Deepification: Learning Variable Ordering Heuristics in Constraint Optimization Problems

Floris Doolaard

Deepification: Learning Variable Ordering Heuristics in Constraint Optimization Problems

by

Floris Doolaard

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday August 26, 2020 at 09:30 AM.

Student number:	4362748	
Project duration:	November 22, 2018 – August 26, 2020	
Thesis committee:	Dr. N. Yorke-Smith,	TU Delft, supervisor
	Dr. M. M. de Weerdt,	TU Delft
	Dr. ir. J. A. Pouwelse,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TUDelft

Abstract

Constraint programming is a paradigm for solving combinatorial problems by checking whether constraints are satisfied in a constraint satisfaction problem or by optimizing an objective in a constraint optimization problem. To find solutions, the solver needs to find a variable and value ordering. Numerous heuristics designed by human experts already exist to guide search and recent research uses machine learning to learn new heuristics. In this work the concept of deep heuristics is introduced. First, data is collected during a probing phase after which a deep heuristic function is learned based on the smallest, anti first-fail, and maximum regret heuristics. The learned deep heuristics look arbitrarily many levels in a search tree instead of a single instant lookup for normal heuristics. The results show that deep heuristics solve 20.5% more problem instances than normal heuristics while improving on overall runtime for the Open Stacks and Evilshop problems.

Preface

This thesis is the final step in completing the MSc Computer Science programme at TU Delft. I would like to take the opportunity to give my sincerest gratitude towards the people that have supported me during this final stage.

First of all I would like to thank Neil Yorke-Smith for supervising me and for your inexhaustible positive encouragement. Thank you for the many interesting discussions and for your feedback when I needed it. I would like to thank the other members of thesis committee Mathijs de Weerdt and Johan Pouwelse for taking the time to assess my thesis.

I am grateful to Christian Schulte, who unfortunately passed away earlier this year. He was a huge expert in the field of constraint programming and helped me to get started with Gecode. I also want to thank Guido Tack, who leads development of MiniZinc and is one of the main developers for Gecode, for helping me out with Gecode and Minizinc.

I could not have started this master's programme without the wonderful bachelor's project I did with David Alderliesten, Jesse Tilro, and Niels Warnars. Thank you for the good times. Overlapping with my thesis project I joined the board of the Delft Student E-sports Association (DSEA) and I would like to thank my fellow old-board members Rutger Bosmans, Remon de Graaf, Tim Rijkers, and Tim Yue for all the support you gave me during the year. I am very grateful for being a part of DSEA from the very beginning 5 years ago, getting to know so many lovely people over the years. Thank you Justin van der Krieken, Tim Rijkers, Quinn Begelinger, and Maxime van Elsué for assisting me with Linux, design, and mental support where I needed it. The wonderful cover was designed by Jesper van den Berg, thanks a lot. Finally, from the bottom of my heart, I want to thank my parents Rob and Conny for always supporting me and believing in me no matter what I do.

Delft, August 2020

Contents

1 Introduction

In 1997 for the first time in history, Deep Blue, a chess computer, beat the human chess world champion Garry Kasparov in a match. This was very much a surprise to the world as most people had underestimated the capabilities of the tin machine. How could a computer ever think like a human? Nowadays we simply do not know any better than that a computer simply out-calculates us in chess and in many other problems.

To go even a step further, AlphaGo [40] by DeepMind added a neural network to its algorithm to beat the Go world champion Lee Sedol. Soon DeepMind implemented AlphaZero [41] to master the game of chess and this again made a huge impact in the chess world by showing that a computer could not only easily calculate a sequence of chess moves, but also take into account the deeper positional features of the game.

The topic that revolves around solving problem like chess, go, but also other problems in the area of planning and scheduling, and logistics, is called `Combinatorial Optimization` (CO). CO problems typically include many `NP-hard` problems, problems for which no algorithm exists that would solve them in polynomial time, which often rules out the use of exhaustive search.

One of the methods to avoid brute-force algorithms is the use of approximation algorithms. For example, based on the features of the current state of a game or problem we could compute some score based on a function. Knowing such a function would give us the best choice in every state, however this function is usually unknown and can be instead approximated based on observations of data. This method of approximation is called `Machine Learning` (ML) and is utilized in this thesis with the use of `Constraint Programming` (CP), a discrete optimization paradigm using constraints on a set of decision variables, to solve CO problems.

1.1. Problem statement

In CP one typically models problems into `Constraint Satisfaction Problems` (CSPs), where the constraints have to be satisfied, or `Constraint Optimization Problems` (COPs), adding an objective value to minimize or maximize next to satisfying the constraints. In these models a constraint solver is given a set of decision variables with a domain of possible values to assign to find a solution. A solution, or failure if there is none, is found by assigning a value to one of the variables one by one. However, the order in which the variables are chosen can have significant effect on the total runtime of the solver [22].

Various variable ordering heuristics have been designed by human experts by learning them with ML or learning which heuristic to use. However, the learned heuristics are based on the current search node. Besides, some ML methods may require a lot of training time before starting search while others have a hard time generalizing data.

In this work we focus on using regression analysis, a basic ML technique, to learn novel `deep variable ordering heuristics` by approximating functions that look at multiple levels of a search tree with the aim to generalize better than normal heuristics.

1.2. Research questions

In order to research the problem statement the main research question is defined as

> **How can machine learning improve variable ordering heuristics for constraint optimization problems?**

To answer this question the following sub-questions are asked:

1. How to collect data to fit a machine learning model in constraint optimization problems?
2. How can the fit of a machine learning model to learn heuristics be improved in constraint optimization problems?
3. How can a search strategy improve search in constraint optimization problems when using machine learning?

1.3. Methodology

Deep heuristics, the learned variable ordering heuristics, will be implemented in Gecode [1], a constraint solver. To do this, search strategies within Gecode are used to start a probing phase in which pseudo-random data is gathered. This data, including features and labels, is then utilized by regression analysis to learn a deep heuristic function. Finally, given the current search state heuristic can predict scores with the learned model and select the variable with the best predicted score. Because the dataset is pseudo-random a restart-based search strategy is added by halting search after 15 minutes and starting from scratch. This allows for multiple machine learning models to be learned which increases the chance of finding solutions.

MiniZinc [33] is able to use the implementation within Gecode which makes it possible to test it on the RCPSP, Evilshop, Amaze, and Open Stacks COPs. We compare the total runtime, average runtime, and number of instances solved of the learned deep heuristics 'Deep Smallest', 'Deep Maximum Regret', and 'Deep Anti-First-Fail' versus Smallest, Maximum Regret, and Anti-First-Fail. To test the quality of the machine learning model we use R^2 and Spearman's rank correlation.

1.4. Document structure

The outline for this thesis is as follows. In chapter 2 the necessary background and definitions of CP and ML are provided. Chapter 3 will introduce and formalize the concept of deep heuristics. Chapter 4 describes the framework which uses the deep heuristics to solve constraint problems. Chapter 5 tests the framework against four COPs. Based on those results, three hypotheses for parameter selection are added to further evaluate performance. Finally, chapter 6 discusses the results and chapter 7 concludes the thesis by indicating limitations, answering the research questions, and prospects for future research.

2 Background and definitions

This chapter starts with an introduction to constraint programming and machine learning and then presents related work in this area.

2.1. Constraint Programming

2.1.1. Constraint Satisfaction Problem

In a CSP a set of variables have to be assigned with values chosen from a range of values. The range of the values that can be assigned is defined in the domain of a variable. More formally CSP(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
- C is a finite set of constraints what values each variable $v \in V$ may take

2.1.2. Constraint Optimization Problem

One can also add an objective function to a CSP which is typically maximized or minimized. This turns the CSP into a Constraint Optimization Problem (COP).

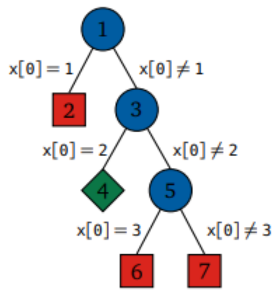
2.1.3. Search

To find solutions in CSPs and COPs a `CP solver` typically uses search trees. Figure 2.1 shows such an example of a search tree where edges present value assignments and nodes represent the current variables with their domains. Assigning certain values may lead to empty domains, resulting in failures such as when assigning $x[0] = 1$. When every domain size is exactly one then a solution has been found denoted by the green node.

In this section it is first shown how a constraint problem can be solved via search after which different popular techniques are explained to improve this process.

The most intuitive search is the brute-force method of backtracking by always selecting the first available variable in V and value v in its domain D. Take the example of the n-queens problem, a constraint satisfaction problem, where the goal is to place n queens on a n times n chessboard such that none of the queens can touch each other if they were to move. Queens may move horizontally, vertically, and diagonally. Figure 2.2 shows the backtracking process for a 4-queens problem where the columns of the chessboard can be seen as variables and the rows as values. Thus, search begins by selecting the first variable with its first value, column 1 and row 1. Then for the second variable the solver has problems with the first two rows as the queens touch and thus that branch results in a failure. This process repeats itself until a solution has been found or when every variable and value have been tried.

Inference Backtracking alone is not at all efficient as we iterate over all variables and values. Constraint programming solvers often implement propagators which aim to remove redundant values from the domain of a variable. For the n-queens problem we can define a simple procedure as in algorithm 1.



node	x[0]	x[1]	x[2]
1	{1,...,5}	{1,...,5}	{2,...,6}
3	{2,...,5}	{1,...,3}	{3,...,6}
4	2	2	4
5	{3,...,5}	{1,...,2}	{4,...,6}

Figure 2.1: Example of a search tree [1]

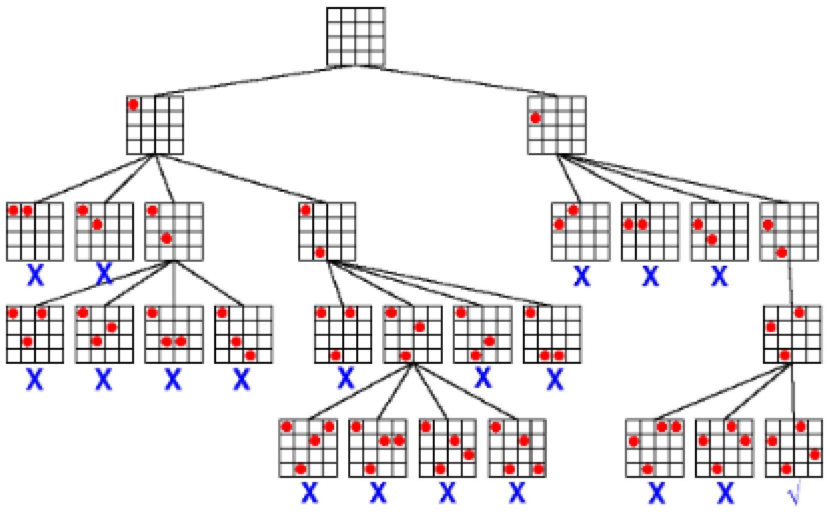


Figure 2.2: Backtracking Search in the 4-queens CSP by Roman Barták, 1998 (https://ktiml.mff.cuni.cz/ bartak/constraints/images/back-track.gif)

Algorithm 1: Propagator n-queens problem

```
$x \leftarrow$ placed queen column;
$y \leftarrow$ placed queen row;
$n \leftarrow$ board width;
**while** $x$ *is not* $n$ **do**
    RemoveValueFromAllDomains$(x, y)$;
    $x = x + 1$;
**Reset** $x$ to placed queen column;
**while** $x$ *and* $y$ *are not* $n$ **do**
    RemoveValueFromAllDomains$(x, y)$;
    $x = x + 1$;
    $y = y + 1$;
```

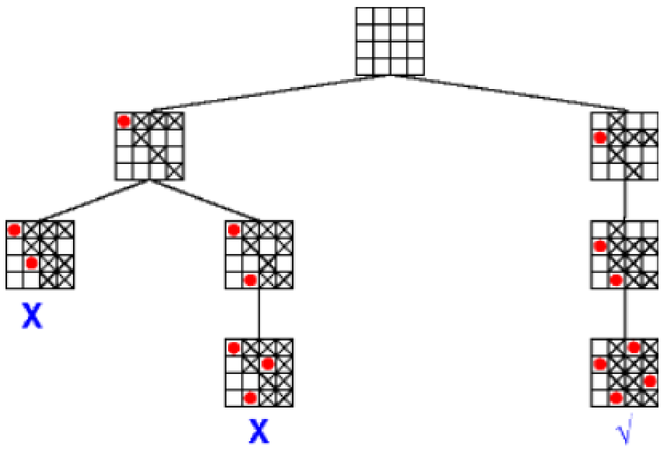


Figure 2.3: Inference by propagation in the 4-queens CSP by Roman Barták, 1998 (https://ktiml.mff.cuni.cz/ bartak/constraints/images/forward.gif)

Combining propagation together with backtracking results in a smaller search tree as seen in figure 2.3 where the crosses on the squares on the board represent the values removed by propagation.

