binary CSPs. The idea is that each arm is a choice for a heuristic; choosing out of multiple arms is a trade-off between exploration and exploitation. Alanazi and Lehre [5] present the limitations of additive reinforcement learning mechanisms for selection hyper-heuristics. Rather than using such selection heuristics, we select an existing variable order heuristic and learn a problem-specific extended version.

[7] Outside the scope of this article, but interesting for the future, is to investigate the relationship between the quality of the learned models used to deepify heuristics and the overall solving performance exploiting those heuristics.

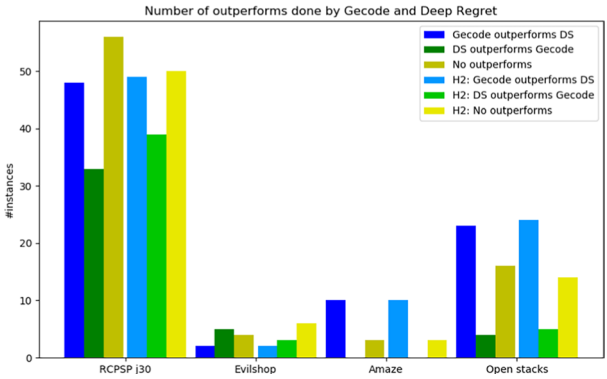
Fig. 13 Reduced job time: Pairwise out-performance counts



