Prediction-Based Adaptive Variable Ordering Heuristics for Constraint Satisfaction Problems

Jitao Xu[1,2], Yaling Wu[1], Hongbo Li[1*], Minghao Yin[1]

[1]College of Information Science and Technology, Northeast Normal Univerisity, Changchun, China.
[2]Department of Computer Science, Old Dominion University, Norfolk, Virginia 23508.
{lihb905, ymh}@nenu.edu.cn

Abstract

Variable ordering heuristics (VOH) play a central role in solving Constraint Satisfaction Problems (CSP). The performance of different VOHs may vary greatly when solving the same CSP instance, so identifying an efficient candidate VOH for a given CSP has been a key issue in the community. In this study, we propose a prediction-based approach to adaptively select efficient VOHs for different CSPs from a set of candidates. Our work demonstrates that efficient candidate VOHs can be identified by learning from the topology of search trees. Specifically, we propose to represent the topology of a binary search tree by the sequence of the *Numbers of Positive Decisions (NPD)* made before each failure occurs. Based on the representation, we predict the total failure number of a search tree from its beginning part. When solving a CSP, we run a probing procedure to obtain the *NPD* sequences generated by candidate VOHs and select an efficient one for the resolution according to the prediction results. Our experiments show that the Long Short Term Memory model and Gradient Boosting Decision Tree models trained with the search trees sampled from easy instances are effective in identifying efficient VOHs for hard instances. The models capture some common structure properties hidden in the search trees of different problems. Our approach outperforms the state-of-the-art adaptive VOHs in terms of the number of solved instances and the PAR2 score of runtime.

Introduction

Constraint Satisfaction Problem (CSP) is one of the foundations of Artificial Intelligence. Many real-world problems can be modeled with CSP, such as Resource Allocation, Scheduling, Vehicle Routing, Configuration, etc. A CSP is NP-hard in its general form. The challenge in a CSP is to find an assignment of values to all variables that satisfies the constraints defined over the variables, or otherwise, to prove that there is no such an assignment. Backtracking search is a complete method for solving CSPs. It performs a depth-first traversal of a search tree to solve CSPs. At each node of the search tree, an unassigned variable is selected to assign a value. The ordering in which the variables are assigned is crucial to the efficiency of backtracking search in solving CSPs. However, determining an optimal variable ordering that minimizes the size of the search tree is difficult (Liberatore 2000). Therefore, the ordering is usually determined by variable ordering heuristics (VOH) in practice.

In the past decades, much effort has been made in developing effective variable ordering heuristics (Boussemart et al. 2004; Refalo 2004; Michel and Van Hentenryck 2012; Wang, Xia, and Yap 2017; Habet and Terrioux 2019; Wattez et al. 2019; Li, Yin, and Li 2021; Audemard, Lecoutre, and Prud'homme 2023). Different VOHs have different performances when solving different problems. The performances of different VOHs can vary greatly while solving the same CSP instance. Thus, if we can find the best VOHs for different CSP instances, then the overall performance of black-box CSP solvers will be significantly improved. In recent years, adaptive variable ordering heuristics that identify efficient VOHs for different CSP instances have attracted much attention. For instance, a reinforcement learning technique, Multi-armed Bandit (MAB), has been used to design adaptive VOHs for CSPs. Xia and Yap proposed the first MAB-based adaptive VOH that applies the Upper Confidence Bound algorithm (UCB1) and Thompson Sampling (TS) algorithm to select a candidate VOH to make a decision at each search tree node (Xia and Yap 2018). The reward of a candidate H_i is updated with the numbers of children of the search tree nodes generated by H_i. Wattez et al. proposed to apply the exponentially weighted forecaster for exploration and exploitation (EXP3), UCB1 and TS to estimate the best VOH for given CSPs (Wattez et al. 2020). The method exploits a restart mechanism with the MAB-based framework, and the candidate VOHs are switched only after restarts. The Adaptive Single Tournament (AST) algorithm (Koriche et al. 2022) uses an ingenious strategy utilizing the different restart cutoffs in Luby sequence (Luby, Sinclair, and Zuckerman 1993) to cope with the exploitation and exploration of the bandits. It explores different candidates at restart runs with initial failure limits and exploits efficient ones at restart runs with increased failure limits. Another approach named SEVOH (Li et al. 2022) runs a probing procedure to sample search trees built by candidate VOHs and uses a scoring function considering search tree depth to select an efficient candidate for the input CSP. The probing procedure is simple yet efficient to solve the instances that are easy for the candidate VOHs, and the scoring function is effective in selecting candidates to solve some hard CSPs.

[*]Hongbo Li and Minghao Yin are corresponding authors.
Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In this paper, we propose a prediction-based approach for developing adaptive variable ordering heuristics to solve CSPs. The idea is to predict the total cost of completing a backtracking search from the beginning part of the search tree. The prediction allows us to select efficient VOHs for various CSP instances. Specifically, We propose to use *NPD* sequence, the sequence of numbers of positive decisions made before each failure occurs, to represent the topology of search trees built by backtracking search. We proved that there is a one-to-one correspondence between an *NPD* sequence and the topology of a binary search tree. A neural network based on Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997; Graves and Schmidhuber 2005) is trained offline. The network maps the topology of the beginning part of a search tree to the total number of failures to complete the search. Another machine learning technique, Gradient Boosting Decision Tree (GBDT) (Friedman 2001), is also employed to enhance the prediction results. The online solving procedure contains two phases. The first phase (probing phase) generates some search trees built by different candidate VOHs. The second phase predicts the total failure numbers of the search trees, and then selects an efficient one for the resolution. The first phase is efficient to solve the instances that are easy for some of the candidates. Thus, only the hard instances proceed to the second phase.

