Generalizing Learning-to-Prune for Constraint Programming

Deepak Ajwani
School of Computer Science, University College Dublin, Ireland

Peter Nightingale
Department of Computer Science, University of York, United Kingdom

Felix Ulrich-Oltean
Department of Computer Science, University of York, United Kingdom

Abstract

Learning-to-Prune (LTP) is a machine learning technique that has been successfully applied to several combinatorial optimization problems. In LTP, a machine learning model is trained to predict parts of a problem instance that can be removed without large changes to the value of the optimal solution. For example, on the maximum clique problem, the model would typically be trained to predict vertices that are unlikely to be part of a maximum clique. After LTP is applied to prune a given problem instance, the remaining part of it can be solved with a conventional solver. LTP is parsimonious with training data, usually requiring only a few training instances. It also produces ML models that are relatively simple and interpretable. In this paper we present early work on LTP for CP: a general framework for training and applying LTP to problems expressed in a CP modelling language. In this context LTP cannot benefit from problem-class-specific features. We have developed a graph-based representation of the constraint network that captures the distinct constraint types and the location of variables within each constraint. We also capture features of the variables such as whether they are in the objective function. We focus on pruning Boolean variables, and on problems with a linear objective function. Our method is evaluated on three problem classes: maximum clique, the combinatorial auction problem, and a simple nurse rostering problem with preferences. In each case, we show that LTP can substantially improve the performance of a systematic solver, with a small cost in terms of the optimality gap.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint programming, machine learning, combinatorial optimization

Funding *Deepak Ajwani*: Partially supported by Science Foundation Ireland under Grant number 18/CRT/6183
Peter Nightingale: Partially supported by UK EPSRC grant EP/W001977/1
Felix Ulrich-Oltean: Supported by UK EPSRC grant EP/W001977/1

1 Introduction

Learning-to-Prune (LTP) is a machine learning technique that has been applied to several combinatorial optimization problems with very promising results. LTP does not entirely solve a given problem instance, but aims to make it easier to solve by removing some parts of it (e.g. assigning some variables). To date, each implementation of LTP is specific to a problem class, typically with problem-class-specific features and machine learning setup. In this paper we present early work on LTP for CP: a general framework for training and applying LTP to problems expressed in a CP modelling language. The generalized LTP framework still needs to be trained for each problem class. However, the set of features is (largely) generated from a graph that is derived from the syntax tree of the CP model. We have developed a graph-based representation of the constraint network that captures the distinct constraint types and the locations of variables within each constraint. We also capture other

features of each variable such as whether it is in the (linear) objective function, and its coefficient in the objective function. While the generalized LTP framework cannot benefit from problem-class-specific features, we have found that it can be effective: experimental results on two problem classes are very promising.

In LTP, a machine learning model is trained to predict parts of a problem instance that can be removed without large changes to the value of the optimal solution. For example, on the maximum clique problem, the model would typically be trained to predict vertices that are unlikely to be part of a maximum clique. After LTP is applied to prune a given problem instance, the remaining part of it can be solved with a conventional solver. If LTP is working well, the conventional solver will be able to solve the pruned problem instance to optimality much faster than the original instance, and with a small optimality gap between the two. LTP is parsimonious with training data, usually requiring only a few training instances. It also produces ML models that are relatively simple and interpretable compared to other ML methods (as described in Section 2).

In this paper we focus on pruning Boolean variables, and on problem classes with a linear objective function stated directly on the same set of Boolean variables. LTP is a supervised learning method. We train a classifier to predict whether a variable will be *true* or *false* in an optimal solution. Training data is generated by solving the training instances to optimality, and labelling each relevant variable with its assignment in the optimal solution. It is important to note that LTP learns to prune variables based on the instance distribution; it may not be accurate if it is applied to instances from a different distribution.

The proposed framework is evaluated on three problem classes: maximum clique, the combinatorial auction problem, and a simple nurse rostering problem with preferences. In each case, we show that LTP can substantially improve the performance of a systematic solver, with a small cost in terms of the optimality gap.

Conceptually, LTP is related to *streamliners* [7]: constraints that are added to a CSP instance in the hope of solving it faster, but with no guarantee that the instance will remain satisfiable. LTP can be thought of as predicting streamliners based on the instance distribution, where the streamliner constraints are all unary and the context is optimization rather than constraint satisfaction.

2 Background and Related Work

In this section, we present an outline of the work in the general area of machine learning for combinatorial optimization followed by recent work on the learning-to-prune framework. We also describe the graph embedding techniques that we use in our approach and the mid-level modelling language used by us in our work.