Rossi et al. [36] provides an overview of other CP techniques.

2.1.4. MiniZinc

MiniZinc is an open-source constraint modelling language[1] in which both CSPs and COPs can be modelled [33]. Modelling happens through defining variables, decision variables, and the constraints on the variables.

Once the modelling has been completed one can solve constraint problems by choosing one of the default solver within the MiniZinc library: Gecode, Chuffed, findMUS, Globalizer or OSICBC. The solver is responsible for the search itself and there are no declarations of how to search for solutions. However, it is possible to communicate with the solver through search annotations. Should the solver support a certain heuristic for variable ordering or value selections then it is possible to add them to an annotation such as in listing 2.1 for a CSP where `first fail` selects the variable which is likely to fail first and `indomain_min` select the lowest value in a variable's domain.

```
solve :: int_search(q, first_fail , indomain_min) satisfy;
```

Listing 2.1: MiniZinc syntax, solving for an array q of decision variables

Picking the appropriate strategy for both modelling and solving may improve search time, but there is typically no heuristic or modelling strategy which works best for any CSP or COP. That is because in general they are NP-hard.

2.2. Heuristics

The term `heuristic` derives from the Greek verb 'heuriskein' which means 'to find'. A `heuristic` or `search heuristic` is a model or technique to solve problems more quickly than exact algorithms and often do so by approximating the solution. We can classify a heuristic into two types: `perturbative` or `local heuristics`, which operate on an already fully instantiated candidate set, and `constructive` heuristics, which may iteratively expand a candidate set [10]. In this work we will be using constructive variable ordering heuristics and some well-known heuristics will be presented in this section.

2.2.1. Variable ordering heuristic

Given a constraint problem with domain D and variables X, a `variable ordering heuristic` should decide which variable to assign a value to first. Below you will find an overview of such heuristics including the

[1]https://www.minizinc.org/

ones used in this work. Other variable ordering heuristics can be found in [36].

First-fail A typical heuristic is `first-fail` [25] (FF) that selects variable x 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 as empty domains result in failures. On the other hand `anti-first-fail` (AFF) would pick the variable with the largest domain, least likely to fail.

Smallest The `smallest` (SM) heuristic simply chooses the variable with the smallest value in its domain.

Maximum regret `Maximum regret` (MR) chooses the variable with the largest difference between the two smallest values in its domain and is based on `regret` theory . If a second choice was more desirable than the picked choice then a human may experience regret, however would the first choice also be the most desirable then one could experience extra pleasure based on the difference with the next value. [31]

Domwdeg The `domwdeg` heuristic by Boussemart et al. [6] stands for 'domain weighted degree' and chooses the variable with the smallest value of domain size divided by weighted degree, where the weighted degree is the number of times the variable has been in a constraint which failed.

2.2.2. Value heuristic

Given a constraint problem with domain D and variable x, a `value heuristic` is to give the order in which the values in domain D should be explored. To create an ordering, typically a scoring function $score(X)$ with features X is constructed to indicate how good it is to assign v to x given D [13]. Such a scoring function can use the variable ordering heuristics explained in section 2.2.1.

2.3. Basics of machine learning

Machine learning (ML) is the act to use stored information to answer questions and to draw new conclusions [37]. In this section we will give an introduction to ML by presenting a few techniques and challenges.

2.3.1. Learning algorithm

Mitchell [32] defines a machine learning algorithm as "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." In this section not all possible parts that may be used are defined. Instead the ones used in this research are explained in more detail.

Task T The task of a machine learning algorithm is not the learning itself but task accomplished by learning. An example could be to determine similarity between two images. Tasks are usually described by what happens when the algorithm gets a certain example input for example filled with features, describing an object or event. Two common machine learning tasks are:

1. **Classification** In this task the algorithm should determine to which class or category a certain input belongs to. Typically a function $f : \mathbb{R}^n \to \{1,\dots,k\}$ is learned by the algorithm to compute this where k is the number of classes. An example is the problem to classify which painting belongs to which famous painter. Given the features, such as paint colors or frame size, the algorithm should decide its class.

2. **Regression** With Regression the algorithm should predict a numerical value given an input. Thus, approximating a function $f : \mathbb{R}^n \to \mathbb{R}$. Examples are predicting how much money there will be on a bank account one month into the future, or predicting the score for a given chess move.

Other tasks are found in the deep learning book by Goodfellow et al. [24].

Performance measure P When a machine learning algorithm has a certain task, we must evaluate how well the task is performed. The performance measure P is usually specific to task T. For classification accuracy can be used to look at how many of the outputted classes are correctly predicted.

It is not always straightforward to choose a metric which works well with the desired system. For instance, should we penalize a regression system more if it makes a lot of small mistakes or rare large mistakes?

Experience E The experience of a machine learning algorithm is about what kind of data it is allowed to experience throughout its learning process. Learning algorithms can be categorized in two types of experiences, unsupervised learning and supervised learning.

1. **Unsupervised learning** In this task the algorithm should determine to which class or category a certain input belongs to. Typically a function $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$ is learned by the algorithm to compute this where k is the number of classes. An example is the problem to classify which painting belongs to which famous painter. Given the features, such as paint colors or frame size, the algorithm should decide its class.

2. **Supervised learning** With Regression the algorithm should predict a numerical value given an input. Thus, approximating a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Examples are predicting how much money there will be on a bank account one month into the future, or predicting the score for a given chess move.

Unsupervised learning In unsupervised learning, machine learning algorithms experience a dataset with many features and try to learn properties to give structure to the dataset. There are also clustering algorithms which try to divide the dataset into examples with the same features.

Supervised learning In supervised learning, learning algorithms experience a dataset with features but attached to the examples are `labels`, `rewards`, or `targets`. For example, given the feature set of $X : \{3, 6, 1\}$ and attached labels $y : \{6, 12, 2\}$ a machine learning model can approximate a function $f(X) = y$ to be $f(x) = 2x$.

Online and offline learning The term `online` is commonly used to refer to algorithms that process input data the moment they are received. Hence, `online learning` is when a machine learning model is fitted or a prediction is made in short time after input data is received. On the other hand, in `offline learning` we gather data, typically for a long period, to fit a model and afterwards use the model to make predictions. [37]

Reinforcement learning adds a feedback loop between the learning process and the experience with the dataset. More information about this can be found in [43]. Another technique, `Deep learning` is to allow computers to learn from experience and understand the world in terms of a hierarchy of concepts, with each concept defined through its relation to simpler concepts. The hierarchy of concepts enables the computer to learn complicated concepts by building them out of simpler ones. If we draw a graph, a `neural network`, showing how these concepts are built on top of each other, the graph is deep, with many layers. Hence, the name `deep learning`. A detailed overview on this is found in the deep learning book by Goodfellow et al. [24].

Figure 2.4 shows the relation of machine learning in artificial intelligence.

2.3.2. Generalization

A huge challenge in machine learning is the ability of the trained algorithm to perform well on new examples. This ability is called generalization and typically the aim is to keep the `generalization error`, or `test error` as low as possible. To test the generalization error the dataset is split up into a training set to train the algorithm and a test set to test the newly trained algorithm on. This prevents training and testing on the same dataset causing `bias`. When training on the training set we can measure a `training error` which we try to minimize by setting paramaters. Finally, we test the trained model on the test set to measure the test error.

The training error should be as small as possible to get a better algorithm but we also need to minimize the gap between the training error and test error. This is to prevent underfitting, where the training error is not small enough, or overfitting, where the gap between training error and test error is too large. We can alter the `capacity` of a model to control whether it will underfit or overfit which is the ability to fit a wide variety of functions. Low capicity and the model will struggle to find a fitting function, otherwise a high capicity and the model may use too much information from the training set to fit a function. Figure 2.5 shows the different scenarios.

2.3.3. Regression analysis

In this section the typical regression methods Support-Vector Regression (SVR), Stochastic Gradient Descent (SGD), and Random Forest Regression (RFR) are explained, of which RFR will be used in this work. We use the implementations of Scikit-Learn in Python [35].

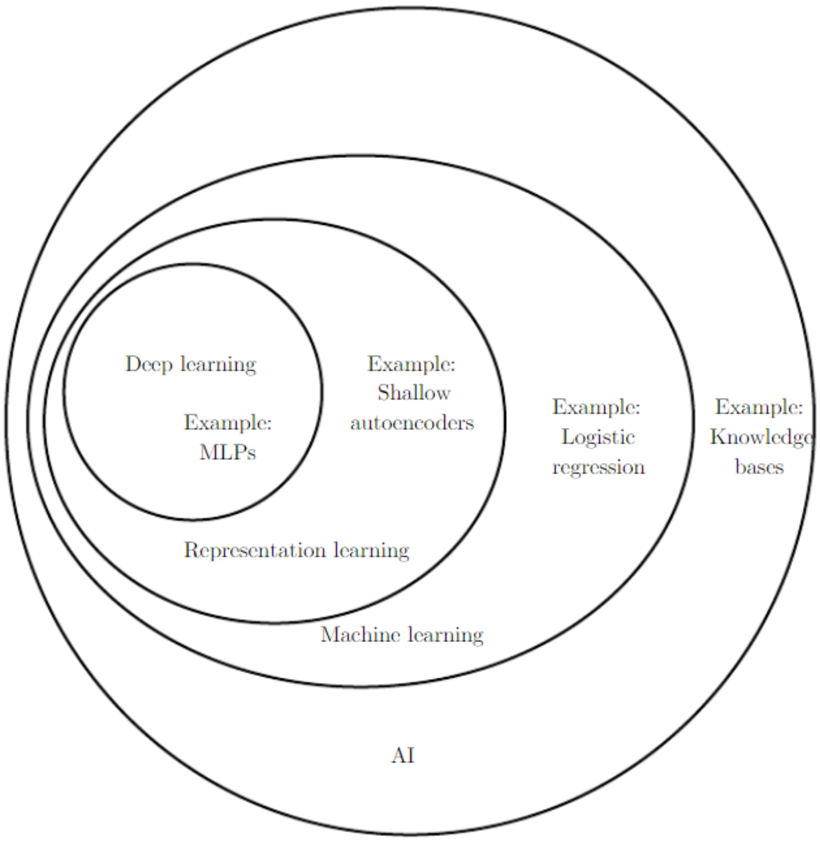


Figure 2.4: Venn diagram showing the relation of areas within machine learning by Goodfellow et al., 2016 [24]

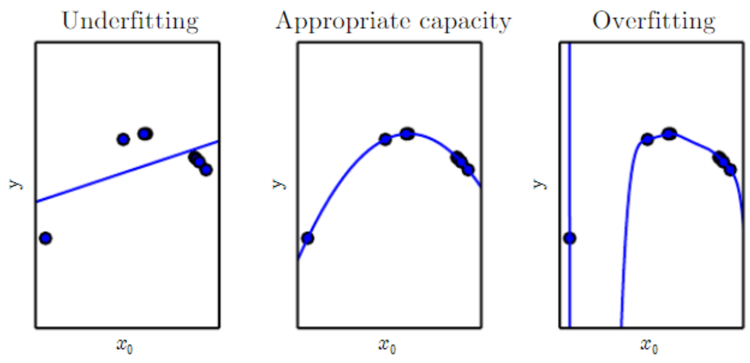


Figure 2.5: Fitting of a model onto a dataset by Goodfellow et al., 2016 [24]

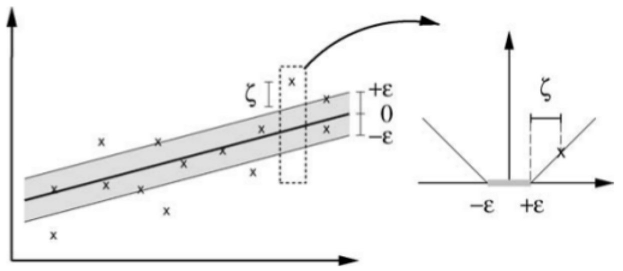


Figure 2.6: SVR with a boundary plane of width 2ϵ in which no point is included in the loss margin ζ by Schölkopf and Smola, 2002 [38]

Support vector regression SVR is an extended form of Support Vector Classification (SVC) or Support Vector Machine (SVM). The SVM is a popular approach for supervised learning if you do not have any specialized prior knowledge about a domain [37]. Three properties make SVMs attractive:

1. SVMs construct a `maximum margin separator` which is a decision boundary with the largest possible distance to example points. An example of such a boundary can be seen in figure 2.6.