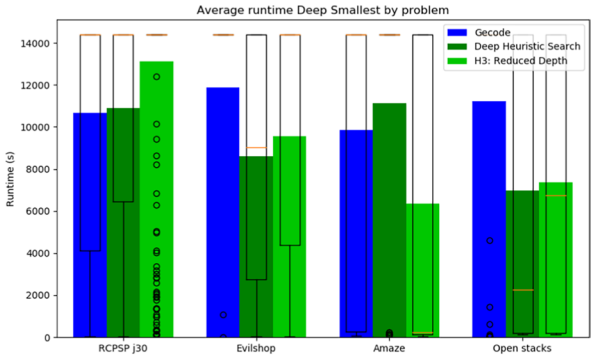
(a) SM versus DS



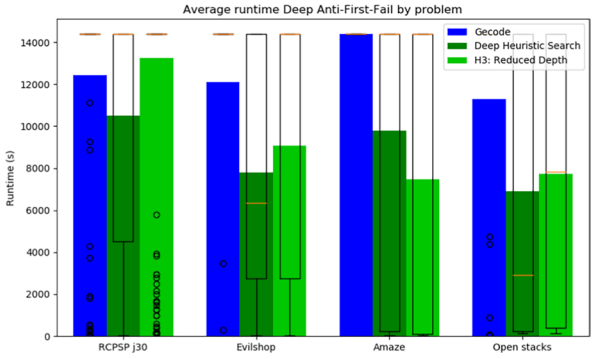
(b) AFF versus DAFF



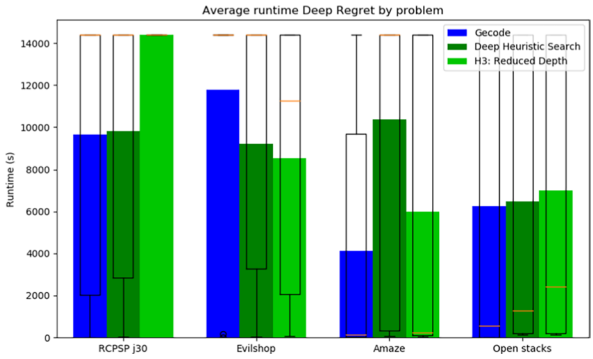
(c) MR versus DR



(a) SM versus DS

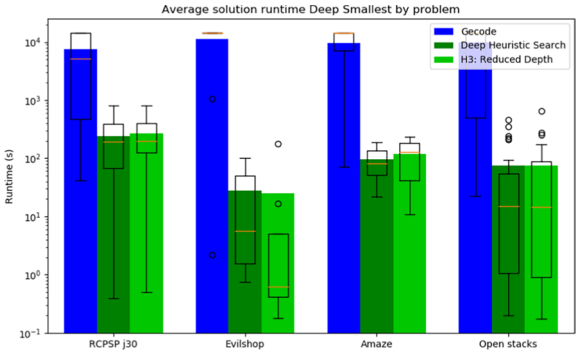


(b) AFF versus DAFF

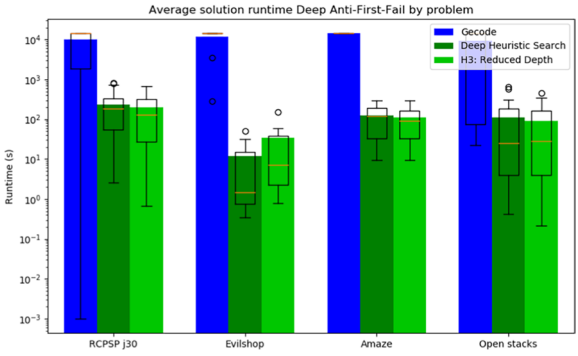


(c) MR versus DR

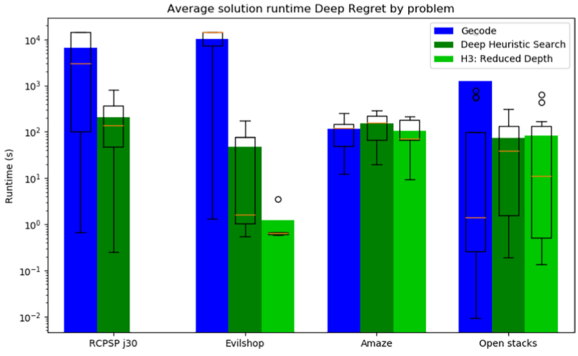
Fig. 14 Reduced depth: Average runtime



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Fig. 15 Hypothesis 3: Average runtime of classical heuristics and deep heuristics solutions

6 Conclusion and Future Work

This article 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 an online probing phase to gather data coupled with a fast regression approach. The deepified heuristics are exploited in restart-based sequential search.

We demonstrate deep heuristics based on three diverse classical 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 representative subset of four MiniZinc benchmarks, while improving on overall runtime for two problem classes.

We compared directly between Gecode using a given classical variable ordering heuristic, and our learn-and-search framework and the (automatically learned) deepified version of the same heuristic. All other aspects were held constant. Our current work is to compare also our approach with Gecode using the same restart-based search, i.e., restarts every 15 minutes instead of the Gecode default strategy.

The result that, overall, deep heuristics solve more instances – albeit with increased average runtime on some problem–heuristic combinations and reduced average on other combinations – suggests development of the approach in a number of possible directions.

First, while we have evidence of the contribution of all the three elements of our approach, a full ablation study of the components is interesting. There is also theoretical interest in seeing how using exact DHSF computation (supposing unlimited computation time) compares with using their approximation.

Second, our framework is implemented in Gecode. To deepify heuristics such as *dom-wdeg*, Gecode needs additional instrumentation: during probing we cannot record, e.g., search node successes, failures, and added or removed constraints. Besides *deep dom-wdeg*, we also attempted to make a deep impact-based heuristic, but faced that Gecode does not implement impact-based search by default; we also could not find suitable problems in the MiniZinc benchmarks which use the impact annotation. On the other hand, deepifying other heuristics such as *first-fail* and *occurrence* face no such hurdles.

Also related to Gecode, further engineering is needed also to parallelise the probing, whatever heuristic is being deepified: Gecode cannot handle our simultaneous node addition to the same search tree.

Third, our exploratory experiments about probing meta-parameters indicate that setting parameters such as probing time and job time has importance. Meta-parameters should be set on separate problem instances to prevent over-fitting. However, based on the presented heuristic–problem results, we think it unlikely that the meta-parameters can be tuned such that deep heuristics work well for any arbitrary problem instance. 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.

Fourth, can we detect (automatically) the cases of instances or problems for which shallow or deep heuristics are likely to be best?

Fifth, we explored the use of random search in probing. The use of specific probing heuristic(s) is interesting. For example, the *smallest* heuristic could be used to gather data for *deep smallest*. However, the risk of using a specific heuristic during probing could be its bias. Sixth, use of the score from the DHSF might be improved by adding exploration

and exploitation attributes. For instance, one could add a discount factor to give more weight to earlier choices adhering to the principle of making good early decisions [12].

Distributing our method into MiniZinc is easy: Gecode allows for the implementation of custom branchers which can be called by adding a specific annotation for flatzinc files. Further, we used Python within C++ to use the latest ML libraries. However, integrating the ML library calls from within C++, i.e., within Gecode, is advised. This prevents Python script calls and the need to install Python as a dependency, avoids calling Python from C++ which risks memory leaks, and provides in principle a faster and more efficient implementation than Python.

This article showed how to deepify variable ordering heuristics. 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. [29]). 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. Besides running the jobs in parallel, 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.

Acknowledgments The authors thank the anonymous reviewers for their suggestions. We thank the late C. Schulte, and G. Tack, for assistance with Gecode and MiniZinc. Thanks to the participants of the IJCAI'21 DSO workshop, the CP'21 PTHG workshop and BNAIC'21, where earlier versions of this work were presented. Thanks to E. Freuder and H. Simonis for their suggestions. This research was partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant number 952215.