2.1 Machine Learning for Combinatorial Optimization

In recent years, researchers have been exploring the usage of machine learning (ML) techniques to efficiently solve combinatorial optimization problems. This is motivated by the following factors:

- In many applications like nurse rostering and vehicle routing problems, instances from the same distribution need to be solved repeatedly (e.g. every week) and ML techniques can learn the patterns from earlier solutions to speed-up solving the newer instances.
- For many optimization problems, solving larger instances of the problem continues to remain a major challenge. A learning model trained on solutions of smaller instances (that

modern solvers can solve in reasonable time) can generalize and be used as a heuristic solver for larger instances from the same distribution.

- In applications where even solving small instances is difficult, reinforcement learning techniques that capture the objective function in terms of rewards can still be quite effective.

There are a large number of techniques that have been developed in this broad area in the last decade. The learning techniques can be used as a standalone heuristic or as a component of the solver or as a helper function to speed up solving the optimization problems. The learning techniques (used in any of the above ways) can be broadly divided into three categories: (i) end-to-end supervised learning techniques (see [3] for a survey), (ii) reinforcement learning techniques (see [15, 30] for a survey) and (iii) unsupervised learning techniques (see [9, 28] for some recent work). Many of these techniques use architectures such as graph neural networks [29] and representations such as node and graph embeddings for solving the problem.

Nonetheless, the application of machine learning to combinatorial optimization is not without its challenges. Unlike many applications of machine learning in computer vision, natural language processing, machine translation, self-driving vehicles etc., combinatorial optimization problems have highly correlated decision variables and the correlations are long-range with very little spatial or temporal coherence. Thus, deep learning architectures such as convolutional neural networks and recurrent neural networks that are designed to leverage the spatial and temporal coherence often do not generalize well for combinatorial optimization problems. In fact, since many of the combinatorial optimization problems are NP-hard, the exact decision boundary separating the optimal solutions from non-optimal solutions is often quite complex. Learning the exact boundary typically requires a fairly complex architecture and a vast amount of training data, which is difficult to obtain for NP-hard optimization problems. For instance, one of the successful learning technique in this domain is the neural combinatorial optimization approach [2]. This is a combination of Pointer networks [27], a Monte Carlo policy gradient and an actor-critic architecture. The pointer network is itself a combination of a sequence-to-sequence deep learning model based on RNN, another deep learning model based on LSTM for attention mechanism to deal with long-range correlations and usage of pointers to deal with the issue of fixed vocabulary size required for neural networks. Since the learnt algorithm (mapping from input to output) is implicit in the learning model, whose complexity in turn is governed by the highly complex underlying architecture, the learnt heuristics are far too complex for humans to interpret (leave aside the mathematical analysis in pursuit of provable guarantees). It is unclear what features of the input instances are being exploited by the learnt heuristic and on which class of data-sets will the learnt heuristic perform well. Furthermore, the non-interpretable nature of the heuristic means that addition of a new constraint or slightest deviation of the input instance from the training distribution can result in unexpected solutions. These limitations severely constrain the usage of such heuristics in real-life deployment of optimization solutions. Note that interpretable learning techniques are also important in many applications from a legal and privacy regulations (e.g., GDPR) view as well as for engineers to trust and deploy them in many real-world applications.

Reinforcement learning techniques for combinatorial optimization usually have a high computational time involved and unsupervised approaches generally struggle to obtain good optimality-time tradeoff for the combinatorial optimization problems.

2.2 Learning-to-Prune

To address the limitations of supervised end-to-end deep learning approaches, an alternative learning framework called learning-to-prune (LTP) has been developed. The high-level idea behind this framework is that instead of learning a complex decision boundary to exactly separate all optimal solutions from non-optimal solutions (that necessitates complex architecture and large amount of training), it predicts the variable values in the optimal solution and learns a simpler decision boundary such that all variables on one side are either all 0s or all 1s in some optimal solution. These variables are then fixed and the remaining problem (hopefully much smaller) is then solved by the exact solver. In fact, this framework is actually solving a sparsification problem but instead of developing a problem-specific sparsification algorithm or heuristic, a supervised classification model is employed. The framework has been successfully used for the maximum clique problem [12, 13], travelling salesperson problem [6, 22], k-median and related problems of facility location, set-cover, max-coverage [23] and Steiner tree problem [31].