2. SVMs have the ability to embed data into a higher-dimensional space, using a so-called `kernel trick`.

3. SVMs are a nonparametric method: they retain training examples and potentially need to store them all. They have the flexibility to represent complex functions, but they are resistant to overfitting.

Stochastic gradient descent If we have the task of finding the best h where

$$h(x) = w_1 x + w_0$$

then this can be typically done by minimizing a loss function such as the squared loss function with for example a training set of n points. Such a loss function can be defined as

$$Loss(h) = \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2$$

which is minimized when its partial derivatives, weights w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (w_1 x_j + w_0))^2 = 0$$

However, often these equations do not have a closed-form and can instead be seen as a general optimization search problem. This problem can be addressed by a hill-climbing algorithm that follows the gradient of a function to be optimized. Because we are minimizing the loss, it is called `gradient descent`. We call it `stochastic gradient descent` when we only consider a single training point at a time [37].

Random Forest Regression Another supervised learning method is the use of `Decision Trees` where we try to predict a target value by learning decision rules from data features. A `Random Forest Regressor` constructs multiple decision trees during training and averages the outcome of those trees as a resulting prediction. [7, 27]

2.4. Related work

There is many work in the field of combining combinatorial optimization with reinforcement learning [2, 15, 18, 29] and deep neural networks [8, 21, 28, 42, 45]. However, in this work our aim is to use regression analysis to prevent long training time and we use constraint programming to solve combinatorial problems.

Chu et al. [13] use ML to learn value heuristics. First training instances are created and features are selected. These features are mostly the lower bounds, upper bounds or assignments of variables close to the decision variable in the constraint graph. Then the `partial least squares regression` method is utilized to learn the score function. Our approach differs in a few ways. Firstly, we learn variable ordering heuristics and we utilize a more complex score function, utilizing multiple heuristic score functions over multiple nodes. Secondly, our framework uses a restart-based approach in probing and in search. Dewanto et al. [19] use `epsilon support vector regression` to learn a search heuristic function. In their proposed algorithm they support both offline and online learning. In offline learning testing is interleaved with training as the learning model is updated between two search planning processes. In the online setting the learning model is updated during every search process. Our framework uses restart-based searching without any training in between search processes.

Bessiere et al. [3] describe a framework to utilize ML in constraint programming (CP) which they call the `inductive constraint programming loop`. In this framework the ML and CP component work closely together. The ML component observes the world and extracts patterns and feeds information to the CP component which solves the general CSP. The solution of the CSP is then applied to the world making the whole process a loop which repeats. The framework may be effective for online learning and working with partial information. In context of learning heuristics the inductive loop could be extended with a ML component for learning heuristics using the CSP, but also patterns from the world.

Bonfietti et al. [5] extend their previously introduced `Empirical Model Learning` (EML) method which embeds a ML model into a CP model. The extension consists of two ML classifiers: `decision trees` and `random forests`. They embed a classifier function f in CP by introducing a vector of variables $\bar{x}$ to represent the attributes and a variable y to represent the class. They then find constraints by enforcing some degree of consistency on the relation $\bar{x} = \bar{v} \bigwedge y = w \iff f(\bar{v}) = w$. This technique could perhaps be used to learn a heuristic which propagates or relaxes a CSP by learning it's constraints. Geschwender et al. [23] also build a classifier which allows for selection of the appropriate consistency algorithm for CSPs.

Lombardi et al. [30] is referred to as an important survey on boosting combinatorial problem modeling with ML. As addition to presenting current work with ML they also highlight in a systematic fashion pivotal themes and shared issues observed with regard to ML: ML input configurations visited at search time may be considerably different from those considered at training time because of the `model inaccuracy`, objective values may have more to do with approximation errors than with the actual quality of the solution due to `optimizer bias`, active learning can provide significant advantages over passive learning in terms of solution `accuracy` and `convergence`, in `non-deterministic systems` the same input example may lead to different results, and lastly when dealing with an approximate model, the practical value of finding global optima appears questionable.

2.5. Summary

Section 2.4 shows a lot of work in the field of combining ML with combinatorial problems, trying to learn heuristics for CSPs and COPs but also the addition or removal of constraints in such models. Even more so, the use of ML can also be difficult due to model inaccuracies and stochastic behaviour. However, ML is a promising field to discover more about combinatorial problems and its features.

Hence, in this work we aim to use deep heuristics in order to use ML more effectively than in related work to learn variable ordering heuristics. The features and labels that we collect for the model can result in a more informative model due to the deep heuristic functions. The stochastic behaviour of the model can be supported by search strategies as discussed in 2.1.3.

3 Deepification: learning heuristics

In this chapter we define the concept of a deep heuristic (DH) and show how they look at more levels of a search tree than standard variable ordering heuristics. Section 3.1 formalizes DHs. Section 3.2 explains how data is generated to train the machine learning model.

3.1. Deepification

First a definition of DHs is given with an example to show how they work. Afterwards we mathematically formalize deep heuristic functions and define the following heuristics: Deep Smallest (DS), Deep Maximum Regret (DR), and Deep Anti-First-Fail (DAFF).

3.1.1. Deep heuristics

Deep heuristics use a `deep heuristic function` (DHF) by inputting a set of features and using the result to select the variable with the best score (definition 1). For a search tree where the nodes represent the current variable selection and the edges are the variable selection options a DHF looks at multiple levels of the tree, iterating over multiple nodes. The DHF consists of a heuristic function which computes a score at each node. For instance, the `smallest` heuristic function would simply return the lowest value in a variable's domain, which we call the `heuristic score` for that node. The DHF could then in turn return a new `deep heuristic score` by averaging all the collected heuristic scores.

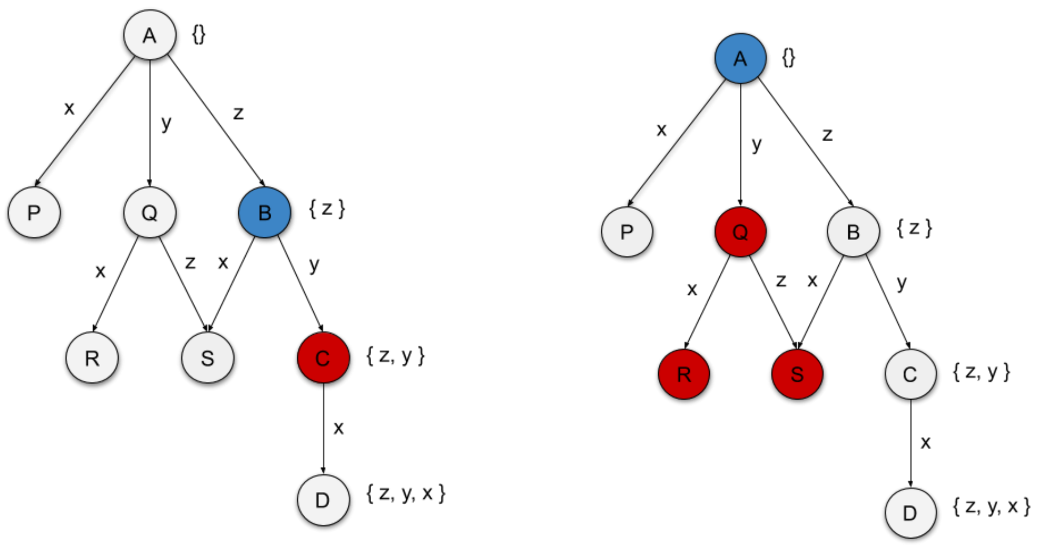
A DHF may give substantial information about many nodes compared to a heuristic score for a single node, but computing multiple heuristic scores is a costly operation during search. That is why machine learning learns a model to approximate the DHF and predicts scores based on a set of features. Figure 3.1 gives an example of what nodes may be used to compute the DHF given a depth value d.

Definition 1. A deep heuristic function is a custom designed function that uses the result of a heuristic function of multiple nodes in d levels of a search tree, given an arbitrary depth d.

Now that the DHF is defined we can select variables based on the outcomes of this function. Let us take the example search tree from figure 3.1 and select a variable at 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 DHF as defined in the next section. We call the heuristic that selects a variable this way `Deep Smallest`. The act to use a heuristic such as the smallest heuristic and define a deep heuristic with is to `deepify` the heuristic.

3.1.2. Formalization of heuristics

To deepify heuristics a DHF needs to be defined. The heuristics that we will deepify in this work are smallest, maximum regret, and anti-first-fail. They are implemented in Gecode and we can deepify them given the features that can be retrieved from Gecode. Some heuristics which use successes, failures, or constraints cannot be used as discussed in section 7.2.1. Table 3.1 defines their heuristic property.



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

(b) Computing a deep heuristic score from node A for the selection of variable y with a depth level of 2.

Figure 3.1: 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 $\{A, B, C, D\}$

Heuristic	**Heuristic attribute**
smallest	the lowest value in a variable's domain
maximum regret	the largest difference between the two smallest values in a variable's domain
anti-first-fail	the largest a variable's domain

Table 3.1: The heuristics used in this work with their heuristic properties

Heuristic function	Function output	Heuristic objective
$h_{smallest,x}$	the lowest value in D_x	minimization
$h_{max-regret,x}$	the difference between the two smallest values in D_x	maximization
$h_{anti-ff,x}$	the size of domain D_x	maximization

Table 3.2: Heuristic functions based on their respective heuristic property along with their objective

For each of the above heuristics we use a heuristic function h computing a score based on the heuristic's property. The property also determines whether we desire the lowest or highest heuristic and show this in table 3.1.

The heuristic functions are then used to define the DHFs which iterate over search nodes, averaging the heuristic scores for a particular heuristic from the function in table 3.2. A recursive function is defined to compute the deep heuristic score:

$$g(x,v,k) = \begin{cases} h_{r,x} + \sum_{y \in M} g(y,v,k+1), & \text{if } k < d. \\ 0, & \text{if } k = d. \end{cases} \tag{3.1}$$

$$H(x,v,k) = \frac{g(x,v,k)}{\text{nodes-counted}} \tag{3.2}$$

where

x	variable in the current search node
v	assigned value to the current variable
k	current level in the tree
M	set of variables of children of the current node
$\boldsymbol{h_{r,x}}$	heuristic score by selecting variable x where r is a heuristic from table 3.1
nodes-counted	total nodes for which a heuristic score is computed
d	the depth for which we compute the heuristic in the number of levels of a tree

For the assignment of values we choose the lowest value in the domain for minimization problems and the highest value for maximization problems. Value heuristics can be used instead, but in this work the variable ordering heuristics are independently tested. Dynamic variable and value ordering is proposed in section 7.3.1.

3.2. Data generation

The deep heuristics introduced in the previous section are approximated through supervised machine learning, which means training samples are needed to fit the machine learning model. At each search node we gather all features X_f along with labels y_{score} to approximate a deep heuristic function h as

$$h(X_f) = y_{score} \tag{3.3}$$

3.2.1. Features

Historical data is not available in this work because search has not started yet so we do not have any information about the search nodes and we assume that no probing or other search has been done before our search. This means we have to artificially create our own dataset. Hence, a probing phase is initiated on the problem that is being solved. This acts as a short pre-search to gather features at every search node of the search tree. This technique is also employed by Chu et al. [13] to learn value heuristics.

To obtain as many different assignments as possible a random variable order and random value assignment chosen. Regardless of which deep heuristic function we are learning, the following features at each search node are saved:

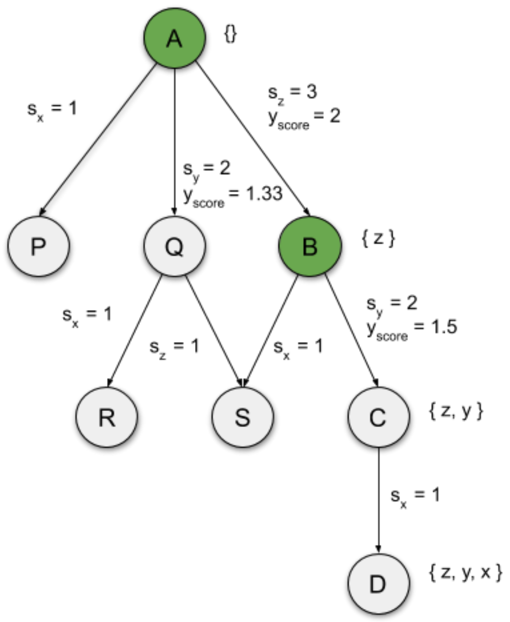


Figure 3.2: Computing labels y_{score} in the green nodes for variable choices which go at least 2 levels deep

D_v	variable domain size
D_V	total domain size of all variables
v	assigned value
v_p	value position in the domain
v_{min}	minimum value in the domain
v_{max}	maximum value in the domain
r_{min}	maximum regret as the difference between the two smallest value in the domain
r_{max}	maximum regret as the difference between the two largest value in the domain

These features are possible to collect during probing, but 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 about the structure of the search tree but cannot actually be used because while during search this is often futuristic data. For example, going through a node for the first time means the node does not have any children nodes yet, thus making that feature unreliable. Some features are not possible to collect due to limitations found in section 7.2.