Declarations

Competing interests None

References

1. Gent, I.P., MacIntyre, E., Prosser, P., Smith, B.M., Walsh, T.: 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), pp. 179–193 (1996)
2. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artif. Intell. **14**(3), 263–313 (1980)
3. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of 16th European Conference on Artificial Intelligence (ECAI'04), pp. 146–150 (2004)
4. Refalo, P.: Impact-based search strategies for constraint programming. In: Proceedings of 10th International Conference on the Principles and Practice of Constraint Programming (CP'04), pp. 557–571 (2004)
5. Alanazi, F., Lehre, P.K.: Limits to learning in reinforcement learning hyper-heuristics. In: Proceedings of 16th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP'16), pp. 170–185 (2016)
6. Xia, W., Yap, R.H.C.: Learning robust search strategies using a bandit-based approach. In: Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18), pp. 6657–6665 (2018)
7. Khalil, E.B., Dilkina, B., Nemhauser, G.L., Ahmed, S., Shao, Y.: Learning to run heuristics in tree search. In: Proceedings of 26th International Joint Conference on Artificial Intelligence (IJCAI'17), pp. 659–666 (2017)
8. Cappart, Q., Moisan, T., Rousseau, L., Prémont-Schwarz, I., Ciré, A.A.: Combining reinforcement learning and constraint programming for combinatorial optimization. CoRR arXiv:abs/2006.01610 (2020)
9. Schulte, C., Tack, G., Lagerkvist, M.Z.: Modeling and Programming with Gecode 6.2.0. www.gecode.org (2019)

10. Chu, G., Stuckey, P.J.: Learning value heuristics for constraint programming. In: Proceedings of 12th International Conference on the Integration of AI and OR Techniques in Constraint Programming (CPAIOR'15), pp. 108–123 (2015)
11. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Foundations of Artificial Intelligence, vol. 2. Elsevier, Amsterdam (2006)
12. Ortiz-Bayliss, J.C., Amaya, I., Conant-Pablos, S.E., Terashima-Marín, H.: Exploring the impact of early decisions in variable ordering for constraint satisfaction problems. Computational Intelligence and Neuroscience **2018**, 6103726–1610372614 (2018)
13. The MiniZinc Benchmark Suite. MiniZinc: https://github.com/MiniZinc/minizinc-benchmarks (2016)
14. Blazewicz, J., Lenstra, J.K., Kan, A.H.G.R.: Scheduling subject to resource constraints: classification and complexity. Discret. Appl. Math. **5**(1), 11–24 (1983). https://doi.org/10.1016/0166-218X(83)90012-4
15. Hartmann, S., Briskorn, D.: A survey of variants and extensions of the resource-constrained project scheduling problem. Eur. J. Oper. Res. **207** (1), 1–14 (2010). https://doi.org/10.1016/j.ejor.2009.11.005
16. Chu, G., Stuckey, P.J.: Minimizing the maximum number of open stacks by customer search. In: Proceedings of the 15th International Conference on the Principles and Practice of Constraint Programming (CP'09), Springer, pp. 242–257. (2009).https://doi.org/10.1007/978-3-642-04244-7_21
17. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. coRR (2016)
18. Chalumeau, F., Coulon, I., Cappart, Q., Rousseau, L.: Seapearl: A constraint programming solver guided by reinforcement learning. CoRR arXiv:abs/2102.09193 (2021)
19. Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., Rousseau, L.: Learning heuristics for the TSP by policy gradient. In: Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR'18), pp. 170–181 (2018)
20. Kool, W., van Hoof, H., Welling, M.: Attention, learn to solve routing problems!. CoRR arXiv:abs/1803.08475 (2018)
21. Hottung, A., Tanaka, S., Tierney, K.: Deep learning assisted heuristic tree search for the container premarshalling problem. CoRR arXiv:abs/1709.09972 (2017)
22. Song, W., Cao, Z., Zhang, J., Xu, C., Lim, A.: Learning variable ordering heuristics for solving constraint satisfaction problems. Eng. Appl. Artif. Intel. **109**, 104603 (2022). https://doi.org/10.1016/j.engappai.2021.104603
23. Mandi, J., Demirovic, E., Stuckey, P.J., Guns, T.: Smart predict-and-optimize for hard combinatorial optimization problems. In: Proceedings of 34th AAAI Conference on Artificial Intelligence (AAAI'20), pp. 1603–1610 (2020)
24. Anderson, D., Hendel, G., Bodic, P.L., Viernickel, M.: Clairvoyant restarts in branch-and-bound search using online tree-size estimation. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI'19), pp. 1427–1434. (2019). https://doi.org/10.1609/aaai.v33i01.33011427
25. Petrovic, S., Epstein, S.L.: Tailoring a mixture of search heuristics. Constraint Programming Letters **4**, 15–38 (2009)
26. Wallace, R.J.: Determining the principles underlying performance variation in CSP heuristics. International Journal of Artificial Intelligence Tools **17**(5), 857–880 (2008)
27. Frost, D., Dechter, R.: Look-ahead value ordering for constraint satisfaction problems. In: Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI'95), pp. 572–578 (1995)
28. Glankwamdee, W., Linderoth, J.: Lookahead Branching for Mixed Integer Programming. Technical Report, Lehigh University. Department of Industrial and Systems Engineering (2006)
29. Cox, J.L., Lucci, S., Pay, T.: Effects of dynamic variable–value ordering heuristics on the search space of sudoku modeled as a constraint satisfaction problem. Intel. Artif. **22**(63), 1–15 (2019)