We performed extensive experimentation with MiniZinc benchmark suite (Stuckey et al. 2014) to examine the efficiency of the proposed approach. Four modern VOHs including Activity-based Search (ABS) (Michel and Van Hentenryck 2012), Conflict-history Search (CHS) (Habet and Terrioux 2019), refined weighted degree ($dom/wdeg^{ca.cd}$) (Wattez et al. 2019) and Failure-based Search (FRBA) (Li, Yin, and Li 2021) are used as candidates. We train a neural network and decision trees with the easy instances solved in the first phase, and then use the trained models to select efficient VOHs for hard instances. The results show that the proposed approach is effective in selecting efficient candidate VOHs for 307 out of 356 (about 86%) hard instances. It outperforms all the candidates, the state-of-the-art adaptive VOHs, AST (Koriche et al. 2022) and SEVOH (Li et al. 2022), and a time-split solver, in terms of the number of solved instances and the PAR2 score of runtime.

The main contributions of this paper are summarized as follows: (1) we propose to represent the topology of binary branching search trees by *NPD* sequence that is amenable to various machine learning methods, and demonstrate that efficient candidate VOHs can be identified by learning from search trees. (2) Our approach is more effective than the existing approach SEVOH in selecting efficient candidate VOHs for hard instances, and it outperforms the state-of-the-art adaptive VOHs in solving the MiniZinc instances.

Background

Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) $\mathcal{P}$ is a triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a set of *n* variables $\mathcal{X} = \{x_1, x_2 \ldots x_n\}$, $\mathcal{D}$ is a set of domains $\mathcal{D} = \{dom(x_1), dom(x_2) \ldots dom(x_n)\}$, where $dom(x_i)$ is a finite set of possible values for variable x_i, and $\mathcal{C}$ is a set of *e* constraints $\mathcal{C} = \{c_1, c_2 \ldots c_e\}$. Each constraint *c* consists of two parts, an ordered set of variables $scp(c) = \{x_{i1}, x_{i2} \ldots x_{ir}\}$ and a subset of the Cartesian product $dom(x_{i1}) \times dom(x_{i2}) \times \ldots \times dom(x_{ir})$ that specifies the allowed combinations of values for the variables $\{x_{i1}, x_{i2} \ldots x_{ir}\}$. A solution to a CSP is an assignment of a value to each variable such that all the constraints are satisfied. Solving a CSP $\mathcal{P}$ usually involves either finding a solution of $\mathcal{P}$ or proving that $\mathcal{P}$ does not have a solution. A CSP is satisfiable if it has at least one solution; otherwise unsatisfiable.

Variable Ordering Heuristics

A CSP is NP-hard in its general form. Backtracking search is widely used in solving CSPs. It performs a depth-first traversal of a search tree. At each search tree node, an unassigned variable is selected and a new node is generated after the assignment to this variable, then a propagation algorithm is applied to filter inconsistent values from the domains of variables. If the propagation leads to a failure, e.g. a domain wipeout, one or more assignments must be canceled and backtracking occurs. The ordering in which the variables are assigned is crucial to the efficiency of backtracking search. It is difficult to find an optimal ordering that minimizes the number of nodes explored in the search tree (Liberatore 2000). Therefore, the ordering is usually determined by variable ordering heuristics.

There are static and dynamic VOHs for solving CSPs. Static VOHs determine the variable orderings before search, while dynamic ones update the orderings dynamically during search. Dynamic ones are more popular in modern constraint solvers, such as Impact-based Search (Refalo 2004), RNDI (Grimes 2008), Activity-based Search (Michel and Van Hentenryck 2012), Correlation-based Search (Wang, Xia, and Yap 2017), Conflict-history Search (Habet and Terrioux 2019), weighted degree (Boussemart et al. 2004; Wattez et al. 2019), failure-based VOHs (Li, Yin, and Li 2021), VOH tracking variables in conflicts (Audemard, Lecoutre, and Prud'homme 2023), and so on. Most of these dynamic VOHs collect information during search to determine the ordering of variables, so they could be considered as adaptive VOHs in a sense. The performances of different VOHs may vary greatly in solving the same CSP instance. Thus, finding the right VOH for a CSP has been a key issue in constraint solving community. In this paper, we focus on how to adaptively select efficient VOHs for different CSPs.

Binary Branching Search Tree

There are two branching strategies in backtracking search, k-way branching and binary branching (Hwang and Mitchell 2005). Binary branching has been shown to be more powerful than k-way branching, so it is widely used in modern constraint solvers. Binary branching considers two types of decisions: positive decisions, e.g. the instantiation $x_i = v_i$ where x_i is an unassigned variable and v_i is a value in $dom(x_i)$ and negative decisions, e.g. the refutation $x_i \neq v_i$. The search trees built by binary branching are binary trees where a left branch corresponds to a positive decision and a right branch corresponds to a negative decision. For example, given a variable x_i with $dom(x_i)$={1,2,3,4,5}, an in-

Figure 1: A search tree built by binary branching.

stantiation $x_i = 1$ reduces the domain to $\{1\}$ and the refutation $x_i \neq 1$ reduces the domain to $\{2, 3, 4, 5\}$. Both types of decisions reduce the domain of the variable, so constraint propagation is applied after both types of decisions.