3.2.2. Heuristic score

The features are acquired and the probing phase has ended. Now the labels y_{score} from equation 3.3 have to be computed. To do this, we compute for each node in the search tree that was probed the deep heuristic score.

After all features are collected, the labels y_{score} are computed by iterating over the nodes in the probed tree that have children who go at least d levels deep. Other nodes are discarded to make sure the features represent the depth. After all, if a machine learning model predicts a heuristic score for a node with a depth value of 25, then gathering features for this node which only has depth value 1 makes it unclear what the data says. Figure 3.2 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 $(s_y + s_x + s_z)/\textit{nodes counted} = (2+1+1)/3 \approx 1.33$. From node A, selecting variable z, would yield a deep heuristic score of $(s_z + s_x + s_y)/\textit{nodes counted} = (3+1+2)/3 = 1.33$. 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 2 shows the naive approach to compute the labels based on the recursive function 3.2. It could be improved by using a dynamic programming algorithm, iterating from leaves to root and memorizing computed values by reusing them.

Algorithm 2: Computing labels y_{score}

```
    $d \leftarrow$ depth value for the heuristic;
    $N \leftarrow$ nodes from the probed tree;
    $h \leftarrow$ height of the probed tree;
    $s_n \leftarrow$ heuristic score or label for node $n$;
    /* Recursive function with node $n$, depth $d$ and current depth $d_c$ */
COMPUTESCORE($n, d, d_c$)
    if $d = d_c$ then
        return $s_n$
    foreach child c of node n do
        $s_c \leftarrow$ COMPUTESCORE($c, d, d_c + 1$)
    return $s_n + s_c$

    /* Call the recursive function for each node in the tree */
    foreach node n in N do
        $k \leftarrow$ level in tree of node $n$;
        $s_n \leftarrow 0$;
        if $d \leq h - k$ then
            $s_n \leftarrow$ COMPUTESCORE($n, d, 0$) / nodes-counted;
```

4 Framework

Now that the idea of deepification has been established we have to use the deep heuristics during search such that they can be used effectively. This chapter defines a framework which is defined as a probing, machine learning, and heuristic search phase along with a search strategy to guide these phases.

4.1. Probing

Probing happens by assigning random values to a random variable order. After all variables have been assigned a value, search normally continues to mutate the current solution until an optimal, in COPs, has been found. This means that probing has the chance of finding an optimal solution during probing which would skip the necessity of starting the machine learning and search phase. However, the goal for probing is to collect data with many different variable orderings and value assignments. Thus, probing is restarted repeatedly by resetting the variables and assigned values. This happens with the use of a constant cutoff value [36] which stops search after a specific constant amount of failures.

The framework makes one exception to the random process which is the selection of the first variable after each restart. To make sure all variables are picked at least once during probing the framework keeps a list of variables which were not selected yet. The reason to do this is because it can be promising to have more data on the first variable selections, which could reduce cost of overall search as investigated by Ortiz-Bayliss et al. [34]. This way of probing also increases the chance to find a set of backdoor variables [44]: a set of variables which can significantly reduce search cost.

4.2. Machine learning

To train a machine learning model the random forest regressor is used with default Scikit-Learn parameters[1]. Support vector regression (SVR) and stochastic gradient descent (SGD) were also considered for use but showed some drawbacks. SVR has a fit time complexity which is more than quadratic with the number of samples which means it is hard to scale with a dataset of more than a couple of 10000 samples[2]. SGD on the other hand supports large-scale data, but has a lot of hyper-parameters to tune which means that the use of cross-validation, trying to set the best hyper-parameters per problem per data instance, can be costly.

A comparison has also been made in terms of prediction latency for the regressor implementations in Scikit-Learn. We want to minimize this latency because search requires a lot of predcitions being made by the ML model. One way of doing this is to make predictions in bulk rather than atomic, one prediction per call to the model. An overview of prediction latency when doing bulk predictions in figure 4.1. From this figure we can see that a linear model is extremely efficient when it comes to making bulk predictions whereas RFR and SVR more experience latency.

Considering the aforementioned comparisons, SVR is not scalable for this framework and SGD is a good candidate in terms of prediction latency but complicated to implement due to hyper-parameter tuning.

Due to the latency it is not tractable to make predictions for every feature combination. Often when the solver is backtracking values will be mutated until the solution no longer fails. However, if one would make

[1]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
[2]https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html

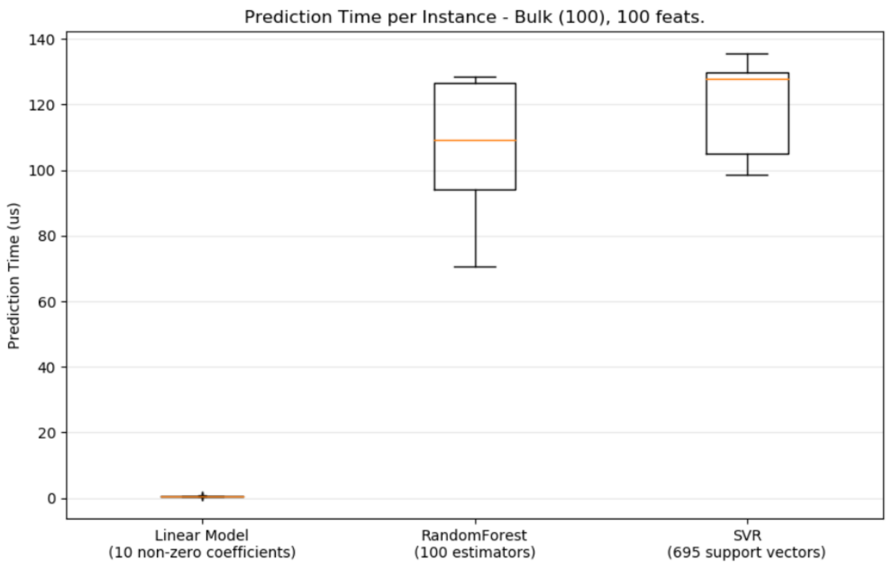


Figure 4.1: The prediction latency in Scikit-Learn when doing multiple predictions at the same time for different models[3]

a prediction each time a value is changed while the variable order remains the same then often the same resulting predictions is made. For a domain size $|D_x|$ of a variable x then the solver could make $O(|D_x|)$ predictions. Is the domain size large, then there could be a lot of prediction latency. To save runtime the framework caches predictions and uses them each time a variable ordering was already 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.

We can even go a step further and introduce a threshold t where predictions are made only for the first t variable positions in the solution. For now, caching saves enough time and this threshold is not utilized in the framework.

4.3. Heuristic search

The framework is implemented as an extension of Gecode 6.2.0 [1]. This solver allows for 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: the lowest predicted heuristic score for Deep Smallest and Deep Max Regret and the highest predicted score for Deep Anti-First Fail. Deep heuristics are not used on value selection which means that the minimum value in the chosen variable's domain is chosen for minimization problems and the maximum value for maximization problems.

4.4. Search jobs

So far we have discussed the three main phases of using a learned deep heuristic: probing, machine learning, and search. We define this as a `job` with a fixed `job time`. To get a wider variety of variable orderings and values we introduce a restart-based process which allows for multiple search jobs within the total search time given. Whenever search does not finish within the given job time then search is halted and the next job started. The data gathered by probing is not transferred between jobs as it would greatly increase the time to fit a model. Jobs restart until an arbitrary search time is reached. Figure 4.2 shows an overview of the restart-based search.

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 a lot. However, by restarting search completely we get a different predicted variable ordering due to the pseudo-

[3]https://scikit-learn.org/stable/auto_examples/applications/plot_prediction_latency.html

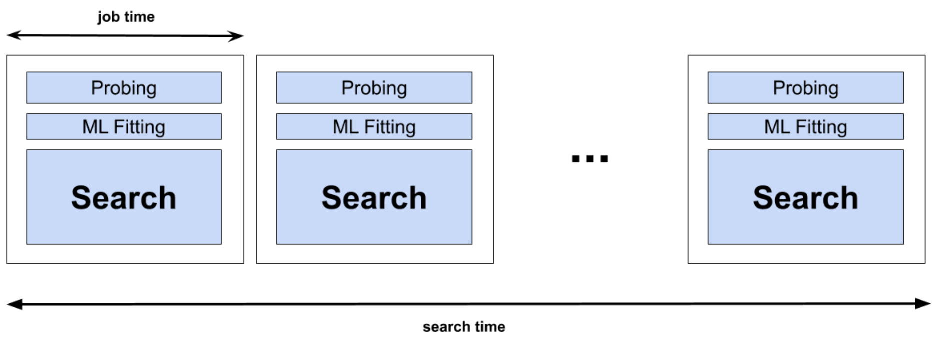


Figure 4.2: Restart-based search: independent jobs with a job time repeat until search time is reached

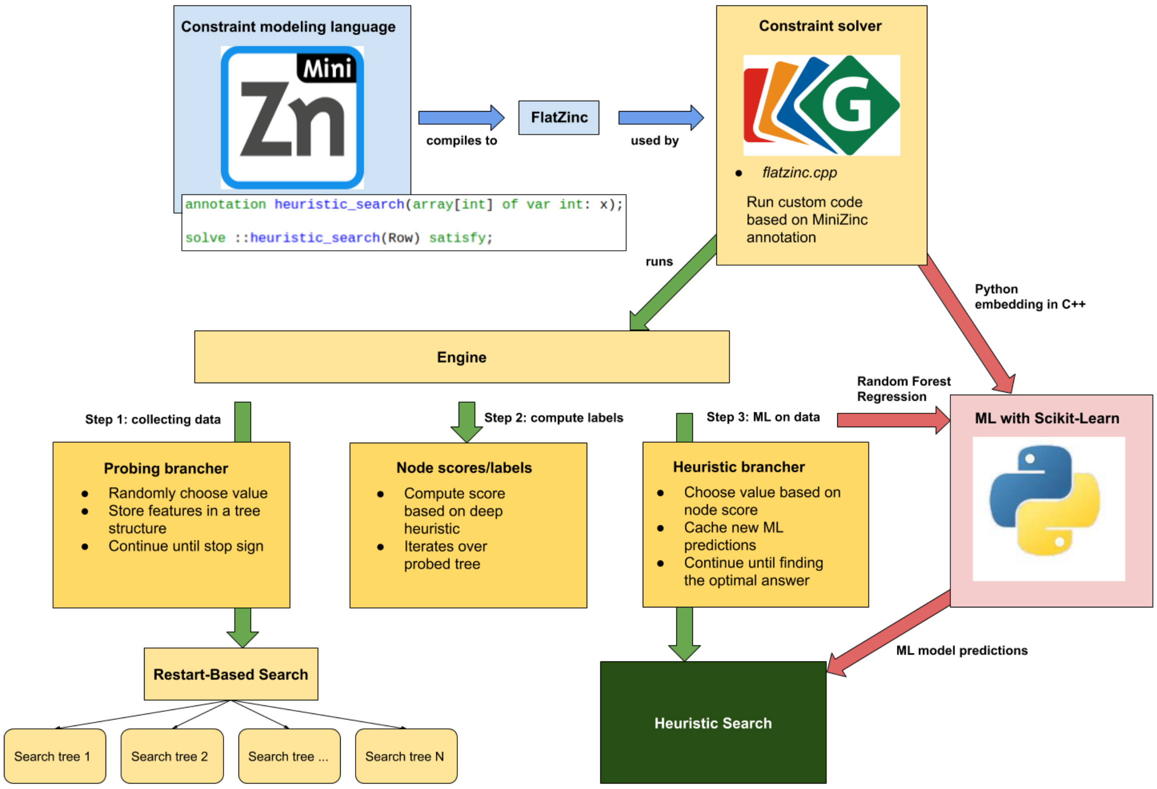
random probing.

4.5. Implementation

The framework is implemented in C++ inside Gecode which in itself can be used in the MiniZinc modelling program. Once the custom Gecode solve is added to MiniZinc one can use deep heuristics with this annotation:

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

where *q* are example decision variables used in a minimisation problem. From the source code in Gecode we call a Python script where Scikit-Learn library is used to train the ML model. Scikit-Learn is used because it is easy to use and has multiple libraries for machine learning such as different learning techniques, dataset utilities, and cross-validation. Figure 4.3 describes the steps to use deep heuristics and how it is implemented within Gecode, using MiniZinc annotations and the Scikit-Learn library.



```
annotation heuristic_search(array[int] of var int: x);

solve ::heuristic_search(Row) satisfy;
```

Figure 4.3: Implementation diagram describing the probing, machine learning, and heuristic search phase to use deep heuristics in Gecode

5 Experiment results

In this chapter the results are shown and the methodology to obtain them. Section 5.1 explains what problems are experimented on, where the data instances come from, what parameters are used, and how we measure quality. Section 5.2 shows the overall results and the results of hypotheses.

5.1. Experimental setup

5.1.1. Data set

To test the deep heuristics implemented in Gecode we will be using Minizinc. They provide a list of modelled constraint problems[1] to which we add our custom annotation as explained in section 4.5. Next to the modelled problems, Minizinc also has a huge amount of data instances[2] available. Many of these were used for the MiniZinc Challenge which is an annual competition for constraint solvers[3].

We select those data instances that run for at least a minute in Gecode with the smallest variable ordering heuristic. This is to prune the dataset of problems that are too easy to solve.

5.1.2. Problems

We test deep heuristics on the Resource Constrained Project Scheduling Problem (RCPSP), Evilshop, Amaze, and Open Stacks. They respectively include 138, 11, 13, and 43 data instances which can be found at the MiniZinc benchmarks repository[4]. Amaze and Evilshop do not contain that many instances which is why Open Stacks was added later during research. Note that we should take this in mind when inspecting the results for Amaze and Evilshop.

RCPSP The RCPSP[5] is an NP-hard problem [4] 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 [26]. 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[6] 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.

[1]https://www.minizinc.org/mznc_list_of_problems_and_globals.html
[2]https://github.com/MiniZinc/minizinc-benchmarks
[3]https://www.minizinc.org/challenge.html
[4]https://github.com/MiniZinc/minizinc-benchmarks
[5]https://github.com/MiniZinc/minizinc-benchmarks/blob/master/rcpsp/rcpsp.mzn
[6]https://github.com/MiniZinc/minizinc-benchmarks/blob/master/evilshop/evilshop.mzn

Besides that, start times are scaled up.

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

Amaze Amaze[7] 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[8] problem, also called Minimizing 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 minimize the maximum amount of open stacks that are needed which is the maximum number of customers whose products are being manufactured simultaneously. [12]

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

5.1.3. Parameter selection

All the instances are run for a maximum search time of 4 hours after which they time out and job times of 15 minutes for the deep heuristics allowing at most 16 jobs in total. This maximum makes it more feasible to experiment with the data instances as some could potentially run for a week or more given an unfitting heuristic. The small runtime window gives the flexibility of testing more data instances more frequently. A huge downside to this method is that we cannot measure exact runtime differences between Gecode and our custom solver for instances that require long search runtimes of close to an hour. To make up for this a comparison is made without looking at instances that timed-out.

Per instance we run each deep heuristic three times and take the average of their results. This is done because there may be variance in the individual results due to the random behaviour of probing.

For all problems we select a depth value of 25. The size of the decision variable set of the problems is at least 30 which means that with a depth value of 25 we make sure that we gather data for at least the first 5 levels of the search tree. More on how this works can be found in 3.2.2.

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. The last column of table 5.2 and table 5.3 show the average dataset size that is used after probing has finished.

5.1.4. Results evaluation

Firstly, the quality of deep heuristics is evaluated by comparing runtime with Gecode having a standard heuristic, i.e. smallest (SM) versus Deep Smallest (DS), max regret (MR) versus Deep Max Regret (DR), anti-first-fail (AFF) versus Deep Anti-First-Fail (DAFF). A comparison in total number of nodes cannot be made which is a limitation to the restart-based search mechanism: when a job is cancelled then Gecode does not show the statistics with the number of nodes. Next to a runtime comparison we also record the number of instances that are solved by each heuristic and the number of times a heuristic outperforms another. This is useful as we have a lot of different data instances with different solving time.

Secondly, we measure the quality of the fitted ML model with the coefficient of determination R^2, which is the proportion of the variance in the dependent variable that is predictable from the independent variable(s)[9]. It is defined as

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

[7]https://github.com/MiniZinc/minizinc-benchmarks/blob/master/amaze/amaze.mzn
[8]https://github.com/MiniZinc/minizinc-benchmarks/blob/master/open_stacks/open_stacks_01.mzn
[9]https://scikit-learn.org/stable/modules/classes.html#regression-metrics

where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value. The outcome can range from 1, an optimal prediction, to an arbitrarily large negative number.

However, the order of the predicted values for selecting variables is more important than predicting the exact true value. This is why we also add Spearman's rank correlation metric to get an idea of how well the relationship between the predicted dataset and true testset can be described using a monotonic function and is defined as

$$r_s = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

where d_i is the difference between the predicted rank and true ranking per sample. The value for r_s can range from -1 to 1 where 1 would be a perfectly predicted ranked order.

R^2 and Spearman's rank correlation are also used to evaluate the top 10 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 20% of the dataset where the other 80% is used for training to prevent bias.

5.1.5. Statistical significance

In the results we would like to make a comparison of the means of deep heuristics versus the mean of normal heuristics. For this we define the following null-hypothesis for each deep heuristics mean versus normal heuristic mean for each problem [16]:

$$H_0 : \mu_{DH} = \mu_H$$

We set the significance level to $\alpha = 0.05$ and compute the p-value for each comparison with Welch's t-test, because the compared datasets have different variances. This test is defined as

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}}$$

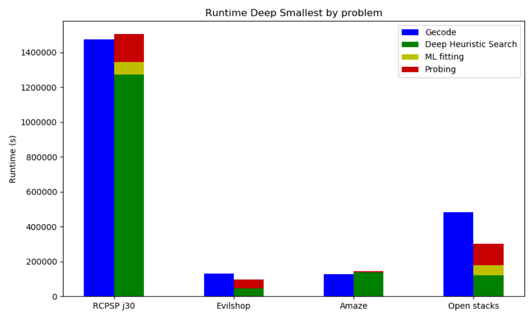
If the p-value is lower than the significance level then we reject the null-hypothesis of the two sets having the same averages. We reject this in favor of a hypothesis where the mean is either a higher average or lower average.

5.2. Results

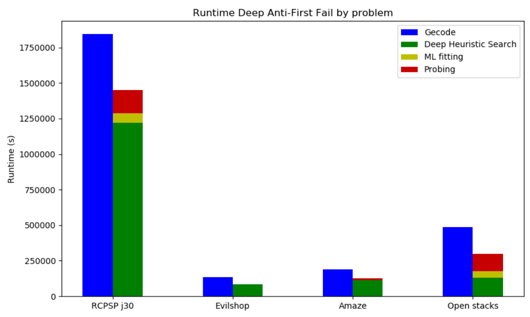
5.2.1. Overall results

To get a good sense of the overall performance of deep heuristics compared to SM, MR, and AFF we first inspect the total runtime of the heuristics on all problems. Figure 5.1 shows the total runtime of Gecode running the SM, MR, and AFF heuristics versus DS, DR, and DAFF on all problems. The runtime bars of the deep heuristics have been divided into the probing, machine learning, and search phases. DS uses 7.7% less total runtime, DAFF 26.1%, and DR uses 4.4% more total runtime than the normal heuristics. DAFF outperforms AFF on all problems, where DS and DR mostly have a higher overall runtime. However, one could argue that DAFF outperforms AFF, because of the fact that AFF performs worse on all problem relative to SM and MR. Hence, we present the average runtime of the heuristics per problem for SM, MR, and AFF in figure 5.2a. Not only do we observe that AFF indeed performes worse than the other heuristics, but also that AFF mostly timed out showing only some solved instances as outliers shown by the boxplot. We observe that MR on average outperforms the other heuristics for each problem where as DR does not as shown in figure 5.2b. It is difficult to say that a deep heuristic outperforms another due to the fact that there is a lot of variance which can be seen through the boxplots.

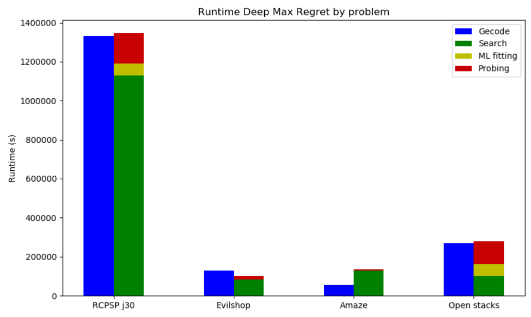
Figure 5.3 compares the total average runtime. It is evident that AFF runs worse that DAFF. These figures include timeouts, which means that the we do not know for how long these instances would have run given unlimited runtime. This is why we also make a comparison in average runtime without timed-out instances. Figure 5.4 shows that deep heuristics are outperformed in average runtime on most problems. Note that this graph does not include instances where deep heuristics outperform timed-out instances. Thus we can reason that normal heuristics perform better than the deep heuristics when they solve instance in 4 hours. Figure 5.4b misses two bars because there are no instances where neither AFF nor DAFF timeouts.



(a) SM versus DS

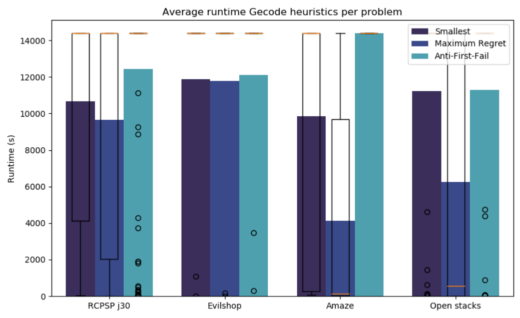


(b) AFF versus DAFF

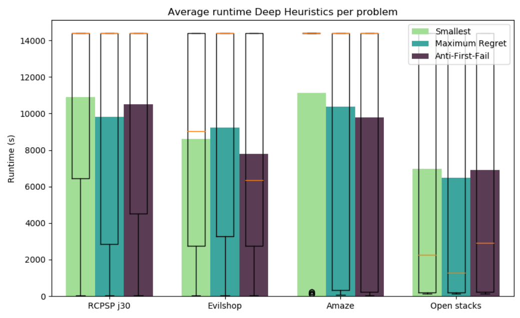


(c) MR versus DR

Figure 5.1: Total runtime divided into probing, ML, and search

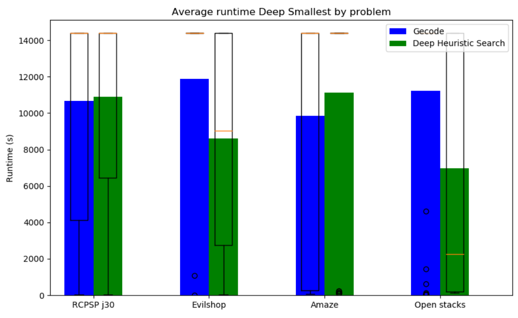


(a) Gecode heuristics

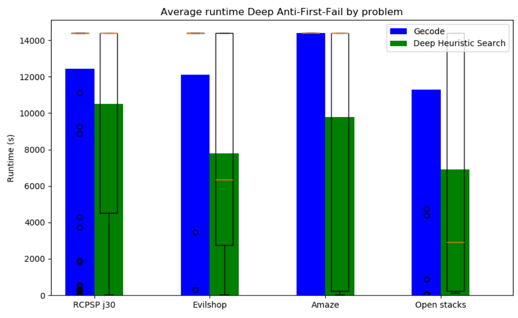


(b) Deep heuristics

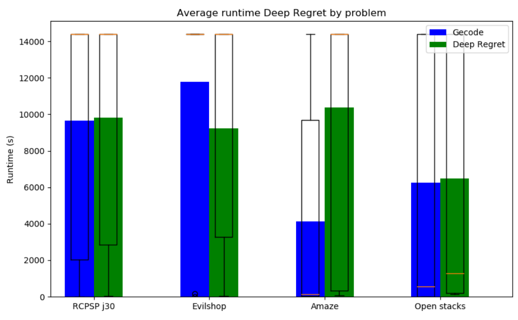
Figure 5.2: Comparison of average runtime between heuristics