There are two cases in which a node is detected as a failure node: (1) The propagation of an instantiation (in a left branch) or a refutation (in a right branch) leads to a failure directly; (2) An internal node is identified as a failure node after exploration, e.g., both of its children are detected to be failures. When a binary search tree is being built, if a left branch (a positive decision) leads to a failure, the corresponding right branch (the negative decision) is generated immediately. If a right branch points to a failure node, its parent node is detected as a failure node by exploration. If no failure is detected, the search continues to make the next positive decision.

An example of a binary search tree is shown in Figure 1. In the example, the empty nodes with a cross are failure leaves in which the failures are directly detected by the propagation of decisions and the dark grey nodes with a cross are internal failure nodes detected by exploration. The solid nodes are those being explored. During the search, every internal failure node is a node being explored before its right child is detected as a failure node. If the instance is satisfiable, then the search will terminate at a leaf node containing the assignment to all variables; otherwise, the root node will be identified as a failure node eventually.

NPD Sequence

In the following, we focus on those failures detected by propagations directly, e.g., the failure leaves. The ultimate aim is to learn from search trees to solve a CSP. If the search tree contains termination nodes, e.g. the leaf node containing a solution or the root node is detected as a failure node, then the prediction is not necessary. Thus, we only consider the partial search trees containing no termination node, which are generated by binary branching within a limited number of failures.

We ignore the instantiations attached to the edges and keep the graph topology of the search tree, i.e. a binary tree. Note that we ignore the details of the decisions, e.g. the variables and the values, but the directions of the decisions, e.g. positive or negative, are still kept in the binary tree. In a binary tree, each left failure leaf corresponds to a failure led by a positive decision and each right failure leaf node corresponds to a failure led by a negative decision. We associate each failure leaf with an id i ($i \geq 1$) that indicates the ordering in which the failures occurred, e.g., a failure f_i occurs earlier than a failure f_j if $i < j$ and f_{i+1} is the one following f_i, as is shown in Figure 1.

Definition 1. *Given a binary search tree T and a failure leaf f_i of T.*

- *NPD(f_i) is the Number of Positive Decisions made before the first failure leaf occurs, if $i = 1$;*
- *NPD(f_i) is the Number of Positive Decisions made between f_i and f_{i-1}, if $i \geq 2$.*

For example, *NPD*(f_1), *NPD*(f_2) and *NPD*(f_4) in Figure 1 are 5, 0 and 1 respectively.

Definition 2. *The NPD sequence of a binary search tree T is the sequence of NPD(f_i)s collected from T, which are sorted in the ordering in which the failures occurred.*

It is easy to know that a binary search tree corresponds to a unique *NPD* sequence. On the other hand, we have the following theorem.

Algorithm 1: RECONSTRUCTION

Input: An array *seq* storing an *NPD* sequence;
Output: The topology of a binary search tree;

```
1  initialize a root node ROOT;
2  currentNode ← ROOT;
3  for i = 0 to array.length - 1 do
4      p ← seq[i];
5      if p ≥ 1 then
6          for j = 1 to p do
7              generate the left branch of currentNode;
8              if j < p then
9                  currentNode ←
                     currentNode.leftChild;
10         mark currentNode.leftChild as a failure leaf;
11         generate the right branch of currentNode;
12         currentNode ← currentNode.rightChild;
13     else
14         mark currentNode as a failure leaf;
15         while currentNode is the rightChild of its
             parent do
16             currentNode ← currentNode.parent;
17             mark currentNode as an internal failure
                 node;
18         generate the right brother of currentNode;
19         currentNode ← currentNode.rightBrother;
20 return the binary tree rooted at ROOT;
```

Theorem 1. *An NPD sequence determines a unique topology of a binary search tree.*

Proof. We describe the procedure of reconstructing a binary tree from an *NPD* sequence here. The pseudocode of the procedure is in Algorithm 1. At the beginning, we set current node to the root node, and then process the numbers of the *NPD* sequence one by one. The numbers are processed according to the rules of binary branching building a search tree. There are two cases of the numbers:

(1) $NPD(f_i) \geq 1$. It indicates there are $NPD(f_i)$ successive left branches and $NPD(f_i)$ - 1 internal nodes connecting current node and failure leaf f_i. This is because the search always makes left decisions before next failure occurs. For example, as is shown in the dash circles surrounded part in Figure 1. The current node is the solid node pointed by $x_2 \neq 2$ and $NPD(f_7)$ is 3. There are 3 left branches and two internal nodes connecting the current node and failure leaf f_7. In addition, the failure node f_i's right brother will be generated immediately. Thus, in this case, $NPD(f_i)$ successive left branches and the corresponding nodes are generated (lines 6-9 of Algorithm 1) and the last node is marked as f_i (line 10). Then the right brother of f_i is generated and set as current node (lines 11-12).

(2) $NPD(f_i) = 0$. It indicates that the current failure is led by the propagation of a negative decision, and then the parent node of f_i is detected as a internal failure node. Thus, in this case, we find the ancestor node of f_i from bottom to up, which is the first one pointed by a left branch (lines 15-17). Then we generate the ancestor's right brother and set current node to the right brother (lines 18-19).

Following the above procedure, we will get a unique topology of a binary tree. □

Besides $NPD(f_i)$, we associate each failure f_i with more features to enhance the representation for learning.