(a) SM versus DS

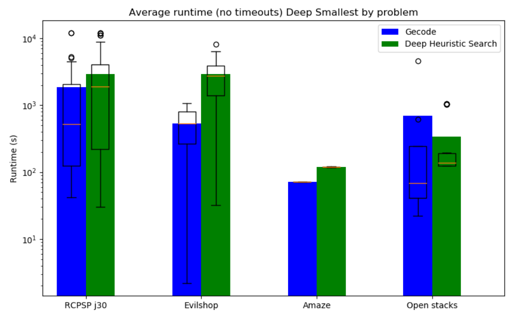


(b) AFF versus DAFF

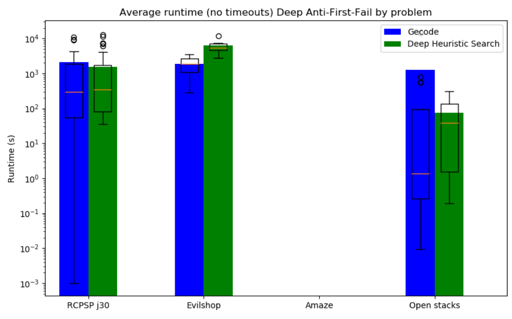


(c) MR versus DR

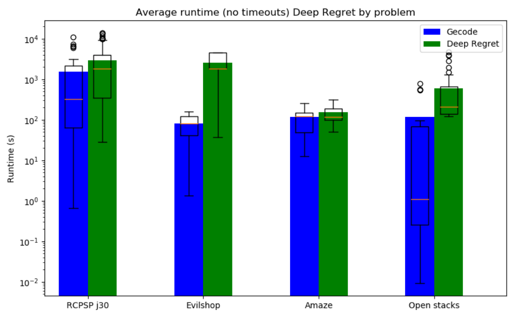
Figure 5.3: Comparison of average runtime between heuristics



(a) SM versus DS



(b) AFF versus DAFF

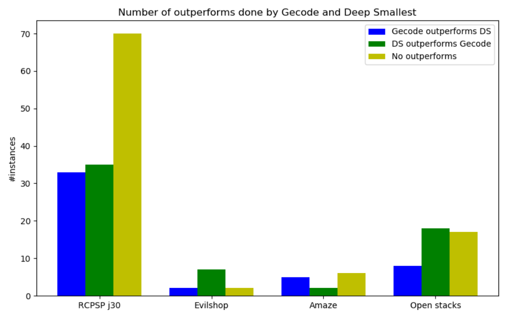


(c) MR versus DR

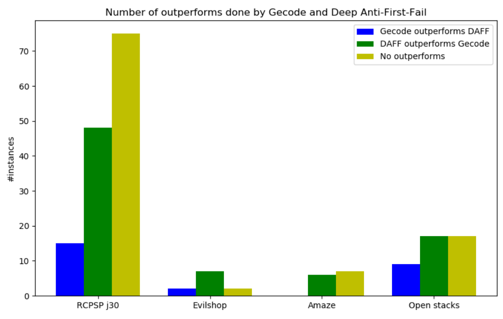
Figure 5.4: Comparison of average runtime without timed-out instances

%instances solved	RCPSP j30	Evilshop	Amaze	Open Stacks
Gecode Smallest	29.7%	54.6%	38.5%	23.26%
Deep Smallest	31.2%	18.2%	23.1%	54.26%
Gecode Anti-First Fail	15.9%	18.2%	18.2%	23.26%
Deep Anti-First Fail	35.4%	63.6%	63.6%	56.59%
Gecode Max Regret	37.7%	18.2%	76.9%	57.36%
Deep Max Regret	41.6%	48.5%	28.2%	58.14%

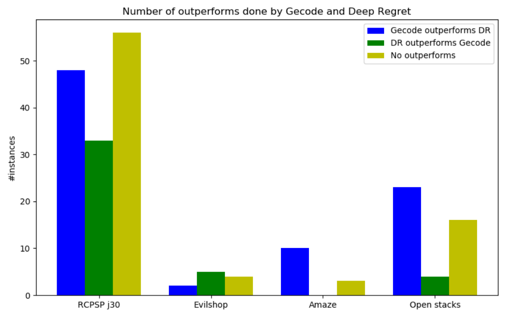
Table 5.1: Percentage of total instances solved by heuristics



(a) SM versus DS



(b) AFF versus DAFF



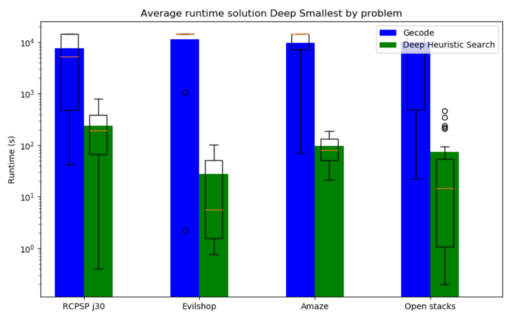
(c) MR versus DR

Figure 5.5: Number of times that heuristics outperform each other

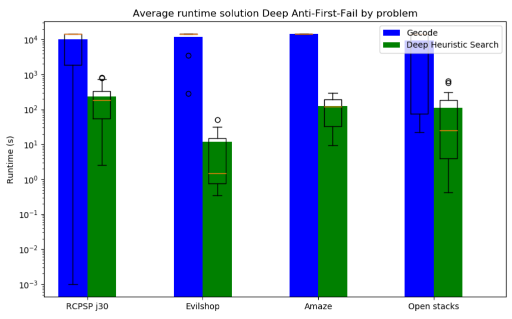
Table 5.1 presents the percentage of instances solved per problem by heuristics. The percentages in the table for deep heuristics are based on all runs as deep heuristics are run three times per instance. This means that RCPSP has 414 instances, Evilshop 33 instances, Amaze 39 instances, and Open Stacks 129 instances. We can also say that we count a success when at least one out of three runs would solve the instance within 4 hours. In this case for all problems, DS outperforms SM by 38 (18.5%) instances, DR outperforms MR by 21 (10.2%) instances, and DAFF outperforms AFF by 67 (32.7%) instances. This way in total deep heuristics solve 126 (20.5%) more instances than normal heuristics. In the table the deep heuristics outperform normal heuristics as they are able to solve more problems within 4 hours. Only in the Amaze problem DS and DR and Evilshop DS is outperformed. It is difficult to reason why because the number of instances of those problems is low, but DAFF solves more problems than DS and DR for them.

Another point of view is to look at the number of instances in which heuristics outperform another which is shown in figure 5.5. Notable is that DS and DAFF outperform SM and AFF in more instances and that 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.

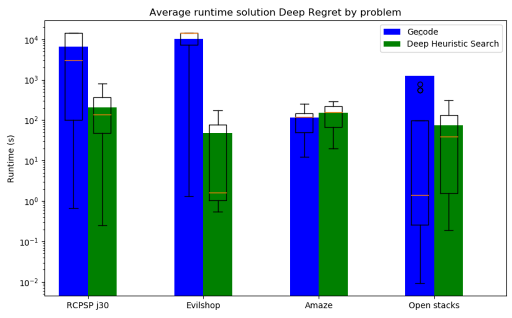
If a deep heuristic search completes within 4 hours then one of the search jobs in the framework succeeded. We denote these successes as 'solutions'. The solutions are independent of other search jobs and hence we compare their runtimes in figure 5.6. It can be said that the solutions have significant less runtime than the normal heuristics which can be partly explained by the fact that search jobs get a maximum runtime



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Figure 5.6: Comparison in runtime between normal heuristics and deep heuristic solutions

of 15 minutes.

Lastly, table 5.2 and 5.3 show the average quality of the fitted model for deep heuristics and solutions. It can be noticed that for the R^2 values the models have a really high error margin. For the R^2 values of the top 10 ranked values this goes to well beyond -1 showing that the predicted value is often not even close to the one in the test set. The Spearman's rank correlation shows that values are mostly around the 0 mark and sometimes above which indicates that the ranking quality of the system is not great, but not that bad either. Note that the true ranking is based on only 20% of the dataset which means that data for early decision making could be located in the other 80%. We also present the true ranks of predicted values. In general we see that the first three predicted ranks are pretty close to the true first rank based on the average size of the dataset. For example, for DS RCPSP the 1st predicted rank has as true 1st rank 706 which is $706/41730 \cdot 100 = 1.7\%$ from the top rank. For DR in Evilshop the metrics could not be recorded due to an unknown implementation reason.

Finally we perform the statistical significance test as found in table 5.4 for the average runtime without timeouts, table 5.5 for the average runtime with timeouts, and table 5.6 for the average runtime of solutions. Note that most p-values for the average runtime without timeouts in table 5.4 are higher than the significance level α and some p-values for Amaze are missing, possibly due to the low amount of samples.

The overall results presented so far give an impression of the quality of deep heuristics. However, it is hard to test the quality of the ML model and it is unclear how different parts of the framework contribute towards performance. To further test this and what parameters could fit best we define three hypotheses:

- **Hypothesis 1** *Halving the probing time results in a lower total runtime.* Figure 5.3 showed the division of probing, ML fitting time, and actual search. Reducing probing time could possibly reduce the total runtime without potentially increasing heuristic search time. A side effect is that the ML fitting time is also reduced, because less data is gathered. For the RCPSP, Amaze, and Evilshop problems we reduce probing time from 1 million nodes to half a million nodes and Open Stacks from 2 minutes to 1 minute. There are a number of reasons to halve the probing. Firstly, if we would halve probing time and only the probing time will change to total runtime than we would outperform normal heuristics in any problem. Secondly, if we reduce probing time by too much then we would have possibly to little data for the ML model. Finally, it can be hard to draw conclusions by lowering probing time only by a little. This is due

average metrics	R^2	R^2@10	Spearman	Spearman @ 10	Rank of 1st predicted value	Rank of 2nd predicted value	Rank of 3rd predicted value	Dataset size
DS RCPSP	0.19	-31787.46	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	-35904.47	-0.02	0.16	13	13	16	385
DS Open stacks	0.05	-28957.35	0.29	0.02	10361	9474	10258	143417
DAFF Fail RCPSP	0.39	-13701.66	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	-7148.79	-0.04	0.15	16	18	21	1198
DAFF Open stacks	-0.02	-35837.92	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 5.2: Average quality of deep heuristics presented through R^2, Spearman's rank correlation, and predicted rankings

solution metrics	R^2	R^2@10	Spearman	Spearman @ 10	Rank of 1st predicted value	Rank of 2nd predicted value	Rank of 3rd predicted value	Dataset size
DS RCPSP	0.27	-35713.61	0.50	0.04	845	1185	1343	65409
DS Evilshop	0.26	-37.16	0.15	0.38	11	16	16	218
DS Amaze	-7.09	-67.34	-0.15	0.68	14	12	13	113
DS Open stacks	-0.05	-22499.85	0.18	0.04	7506	6424	7497	94663
DAFF RCPSP	0.44	-20307.12	0.65	0.06	2878	3140	2866	68747
DAFF Evilshop	-0.03	-44.12	0.25	0.11	15	20	19	224
DAFF Amaze	-0.90	-222.80	-0.15	0.43	10	17	17	198
DAFF Open stacks	-0.07	-20591.68	0.15	0.02	2944	2786	2702	55869
DR RCPSP	-0.13	0.00	0.02	-0.02	1199	1092	586	63398
DR Evilshop	err	err	err	err	err	err	err	213
DR Amaze	-0.39	-0.19	nan	0.38	14	19	29	227
DR Open stacks	-0.16	0.00	0.05	0.00	3504	3929	4205	86904

Table 5.3: Average quality of deep heuristic solutions presented through R^2, Spearman's rank correlation, and predicted rankings

Mean average runtime (no timeouts)	p-value	Significant difference
DS vs SM: RCPSP	0.12	No
DR vs MR: RCPSP	0.014	Yes
DAFF vs AFF: RCPSP	0.45	No
DS vs SM: Evilshop	0.19	No
DR vs MR: Evilshop	0.15	No
DAFF vs AFF: Evilshop	0.13	No
DS vs SM: Amaze	-	No
DR vs MR: Amaze	0.45	No
DAFF vs AFF: Amaze	-	No
DS vs SM: OS	0.33	No
DR vs MR: OS	0.016	Yes
DAFF vs AFF: OS	0.14	No

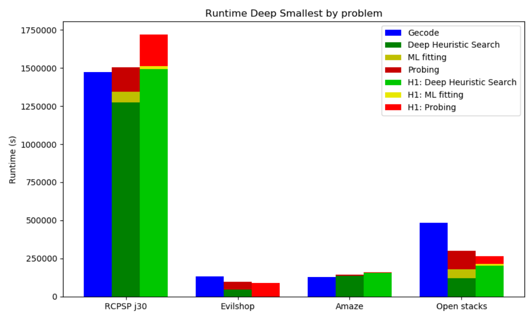
Table 5.4: Significance of mean comparison average runtime without timeouts, comparing the p-value to $\alpha = 0.05$

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

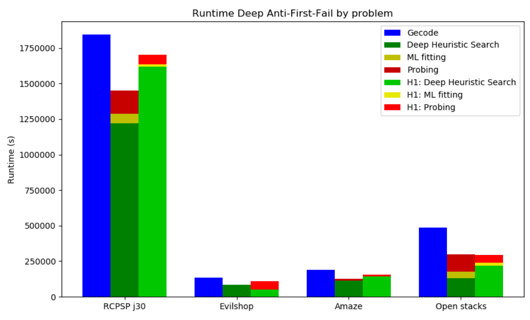
Table 5.5: Significance of mean comparison average runtime with timeouts, comparing the p-value to $\alpha = 0.05$

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

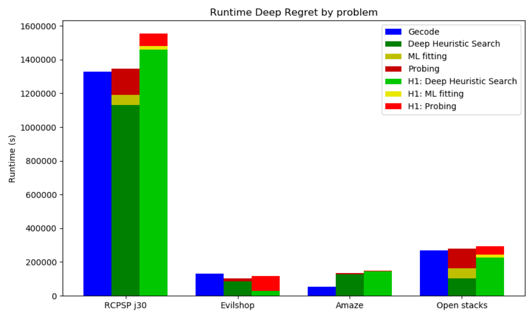
Table 5.6: Significance of mean comparison average solution runtime with timeouts, comparing the p-value to $\alpha = 0.05$



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Figure 5.7: Hypothesis 1: Total runtime divided into probing, ML, and search phase

to the variance of the results: a small change in total runtime would not be possible to fully relate to the small change in probing time.

- **Hypothesis 2** *Halving the job time leads to more instances solved.* Figure 5.6 shows that the mean of the average solution runtime is usually below 200 seconds which means we can test the system by reducing the job time, which is defined in section 4.4, from 900 to 450 seconds. A lower job time will increase the total number of jobs for search from 16 to 32 and should thus have more chance of finding a solution in one of the jobs.

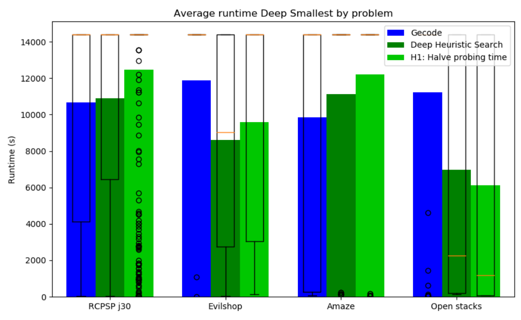
- **Hypothesis 3** *Reducing the depth value by 5 will decrease the average runtime of the framework.* First of all, reducing depth will increase the amount of data gathered by including more feasible nodes as explained in section 3.1. This will increase the ML fitting time, but gather more data about nodes further away from the root. Secondly, using less depth may lead to finding less promising nodes at deep levels. We reduce depth by 5 too see whether such a small value change has any impact on the average runtime while a larger decrease in depth can result in really long probing and ML fitting time.

5.2.2. Hypothesis 1: halving probing time

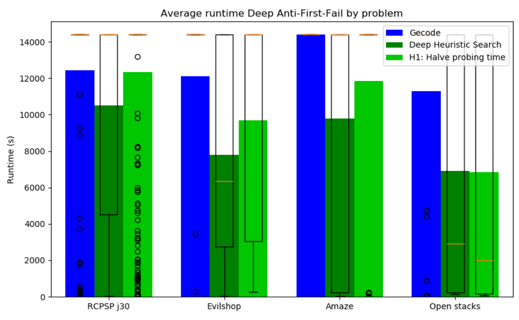
We present the total runtime in figure 5.7 and average runtime including timed-out searches in figure 5.8. Runtime has increased for most problems, with a respective total increase of 9.1%, 15.2%, and 13.7% for DS, DAFF, and DR. We can see that the probing time is sometimes larger even though probing 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. This is also pointed out by the increase in average runtime: respectively 7.4%, 16.5%, and 12.7% for DS, DAFF, and DR. Thus, for these particular parameters, halving probing time does not improve overall runtime.

5.2.3. Hypothesis 2: halving job time

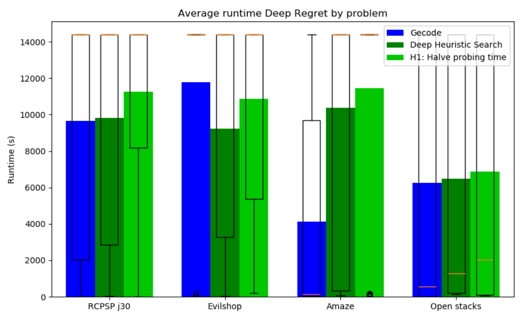
Table 5.7 presents the number of instances solved when we halve the job time from 900 to 450 seconds. If we compare the results with the ones from table 5.1 then it can be noticed that halving the job time leads to



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Figure 5.8: Hypothesis 1: Average runtime of normal heuristics versus deep heuristics

%instances solved	RCPSP j30	Evilshop	Amaze	Open stacks
Deep Smallest	26.57%	9.09%	20.51%	56.59%
Deep Anti-First Fail	34.78%	15.38%	28.21%	44.19%
Deep Max Regret	35.51%	15.15%	25.64%	49.61%

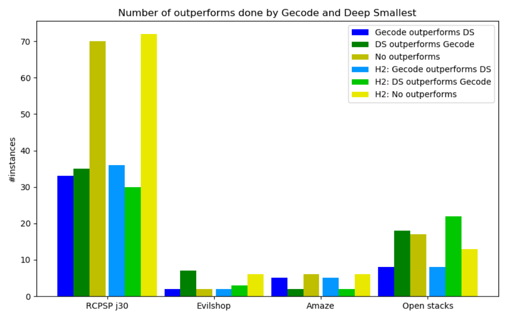
Table 5.7: Percentage of total instances solved

a considerable decrease of the number of instances that are solved. That means that we do not profit from the increase in the number of search jobs that can be performed. Instead, there is simply not enough time to solve certain data instances within 450 seconds.

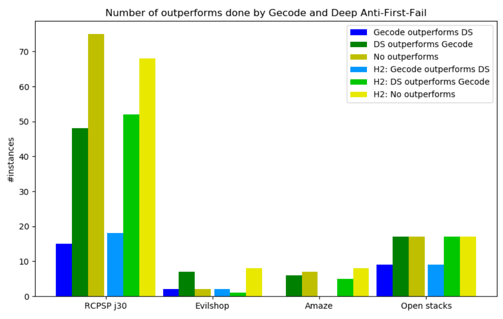
Figure 5.9 shows the number of times normal heuristics and deep heuristics outperform one another. Notable is that SM outperforms DS in more instances but DAFF and DR show an improvement in number of outperforms. There is a slight improvement for the open stacks problem and no improvement for the Amaze and Evilshop problem.

5.2.4. Hypothesis 3: reducing depth

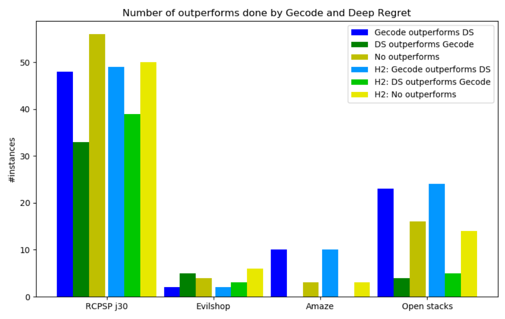
Figure 5.10 presents the average runtime of the heuristics figure 5.11 the average runtime of solutions. Reducing depth results in an overall increase of 1.3% average runtime and 21.4% decrease in average solution runtime. Through inspection we see that DHs with a lower depth value perform worse on RCPSP 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 to DR on RCPSP we see a large increase in average runtime. Figure 5.11c misses a bar because there are no solutions without timeouts, again indicating the decreased performance. Overall, reducing depth does not seem to positively affect runtime, possibly indicating that DHs need a high depth value to find promising solutions.



(a) SM versus DS

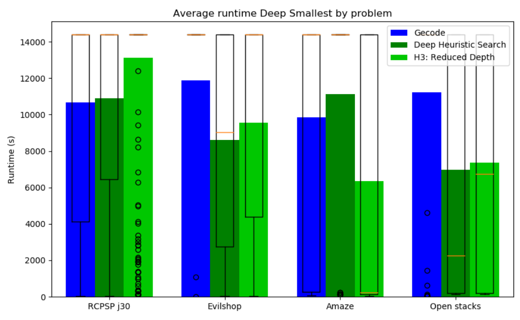


(b) AFF versus DAFF

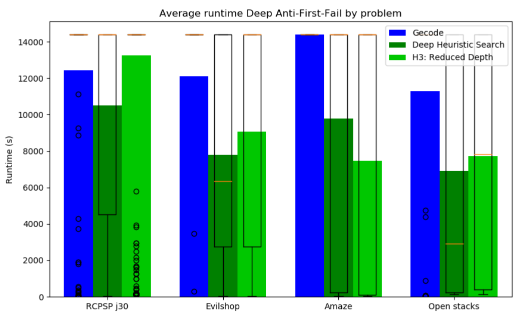


(c) MR versus DR

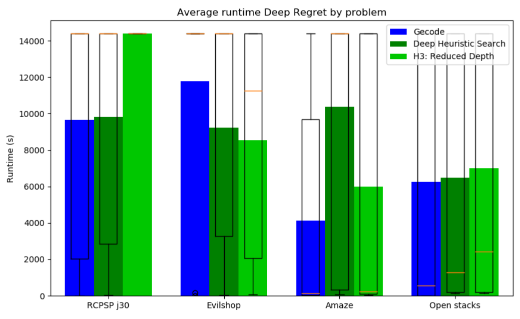
Figure 5.9: Hypothesis 2: Outperforms by heuristics versus deep heuristics



(a) SM versus DS

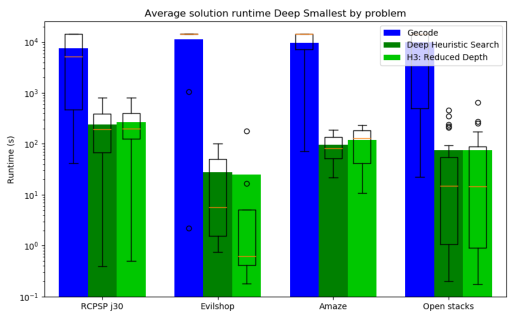


(b) AFF versus DAFF

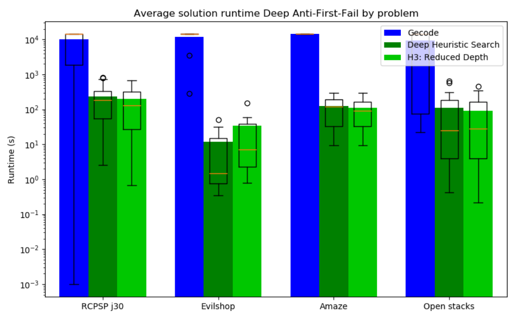


(c) MR versus DR

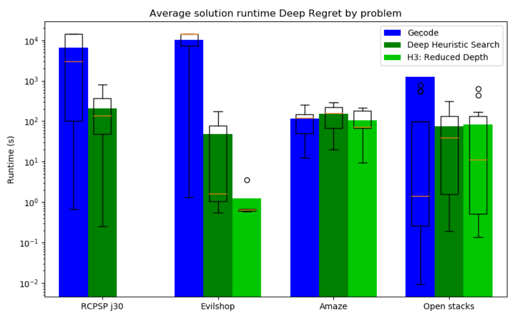
Figure 5.10: Hypothesis 3: Average runtime of normal heuristics and deep heuristics



(a) SM versus DS



(b) AFF versus DAFF



(c) MR versus DR

Figure 5.11: Hypothesis 3: Average runtime of normal heuristics and deep heuristics solutions

5.2.5. Summary

In this chapter we have compared SM, MR, and AFF respectively against DS, DR, and DAFF by comparing number of instances solved, overall runtime, average runtime, average runtime of solutions, number of instances that heuristics outperform each other, and quality of the machine learning model through R^2 and Spearman's rank correlation. Overall deep heuristics solve more instances while overall runtime is worse, but the conclusion differs for each problem-heuristic combination. The model's prediction accuracy does not show good quality in terms of R^2 and Spearman's and it is unclear how these metrics can give a good description of the system's performance. Hence, three hypotheses were defined which aim to show the performance of a parameter selection different to the one defined in section 5.1.3. Firstly, halving probing time results in a higher total runtime, because the framework does not gather enough data and needs more search jobs to find solutions. Secondly, halving job time does not result in more instances solved due to the doubling of search jobs. Thirdly, a decreased depth value, to gather more data, does not decrease overall average runtime.

6 Discussion

In the previous sections we have presented a framework to learn deep variable ordering heuristics and tested their performance. In this chapter we discuss the different parts of the framework along with the results.