- $APD(f_i)$: This is the number of All Positive Decisions in the path from the root to f_i.
- $AND(f_i)$: This is the number of All Negative Decisions in the path from the root to f_i.
- $Fix(f_i)$: This is the number of variables that are fixed before the propagation of the decision leading to f_i. Note that some of the variables are fixed by the propagations of previous decisions, so $Fix(f_i)$ may be larger than $APD(f_i)$.
- $DIR(f_i)$: This is the label of the direction *DIR* of the branch leading to failure f_i, 1 and -1 for left and right branches respectively.

Each failure f_i contains 5 features for the prediction task. If we run a backtracking search within k failures, then we can obtain 5 sequences, each with length k, to represent the partial search tree. Besides the original sequences, we add the maximum values and the upper bound of *NPD*, *APD*, *AND*, and *Fix* sequences at their ends respectively. The number of variables is the upper bound of *NPD*, *APD*, *AND*, and *Fix*. The *DIR* sequence contains values from {-1, 1}, so we use the sum of the sequence instead of the maximum value and use the number of failures instead of the upper bound.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	*max*	*ub*
NPD	5	0	0	1	0	1	3	2	5	n
APD	5	4	3	3	2	2	4	5	5	n
AND	0	1	1	1	2	1	2	3	3	n
Fix	5	4	3	3	2	2	4	5	5	n
DIR	1	-1	-1	1	-1	1	1	1	2	8

We assume that no variable is fixed by propagation, so the *Fix* and *APD* sequences are same. n is the variable number.

Table 1: Representation of the binary search tree in Figure 1

Finally, a binary search tree is represented by a sequence set containing the 5 sequences. Table 1 presents the representation of the binary search tree in Figure 1.

It takes constant time to collect these numbers from each failure leaf, so building a binary search tree costs the major time of obtaining these sequences. Modern constraint solvers provide various constraints with different propagation algorithms. These algorithms vary in their time costs, making it difficult to determine the time complexity of propagating a decision across different CSPs. To discuss the time cost of generating a binary search tree, we use O(P)[1] to represent the worst-case time cost of propagating a decision.

Theorem 2. *Given a CSP with n variables, backtracking search with binary branching costs O($(k+n)P$) time to build a partial search tree within k failures.*

Proof. A partial search tree contains at most $n-1$ succeed positive decisions. There are at most k positive decisions leading to failures, which are followed by the corresponding negative decisions. The first failure is caused by a positive decision, so there are at most $k-1$ negative decisions. Each decision is followed by a propagation with time cost O(P). Thus, it costs O($(k+n)P$) time to build a search tree within k failures. □

Model Architectures

Based on the proposed representation of binary branching search trees, we employ two supervised learning techniques for the prediction task.

Neural Network Based on LSTM

The representation of a binary search tree contains 5 sequences of integers, which are time series data. Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber 1997; Graves and Schmidhuber 2005) is well-suited to classify and make predictions based on time series data. Therefore, we design a network based on LSTM for the classification task. A set of sequences, derived from the beginning part of a search tree, is labeled with the total failure number of the search tree. We normalize these labels to fall between 0 and

[1]Most of the propagation algorithms used in modern constraint solvers are linear and polynomial, but there are also a few exponential ones, such as the generalized arc consistency algorithm (Lecoutre 2009) for general constraints, the simple tabular algorithms (Lecoutre 2011; Li et al. 2013) for table constraints. Therefore, O(P) might be linear, polynomial, or even exponential.

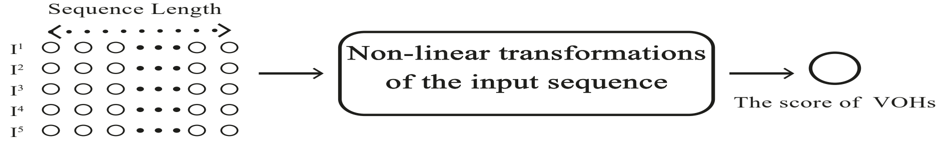

Figure 2: The prediction process.

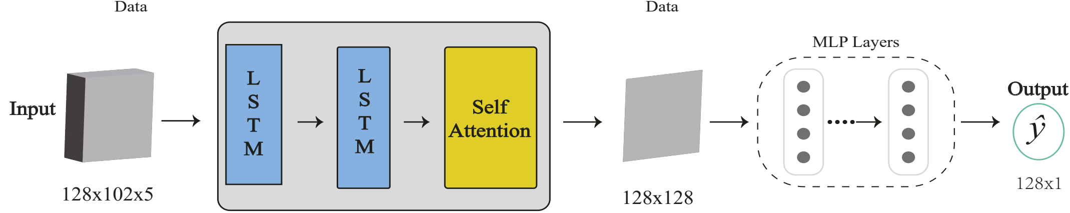

Figure 3: Architecture of the neural network.

1 (the details will be shown in the experiment section), so our model owns the ability to map the representation of a binary search tree to a scalar output $y \in (0, 1)$. The prediction process is shown in Figure 2.

Given a data point (a sequence set) $I = \{I^1, I^2, I^3, I^4, I^5\}$ consisting of 5 different time series with $I^i \in \mathbb{R}^d$. I is standardized by Standard Scaler (Buitinck et al. 2013) first. Figure 3 depicts the architecture of our model, which takes the standardized data point as input and employs two `LSTM` layers to capture sequence information. The classifier depends on the final output state of the `LSTM`. Additionally, a Self-Attention (Bahdanau, Cho, and Bengio 2014) layer follows the `LSTM`, enhancing the model's ability to capture distant dependencies and global patterns within sequences.

$$v_i = Attention(LSTM(I)) \tag{1}$$

To prevent overfitting, we apply the dropout operator (Hinton et al. 2012) with a rate of 0.1 after a fully connected layer that accepts the outputs of the Attention mechanism. The output from this layer is then forwarded to a dense layer for further processing, which is formally described as the following.

$$s_i = \mathbf{D}(ReLU(Linear_p(v_i))) \tag{2}$$

where $Linear_p : R^h \to R^h$ is a fully connected neural network with dimension of h, $\mathbf{D}(.)$ is a dropout operator. The Rectified Linear Units ($ReLU$) activation function (Fukushima 1975) is used as the non-linear activation function. A linear transformation is then applied to the output to obtain the classification result.

$$\hat{y} = Linear_m(s_i) \tag{3}$$

where $Linear_m : R^h \to R^1$.

We train the model by minimizing the binary cross-entropy loss function (Cox 1958). This loss function is commonly used in binary classification tasks and measures the difference between the predicted probability distribution and the true probability distribution of binary outcomes. The calculation formula for binary cross-entropy loss is as follows:

$$\mathcal{L}(\theta) = -\frac{1}{B}\sum_{i=1}^{B}(p \times y_i \log(g(\hat{y})) + (1 - y_i)\log(1 - g(\hat{y}))) \tag{4}$$

where B denotes the batch size, y_i is the label, $g(\hat{y})$ is the outputs of the model, and p is set to 0.1. We chose the value because our dataset is imbalanced, and our objective is to allocate more attention to the negative class for our task.

Gradient Boosting Decision Tree

The second model we adopt is the Gradient Boosting Decision Tree (`GBDT`) (Friedman 2001), a powerful ensemble learning technique that is widely used due to its efficiency, accuracy, and interpretability. One of the distinctive features of `GBDT` is its ability to handle a variety of data types, including categorical and numerical features, while also demonstrating resilience to overfitting. This makes it well-suited for addressing our problem.

We train a model for each of the 4 sequences, *NPD*, *APD*, *AND*, and *Fix*. The features are processed individually because `GBDT` can not process the combination of the 4 sequences. The loss function used in our work is binary cross-entropy loss (Cox 1958). In the prediction phase, the outputs of the four models are simply combined by addition.

Prediction-based Adaptive VOH

Based on the *NPD* representation of binary search trees and the model architectures, we propose a novel approach for developing adaptive VOHs, namely Prediction-based Adaptive Variable Ordering Heuristics (`PBAVOH`). The approach begins with an offline training task, e.g. a `LSTM` network and `GBDT` models are trained to map the *NPD* sequence of the beginning part of a search tree to its total failure number. Then the online solving task runs a probing procedure to sample search trees built by different candidate VOHs,

Algorithm 2: PBAVOH

Input: a CSP P; k candidate VOHs: H_1, H_2, ..., H_k; the maximum number of probing rounds: $roundLimit$;
Output: unsatisfiable, or a solution.

```
1  for i = 1 to k do
2      score[i] ← 0 ;
3  round ← 1;
4  while round ≤ roundLimit do
5      for i = 1 to k do
6          use H_i to guide search until either P is
             solved or restart limit is reached;
7          if a solution has been found then
8              return the solution;
9          if unsatisfiable has been proved then
10             return unsatisfiable;
11         record the feature sequences of the search
             tree in matrix ;
12         score[i] ← score[i] + predict(matrix) ;
13         restart the search;
14     round ← round + 1;
15 select the H_best with smallest score[i] to solve P;
```

and uses these search trees to select an efficient candidate. The heuristic informations of all candidate VOHs are accumulated throughout the resolution, so the heuristic scores evolve after each round and the probing restart runs generate different search trees. The main framework of the online solving task is shown in Algorithm 2. An array *score* is used to record the scores of the candidates. The loop from lines 4 to 14 is the probing phase. If the CSP is solved in a restart run of this phase at line 6, the result will be returned at line 8 or line 10; otherwise, the feature sequences of the first $failLimit$ failures are recorded in $matrix$ at line 11 and the neural network trained offline takes the $matrix$ as input to predict a score for candidate H_i at line 12. The $roundLimit$ scores of each candidate are accumulated and finally, the candidate with the smallest score is selected to guide a backtracking search until the search completes.

We have employed two machine learning techniques in the prediction task, so we have three different versions of PBAVOH. The PBAVOH_{LSTM} uses LSTM in the learning and prediction task. The PBAVOH_{GBDT} uses GBDT. The PBAVOH_{Blend} combines the output of two ML models by simply adding them together at line 12.

Experiments

To examine the performance of the proposed approach, we perform extensive experimentation with the MiniZinc benchmark suite. We include 46 MiniZinc models with 1876 instances in the experiments. The baselines are the state-of-the-art adaptive VOHs, AST (Koriche et al. 2022) and SEVOH (Li et al. 2022). Four efficient VOHs including ABS (Michel and Van Hentenryck 2012), $dom/wdeg^{ca.cd}$ (Wattez et al. 2019), CHS (Habet and Terrioux 2019) and FRBA (Li, Yin, and Li 2021) are used as candidates. All search strategies use lexicographic ordering as a value selector. The performance of searching for the first solution or proving unsatisfiable is measured by the number of instances solved in a timeout limit of 1200 seconds and the PAR2 score[2] of runtime. The experiments[3] were run in Choco solver (Prud'homme, Fages, and Lorca 2017). The environment is JDK8 under CentOS 6.4 with 4 Intel Xeon CPU E7-4820@2.00GHz processors (40 CPU cores in total). Each run is allocated with 1 CPU core and 1 GB RAM. The MiniZinc instances are flattened offline. After eliminating some large instances that cannot be flattened in 1 hour and the problems where unsatisfiable are proved at the root node, we include 46 MiniZinc models with 1876 instances in the experiments. Most of the problems contain non-binary constraints including many global constraints, such as AllDifferent, GlobalCardinality, Cumulative, Member, etc. The best one is in bold in the following tables.