6.1. Probing

In chapter 3 and 4 we described the pseudo-random probing process by randomly selecting variables and values. However, there are many different ways to guide probing. One way is to use different heuristics which do not select randomly, but for instance aim to gather data based on the deep heuristic that we want to learn. For example, the smallest heuristic could be used to gather data for Deep Smallest. However, the risk of using a heuristic during probing could be its bias. After all, the deep heuristic should be able to look much deeper into a tree at at nodes which a normal heuristic would perhaps not reach. A benefit is that the heuristic can find a solution before probing finishes.

Another way or probing would be use multiple different heuristics. This would benefit not only the variety of data, but also the ability for the solver to find a solution during probing. In a way this probing method can function as a hyper-heuristic [9], selecting the best deep heuristic to learn based on the data gathered by different heuristics. For instance, if a certain heuristic provides better scores in comparison to other heuristics then it may be a good idea to deepify that heuristic.

6.2. Machine learning quality

In machine learning we can often assess the system by the model's accuracy. However, as seen by the results in chapter 5, the quality of the model is not descriptive relative to the fact hat we observe an improvements in runtime on Gecode's heuristics for some deep heuristics. This can be partially explained by the fact that the model does not have a relation to underlying combinatorial problem, but instead functions as a proxy that in a later phase of the system returns an optimal result. The predict+optimise problem is an example of trying to overcome this problem by letting machine learning and combinatorial optimization interact with each other as [17, 20]. Another research field in ML is feature selection [11, 39]. In this work a fixed set of features was chosen, but a discussion remains on whether to dynamically select features or not and what kind of features to select.

6.3. Heuristic functions

In chapter 3 we formalized deep heuristics by averaging heuristic scores. However, one can argue that the deep heuristic score function may 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 [34] or should we instead of averaging simply take the minimum or maximum heuristic score over all nodes?

6.4. Parameter selection

Three hypotheses were tested in chapter 5, but there are still many other possible parameter choices. The hypotheses test only the change of a single parameter. This can result in finding a local optimum for that parameter, but the combination of changing multiple parameters could still find a global optimum. Even more so, changing parameters improves performance for one deep heuristic or problem but decreases performance for another. This means that parameter selection is difficult and perhaps the parameters of the framework should only be tuned after selecting the right heuristic for the problem. A particular extension of the framework would be to use a hyper-heuristic [9] which selects the best heuristic.

Another difficulty with parameter selection is choosing the job time. If a search problem can never be solved within 30 minutes then a job time below 30 minutes could could never solve the problem. Perhaps the job time can depend on the meta-data of the problem, for instance the number of variables or constraints, to estimate the solve runtime.

7 Conclusion

This chapter concludes the thesis by first answering the research questions defined in section 1.2. Then the limitations of this project are discussed. Finally, potential future research is suggested to further improve and build upon the idea of deep heuristics.

7.1. Answers to research questions

The main research question is answered by first answering the sub-questions:

1. **How to collect data to fit a machine learning model in constraint optimization problems?**
   When there is not enough historical data available one can introduce a probing phase as explained in section 3.2 which gathers the features of search nodes of the search tree. To get a wide variety of data, probing uses pseudo-random value and variable selection. Once probing has made a complete variable ordering we restart the probing with a constant cutoff value to gather a new selection of variables and values. At the start of each probing search we remember which variables were assigned first. This way we assign each variable at least once to start the solution with.

2. **How can the fit of a machine learning model to learn heuristics be improved in constraint optimization problems?**
   In section 3 we formalized deep heuristics which look multiple levels into a search tree. In combination with the pseudo-random probing to gather data this may improve the fit of the model. The learned heuristics did not show good quality scores by using standard regression metrics, but we can see an overall improvement over Gecode in the number of instances that deep heuristics solve within a 4 hour time limit. For specific problem deep heuristic combinations the runtime is also improved over Gecode heuristics.

3. **How can a search strategy improve search in constraint optimization problems when using machine learning?**
   The data that is gathered during probing provides a snapshot of the overall search tree by creating random assignments. If the probing is done only once then there is no guarantee that promising nodes with the best heuristic scores have been collected. This is where restart-based search can improve search as described in section 4.4. With this multiple searches are tried with independent probing phases each time having a different dataset.

The main research question is

> **How can machine learning improve variable ordering heuristics for constraint optimization problems?**

Variable ordering heuristics can be improved by learning a deep heuristic function. Deep heuristics use predictions from the learned function to define a score per variable ordering with the aim for to generalize better than normal heuristics. The deep heuristics outperform normal heuristics by solving 20.5% more instances and improve on total runtime for the Open Stacks and Evilshop problems.

7.2. Limitations

7.2.1. Data generation

Gecode uses propagation to remove redundant values and will automatically skip the branching step if it is not required. For example, if variable x has a domain $D_x = \{2\}$ then the value 2 is automatically picked. This means that that the probing process explained in chapter 3 cannot add all nodes that Gecode iterates over as data to the data tree.

Another probing limitation is the recording of successes, failures, and added or removed constraints. By inspection of the codebase and documentation of Gecode it was not possible to find out how this information is accessed. However, this kind of data may prove to be useful as features for any deep heuristic, but mostly to able to deepify heuristics like `domwdeg` which uses the number of times that a variable has been in a constraint that failed.

7.2.2. Parallelization

During probing we gather data by adding nodes to a tree structure. Due to the current implementation, parallelization of this process leads to concurrent exceptions because of the addition of multiple nodes to the same tree. A possible workaround could be to create multiple trees at the same time and for each of the trees learn its own ML model.

7.3. Future work

7.3.1. Dynamic variable & value ordering

In this work we have used a fixed value heuristic, the minimum or maximum value of a domain. However, we can also implement value selection with deep heuristics. Learning the value selection heuristic should be almost the same as with variable ordering. But instead of a fixed variable ordering, for example choosing variables in *input order*, we can also design a dynamic approach. For instance, the deep heuristic function that is learned can include both variable and value chosen. This way variable and value selection depend on each other and machine learning should try generalizing on assigning both. Cox et al. [14] show that picking a variable-value pair instead of just variables gains more insight on which path to search on. Thus, dynamic variable and value ordering may allow for a different heuristic approaches.

7.3.2. Variable types

The current implementation only allows for integer decision variables. Gecode supports boolean and floating point types and deep heuristics could show different results then with the use of integer decision variables. This way future research can also compare the framework to mixed integer linear programming (MILP) and non-linear programming (NLP) solvers.

7.3.3. Deep learning

The regression technique used in this work is very fast in fitting a machine learning model and provides an almost online learning setting. However, feature selection may still prove difficult as it is unclear what features work best on deep heuristics. Deep Learning (DL) would try to learn the features on its own in an offline setting an it would be interesting to see whether a DL model could for instance solve more instances, but also perform well over multiple problems.

7.3.4. Towards online learning

The current framework uses a restart-based search mechanic where new data is gathered at every restart without saving data from the previous search jobs. A promising feature to the framework may be to allow for the total dataset to grow over time. This means a lot of data will be gathered from different problems and data instances which brings its own difficulties. Stochastic Gradient Descent may be an interesting scalable option for this.

Bibliography

[1] *Gecode: open source C++ toolkit for developing constraint-based systems and applications.* URL `https://www.gecode.org/`.

[2] Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, 2016. URL `http://arxiv.org/abs/1611.09940`.

[3] C. Bessiere, L. De Raedt, T. Guns, L. Kotthoff, M. Nanni, S. Nijssen, B. O'Sullivan, A. Paparrizou, D. Pedreschi, and H. Simonis. The inductive constraint programming loop. *IEEE Intelligent Systems*, 32(5): 44–52, 2017.

[4] Jacek Blazewicz, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discret. Appl. Math.*, 5(1):11–24, 1983. doi: 10.1016/0166-218X(83) 90012-4.

[5] Alessio Bonfietti, Michele Lombardi, and Michela Milano. Embedding decision trees and random forests in constraint programming. In *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, pages 74–90, 2015.

[6] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, page 146–150, 2004.

[7] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001. doi: 10.1023/A:1010933404324.

[8] R. Brunetto and O. Trunda. Deep heuristic-learning in the rubik's cube domain: An experimental evaluation. volume 1885, pages 57–64, 2017.

[9] E.K. Burke, M.R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J.R. Woodward. A classification of hyper-heuristic approaches: Revisited. *International Series in Operations Research and Management Science*, 272:453–477, 2019.

[10] E.K. Burke, M.R. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J.R. Woodward. A classification of hyper-heuristic approaches: Revisited. *International Series in Operations Research and Management Science*, 272:453–477, 2019.

[11] Girish Chandrashekar and Ferat Sahin. A survey on feature selection methods. *Comput. Electr. Eng.*, 40 (1):16–28, 2014. doi: 10.1016/j.compeleceng.2013.11.024.

[12] Geoffrey Chu and Peter J. Stuckey. Minimizing the maximum number of open stacks by customer search. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2009.

[13] Geoffrey Chu and Peter J. Stuckey. Learning value heuristics for constraint programming. In *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, page 108–123, 2015.

[14] 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 Artif.*, 22(63):1–15, 2019.

[15] H. Dai, E.B. Khalil, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. volume 2017, pages 6349–6359, 2017.

[16] F.M. Dekking, C. Kraaikamp, H.P. Lopuhaa, and L.E. Meester. *A Modern Introduction To Probability And Statistics*. Springer London Ltd., 2005.

[17] Emir Demirovic, Peter J. Stuckey, Tias Guns, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, and Jeffrey Chan. Dynamic programming for predict+optimise. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 1444–1451. AAAI Press, 2020.

[18] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau. Learning heuristics for the tsp by policy gradient. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10848:170–181, 2018.

[19] V. Dewanto. Learning the search heuristic for combined task and motion planning. pages 309–316, 2016.

[20] Priya L. Donti, J. Zico Kolter, and Brandon Amos. Task-based end-to-end model learning in stochastic optimization. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5484–5494, 2017.

[21] A. Galassi, M. Lombardi, P. Mello, and M. Milano. Model agnostic solution of csps via deep learning: A preliminary study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10848:254–262, 2018.

[22] 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 Eugene C. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 1996.

[23] D.J. Geschwender, S. Karakashian, R.J. Woodward, B.Y. Choueiry, and S.D. Scott. Selecting the appropriate consistency algorithm for csps using machine learning classifiers. pages 1611–1612, 2013.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[25] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980.

[26] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *Eur. J. Oper. Res.*, 207(1):1–14, 2010. doi: 10.1016/j.ejor.2009.11.005.

[27] Tin Kam Ho. Random decision forests. In *Third International Conference on Document Analysis and Recognition, ICDAR 1995, August 14 - 15, 1995, Montreal, Canada. Volume I*, pages 278–282. IEEE Computer Society, 1995. doi: 10.1109/ICDAR.1995.598994.

[28] André Hottung, Shunji Tanaka, and Kevin Tierney. Deep learning assisted heuristic tree search for the container pre-marshalling problem. *CoRR*, 0, 2017. URL `http://arxiv.org/abs/1709.09972`.

[29] W. W. M. Kool and M. Welling. Attention solves your tsp. *CoRR*, 2018. URL `http://arxiv.org/abs/1803.08475`.

[30] M. Lombardi and M. Milano. Boosting combinatorial problem modeling with machine learning. volume 2018, pages 5472–5478, 2018.

[31] G. Loomes and R. Sugden. Regret theory: an alternative theory of rational choice under uncertainty. *The economic journal*, 92(368), 1982.

[32] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

[33] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. doi: 10.1007/978-3-540-74970-7_38.

[34] 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. *Comput. Intell. Neurosci.*, 2018:6103726:1–6103726:14, 2018. doi: 10.1155/2018/6103726.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[36] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4.

[37] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010. ISBN 978-0-13-207148-2.

[38] Bernhard Schölkopf and Alexander Johannes Smola. *Learning with Kernels: support vector machines, regularization, optimization, and beyond*. Adaptive computation and machine learning series. MIT Press, 2002. ISBN 9780262194754.

[39] Razieh Sheikhpour, Mehdi Agha Sarram, Sajjad Gharaghani, and Mohammad Ali Zare Chahooki. A survey on semi-supervised feature selection methods. *Pattern Recognit.*, 64:141–158, 2017. doi: 10.1016/j.patcog.2016.11.003.

[40] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

[41] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, and Thore Graepel. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419), 2018.

[42] Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning variable ordering heuristics for solving constraint satisfaction problems. *CoRR*, abs/1912.10762, 2019. URL `http://arxiv.org/abs/1912.10762`.

[43] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998. ISBN 978-0-262-19398-6.

[44] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1173–1178. Morgan Kaufmann, 2003.

[45] H. Xu, S. Koenig, and T.K. Satish Kumar. Towards effective deep learning for constraint satisfaction problems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11008:588–597, 2018.