There are 937 easy instances solved during the probing phase and are used for the training task. They are randomly divided into the ones for training and validation with a ratio of 9:1. For each of the easy instances, we sample search trees by running the four candidate VOHs equipped with lexicographic value selector, and three restart strategies, Luby restart, geometrical restart and no restart. Each of the search strategies was run with 5 random seeds from 1 to 5. The feature sequences of the first 100 failures of the last search tree that solves the instance are collected as a training data point. After eliminating those runs that can not solve the instance and those runs solving the instance within 100 failures, we have 27923 data points for the training task. The original label of each data point is the total failure number. We normalize original labels between 0 and 1. The labels of data points from each instance are grouped and normalized as follows,

$$N(y) = \begin{cases} S(y), & y < P_{10} \vee Zscore(y) < 1 \\ 1, & y > P_{10} \wedge Zscore(y) \geq 1 \end{cases} \quad (5)$$

where the P_{10} represents $10th$ percentile, and $S(y)$ is a scaling function. We use Y to denote the set of all labels of the data points from an instance and y is one of the labels. The function N is used to decide whether y should be set to 1 or mapped to [min, max] by the S function. In the experiments, we set max and min to 0.99 and 0 respectively.

$$y_{std} = \frac{(y - Y.min)}{(Y.max - Y.min)} \quad (6)$$

$$S(y) = y_{std} \times (max - min) + min \quad (7)$$

where the $Y.min$ and $Y.max$ are the maximum and minimum ones in Y.

Firstly, we report the time costs of the training and prediction. It costs less than 30 minutes to train the LSTM model

[2]The PAR2 score of a solver is defined as the sum of all runtimes for solved instances + 2×timeout for unsolved instances, which is used in used SAT competition.

[3]The source codes and the trained models are available at https://github.com/lihb905/pbavoh.

(the one used in Table 2, Table 3 and Table 5) with a single NVIDIA 3060 GPU with 12GB RAM. It costs less than 5 minutes to train the decision trees with one core of an i5-10400F @2.90GHz CPU. It takes about 2 seconds to do the predictions in each instance. The probing phase is an indispensable procedure because it is effective in solving a number of easy instances. For the hard instances, the sequences are extracted incidentally during the probing phase, so the time cost of the extraction can be ignored.

Secondly, we compare PBAVOH with SEVOH to examine their performance of identifying efficient candidate VOHs for different CSPs. The two approaches run the same probing procedure with $roundLimit$ = 100, so they are compared in the hard instances not solved in the probing phase. For each of the hard instances, we run the probing procedure and then specify each of the candidates in the second phase to determine the best candidate for the instance according to the total time cost. After eliminating the instances where all the candidates fail to solve them in the second phase, Table 2 contains the results of 356 instances. The result of random selection is presented as a baseline. The %Best (#Best) row is the percentage (number) of instances where the selected VOH is the best one and the %Solved (#Solved) row is the percentage (number) of instances solved by the selected VOH. We present the Solved rows because the performance of the best candidate and the second best one (or even the third one) may be close in some instances, so finding the best one is difficult in these cases and finding an efficient one that solves the instances is more practical. We can see that $\texttt{PBAVOH}_{GBDT}$ performs better than the random selection and worse than the others. It is interesting that combining the two ML models results in the best one, $\texttt{PBAVOH}_{Blend}$, which solves the largest number of instances. We checked the detailed results and found that $\texttt{PBAVOH}_{GBDT}$ is helpful in some instances where $\texttt{PBAVOH}_{LSTM}$ makes bad selections. This is reasonable because it performs better than the random selection, so it does have a certain effect and could make good selections in some instances. SEVOH finds the best candidates for more instances than the others. Although $\texttt{PBAVOH}_{Blend}$ has a fair performance in finding the best one, it works well in finding a candidate that solves the input instance. $\texttt{PBAVOH}_{Blend}$ selects efficient candidates for 307 out of 356 hard instances and solves more instances than the others. In the following, $\texttt{PBAVOH}_{Blend}$ is used as the representation of PBAVOH.

Thirdly, we compare PBAVOH with SEVOH and AST in Table 3 to examine their performance in solving various CSPs. The results of each candidate, a time split solver (TSS, that uniformly assigns 300 seconds to each candidate), the virtual best candidate (VBC, that selects the best candidate in hindsight) and the virtual best of all (VBA, that selects the best one of all the 9 strategies in hindsight) are also presented for reference. In the second phase of SEVOH and PBAVOH, a geometric restart with initial cutoff of 10 failures and a growing factor 1.1 is employed, and the AST employs a Luby restart with initial cutoff of 150 failures (the recommended setting in (Koriche et al. 2022)). The time cost of PBAVOH includes the probing phase, extracting features, predictions, and the searching phase. The AST and the SEVOH use the recommended settings in the papers and PBAVOH uses the same setting as SEVOH. Each candidate VOH is equipped with the probing procedure to get a better performance for a fair comparison. The instances where all approaches result in timeout are eliminated. The All rows contain all involved instances, and the Hard contains 356 instances which are not solved in the probing phase and 25 more instances solved by AST and TSS. It is interesting that ABS equipped with the probing procedure performs better than the existing adaptive VOHs. PBAVOH makes more accurate selections than SEVOH, and it outperforms ABS and all the others in both the number of solved instances and the PAR2 score. The results demonstrate that learning from the topology of search trees effectively identifies efficient VOHs for different CSPs.

	Random	SEVOH	LSTM	GBDT	Blend
%Best	25%	**60%**	48%	28%	49%
#Best	89	**212**	170	101	174
%Solved	50%	80%	84%	69%	**86%**
#Solved	177	285	298	247	**307**

AST does not make a final selection, so it is not in this table.

Table 2: Results of selecting candidates in hard instances

One might be curious about how PBAVOH performs on unseen problems and how it performs when being trained with only the instances from the problem to be solved. We examine that with 6 problems[4]. The result (#Solved) is presented in Table 4. We can see that $\texttt{PBAVOH}_{unseen}$ significantly outperforms the random selection strategy. The result indicates that our approach captures some common structure property hidden in the search trees of different problems, so it can be used to select efficient VOH for unseen problems. Compared with SEVOH, $\texttt{PBAVOH}_{unseen}$ wins in 4 problems and loses in 1 problem. $\texttt{PBAVOH}_{only}$ also outperforms the random selection. If the two training sets are combined, better performance is obtained by $\texttt{PBAVOH}_{all}$. There is an exception problem *nsp* where $\texttt{PBAVOH}_{all}$ is outperformed by $\texttt{PBAVOH}_{only}$ and $\texttt{PBAVOH}_{unseen}$. From the random selection and #hard columns, we can see that each *nsp* instance has only one candidate solving it on average. It is reasonable that $\texttt{PBAVOH}_{all}$ misses the best one in some instances. Note that $\texttt{PBAVOH}_{all}$ still performs much better than the random selection in *nsp*. The results demonstrate that PBAVOH is robust to select efficient VOHs for different CSPs.

Table 5 presents the result (#Solved) of the hard instances of each problem. We can see PBAVOH solves the largest number of instances in 13 out of 20 MiniZinc models. The results show that PBAVOH is more robust than the existing adaptive VOHs in general.

We further examine whether the satisfiability of the instances affects the performance. The results are present in Table 6. We can see that AST and SEVOH get the best performance in SAT and UNSAT instances respectively. It is interesting that PBAVOH gets the second-best performance

[4]The 6 problems are selected because they have more hard instances than the other problems.

		ABS	CHS	$\frac{dom}{wdeg}$	FRBA	VBC	TSS	AST	SEVOH	PBAVOH	VBA
All	#Solved	1224	1006	1041	1180	1291	1205	1205	1222	**1244**	1328
	PAR2(1328)	258.5	618.3	561.3	339.9	147.2	328.5	283.4	267.0	**238.1**	65.2
Hard	#Solved	287	69	104	243	356	283	273	285	**307**	381
	PAR2 (381)	774.8	2012.3	1814.8	1057.7	392.0	838.1	854.8	787.6	**686.8**	182.7

Table 3: The performance of number of solved instances and PAR2 score of runtimes in seconds

Problem	#easy	#hard	AST	SEVOH	PBAVOH_{all}	PBAVOH_{unseen}	PBAVOH_{only}	Random
cars	16	26	18	18	**24**	23	22	7.6
marketsplit	10	33	20	30	**32**	**32**	**32**	29.9
nsp	78	52	**49**	38	39	41	43	13.8
QCP	20	17	10	12	**15**	14	13	13
sb	9	18	10	13	14	13	**15**	11.4
wwtpp	402	161	119	137	**144**	115	133	64.5
Total	535	307	226	248	**268**	238	258	140.2

#easy is the number of instances used for generating training data; #hard is the number of instances used for testing; PBAVOH_{all} uses all the 535 instances for the training set; PBAVOH_{unseen} removes the instances of each of the problems from the training set; PBAVOH_{only} uses only the easy instances of each of the problems for training; Random is the number of instances solved by making random selections at the second phase (the average of 10 tests), which reflects the average number of candidate solving each instance.

Table 4: The results of PBAVOH trained with different problems

Problem	#hard	AST	SEVOH	PBAVOH
amaze2	9	**7**	6	6
amaze3	2	**2**	1	**2**
bibd	3	2	2	**3**
carseq	5	**5**	1	0
cars	26	18	18	**24**
cube	1	0	**1**	**1**
CostasArray	3	2	**3**	**3**
fillomino	5	**4**	**4**	2
golfers2	2	**2**	0	0
market-split	33	20	30	**32**
mknapsack	6	4	4	**6**
nmseq	1	**1**	0	0
non	5	4	**5**	**5**
nsp-1	52	**46**	35	35
nsp-2	7	3	3	**4**
oocsp	1	**1**	0	**1**
QCP	17	10	12	**15**
sb	15	10	13	**14**
wwtpp-random	61	56	49	**58**
wwtpp-real	97	63	**88**	86

#hard is the total number of instances used for testing. We didn't present the results of six problems where all methods solve the same number of instances.

Table 5: The detailed results of different problems

	AST	SEVOH	PBAVOH
SAT (227)	**209**	159	191
UNSAT (134)	61	**126**	116
SAT+UNSAT (361)	270	285	**307**

Table 6: Results on instances classified by satisfiability

in both SAT and UNSAT instances, which are not far from that of the best ones, so it gets the best performance overall. The satisfiability is unknown in most practical scenarios, so PBAVOH is a good choice.

In summary, the machine learning models capture some common structures hidden in the search trees of different problems. Learning from the topology of search trees is effective in selecting efficient VOHs for different CSPs, which results in an efficient adaptive VOH that outperforms the state-of-the-art adaptive VOHs, AST and SEVOH.

Conclusion

In this paper, we propose a novel approach, namely PBAVOH, to adaptively select efficient variable ordering heuristics for different CSPs. Our study finds that the efficiency of a variable ordering heuristic solving a CSP instance can be estimated by learning from the topology of search trees. We show that the *NPD* representation of binary branching search trees is amenable to various machine learning techniques, and then we can predict the total failure number of a search tree from its beginning part to implement the adaptive variable ordering heuristic. The experimental results show that the LSTM model and GBDT models trained with the data sampled from easy instances are effective in selecting efficient candidate VOHs for hard instances. PBAVOH outperforms the state-of-the-art adaptive VOHs, AST and SEVOH in terms of the number of solved instances and the PAR2 score of runtime.

Acknowledgments

We thank the anonymous referees for their constructive comments. This work is supported by the National Natural Science Foundation of China under Grant 62276060.

References

Audemard, G.; Lecoutre, C.; and Prud'homme, C. 2023. Guiding Backtrack Search by Tracking Variables During Constraint Propagation. In *Proc. CP'23*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 9:1–9:17. ISBN 978-3-95977-300-3.

Bahdanau, D.; Cho, K.; and Bengio, Y. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proc. ECAI'04*, 146–150.

Buitinck, L.; Louppe, G.; Blondel, M.; Pedregosa, F.; Mueller, A.; Grisel, O.; Niculae, V.; Prettenhofer, P.; Gramfort, A.; Grobler, J.; et al. 2013. API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*.

Cox, D. R. 1958. The regression analysis of binary sequences. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 20(2): 215–232.

Friedman, J. H. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 1189–1232.

Fukushima, K. 1975. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3): 121–136.

Graves, A.; and Schmidhuber, J. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural networks*, 18(5-6): 602–610.

Grimes, D. 2008. A study of adaptive restarting strategies for solving constraint satisfaction problems. In *Proc. 19th Irish Conference on Artificial Intelligence and Cognitive Science-AICS*, volume 8, 33–42. Citeseer.

Habet, D.; and Terrioux, C. 2019. Conflict history based search for constraint satisfaction problem. In *Proc. of the 34th ACM/SIGAPP Symposium on Applied Computing*, 1117–1122. ACM.

Hinton, G. E.; Srivastava, N.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. R. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 9(8): 1735–1780.

Hwang, J.; and Mitchell, D. G. 2005. 2-way vs d-way branching for CSP. In *Proc. CP'05*, 343–357. Springer.

Koriche, F.; Lecoutre, C.; Paparrizou, A.; and Wattez, H. 2022. Best Heuristic Identification for Constraint Satisfaction. In *Proc. IJCAI'22*, 1859–1865.

Lecoutre, C. 2009. *Constraint Networks: Techniques and Algorithms*, chapter 4, 185–237. John Wiley Sons, Ltd.

Lecoutre, C. 2011. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16: 341–371.

Li, H.; Liang, Y.; Guo, J.; and Li, Z. 2013. Making simple tabular reductionworks on negative table constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, 1629–1630.

Li, H.; Wu, Y.; Yin, M.; and Li, Z. 2022. A Portfolio-Based Approach to Select Efficient Variable Ordering Heuristics for Constraint Satisfaction Problems. In *Proc. CP'22*, 32:1–32:10.

Li, H.; Yin, M.; and Li, Z. 2021. Failure Based Variable Ordering Heuristics for Solving CSPs. In *Proc. CP'21*, 9:1–9:10.

Liberatore, P. 2000. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116(1): 315–326.

Luby, M.; Sinclair, A.; and Zuckerman, D. 1993. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4): 173–180.

Michel, L.; and Van Hentenryck, P. 2012. Activity-based Search for Black-box Constraint Programming Solvers. In *Proc. CPAIOR'12*, 228–243. Springer.

Prud'homme, C.; Fages, J.-G.; and Lorca, X. 2017. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S.

Refalo, P. 2004. Impact-based Search Strategies for Constraint Programming. In *Proc. CP'04*, 557–571. Springer.

Stuckey, P. J.; Feydy, T.; Schutt, A.; Tack, G.; and Fischer, J. 2014. The MiniZinc Challenge 2008–2013. *AI Magazine*, 35(2): 55–60.

Wang, R.; Xia, W.; and Yap, R. H. C. 2017. Correlation Heuristics for Constraint Programming. In *Proc. ICTAI'17*, 1037–1041. IEEE.

Wattez, H.; Koriche, F.; Lecoutre, C.; Paparrizou, A.; and Tabary, S. 2020. Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. In *Proc. ECAI'20*, 371–378. IOS Press.

Wattez, H.; Lecoutre, C.; Paparrizou, A.; and Tabary, S. 2019. Refining constraint weighting. In *Proc. of ICTAI'19*, 71–77. IEEE.

Xia, W.; and Yap, R. H. C. 2018. Learning robust search strategies using a bandit-based approach. In *Proc. AAAI'18*, 6657–6665. AAAI.