In the existing literature, the features used in the LTP framework are problem specific and are carefully selected based on the algorithmic and heuristic literature on the optimization problem. In this paper, we explore if we can leverage it to solve a generic constraint programming formulation. Thus, instead of problem-specific features, we only use features that can be derived just by looking at the constraint and objective function formulation. This can be in the form of a primal graph where nodes correspond to variables in the formulation and two nodes share an edge if the corresponding variables co-occur in the scope of a constraint. Alternatively, the features can be derived from the bi(tri)-partite graph where both variables and constraints are represented as nodes and an edge exists between a variable node and a constraint node if and only if the corresponding constraint contains the variable corresponding to the variable node. We use generic features from the network analysis literature such as centrality measures, clustering coefficients and from the node embedding and the graph embedding literature briefly reviewed in the next subsection.

2.3 Graph Embeddings

A popular way to get feature representation for nodes in a graph is to build a low dimensional vector embedding for each node. The key idea behind these node embeddings is that similar nodes should have embeddings vectors that are close whereas dissimilar nodes should have embedding vectors that are further away. In the last decade, many different techniques have been developed for computing node embeddings that capture different notions of similarities between nodes (see [11] for a very recent survey). We briefly review a couple of node embedding techniques that we use in this work.

GLEE [24] leverages the simplex geometry of a graph and focuses on geometric properties instead of the traditionally used spectral properties. It extracts the Eigenvectors corresponding to the largest eigenvalues of the graph Laplacian and uses these vectors as the node embeddings. It aims to avoid just concentrating on the local community of nodes, it has fewer configuration parameters and so, we think that it is a good complement to the structure-based node attributes from [8].

The Multi-scale Attributed Node Embedding [21] concatenates node attributes from different neighbourhood sizes keeping the information from different distances separate.

In our LTP framework, we use these embeddings in isolation or together with network analytics features as the node representation. The classification models then learn a mapping from these representations to a binary target indicating whether the node is in the optimal

solution or not (whether the corresponding variable is 1 in the optimal solution or not). As indicated before, the goal is not to get an exact decision boundary, but only a simple boundary such that all variables on one side can be fixed. The node representations (and the embeddings involved in it) are crucial to obtain a good classification model.

2.4 Constraint Programming Background

For this work we use Savile Row [17], a constraint modelling tool that works with models written in the language Essence Prime and produces problem definitions for a range of solvers, including CP, SAT, MaxSAT, and SMT. Essence Prime is a mid-level modelling language which allows boolean and integer decision variables (including in matrix form), along with quantifiers, the common logic and arithmetic operators, and a selection of global constraints. Essence Prime has some similarities to other constraint modelling languages such as OPL [25] and MiniZinc [16]. Savile Row reformulates constraint models to apply various optimisations and simplifications and provides a variety of encodings for SAT, MaxSAT and SMT backend solvers. We use the MaxSAT backend because it is broadly the most successful solver type for the problems we are considering. The backend solver is WMaxCDCL, which scored very highly in the MaxSAT Evaluation 2023 [4]. We use the default MaxSAT encoding implemented in Savile Row except that all linear constraints (i.e. pseudo-Boolean and linear integer constraints) are encoded with GGPW [5] with automatic AMO detection [1].

Internally Savile Row represents constraints in an abstract syntax tree (AST). Common operators such as $\leq$, $=$, $+$, and $\vee$ are represented as nodes in the AST, as are named global constraints, constants, and references to decision variables. The objective function is also represented as an AST. Decision variables are listed separately. The indices of a variable in a matrix can be recovered from the variable name (e.g. `M[2,2]` would be named `M_00002_00002` when LTP is applied). Some LTP features are extracted directly from the AST representation (e.g. coefficient of a variable in the objective function). Other features are generated by traversing the constraint ASTs, for example to find the positions of variables within a constraint. A graph is created to represent the constraint network, as an intermediate step between the AST representation of constraints and the vector of numerical features that is used by LTP (as described below).

3 Learning-to-Prune in CP

We outline the steps involved in learning-to-prune (LTP) for CP in Figure 1. The problem of deciding whether or not to fix a variable is treated as a classification problem. The features for training the classifier are derived in two main ways: some are extracted directly from Savile Row, such as the value contributed to the objective function by a given Boolean variable; other features are derived from a graph representation of the constraint model – these features can be embeddings of the structure of the graph around each node, or can also take into account the attributes of other nodes in the graph using attributed node embeddings.

A classification model is trained using a dataset of problem instances for which optimal solutions can be determined. The values of the decision variables in the optimal solutions become the classification targets. A trained classifier is then employed to fix a portion of the variables, reducing the number of decision variables for which a value must be determined. The classifier provides estimated probabilities for each class (i.e. for each value of the Boolean variable). We use the estimated probabilities to sort the assignments from most to least confident. The proportion of variables to assign is a tunable parameter of the system.

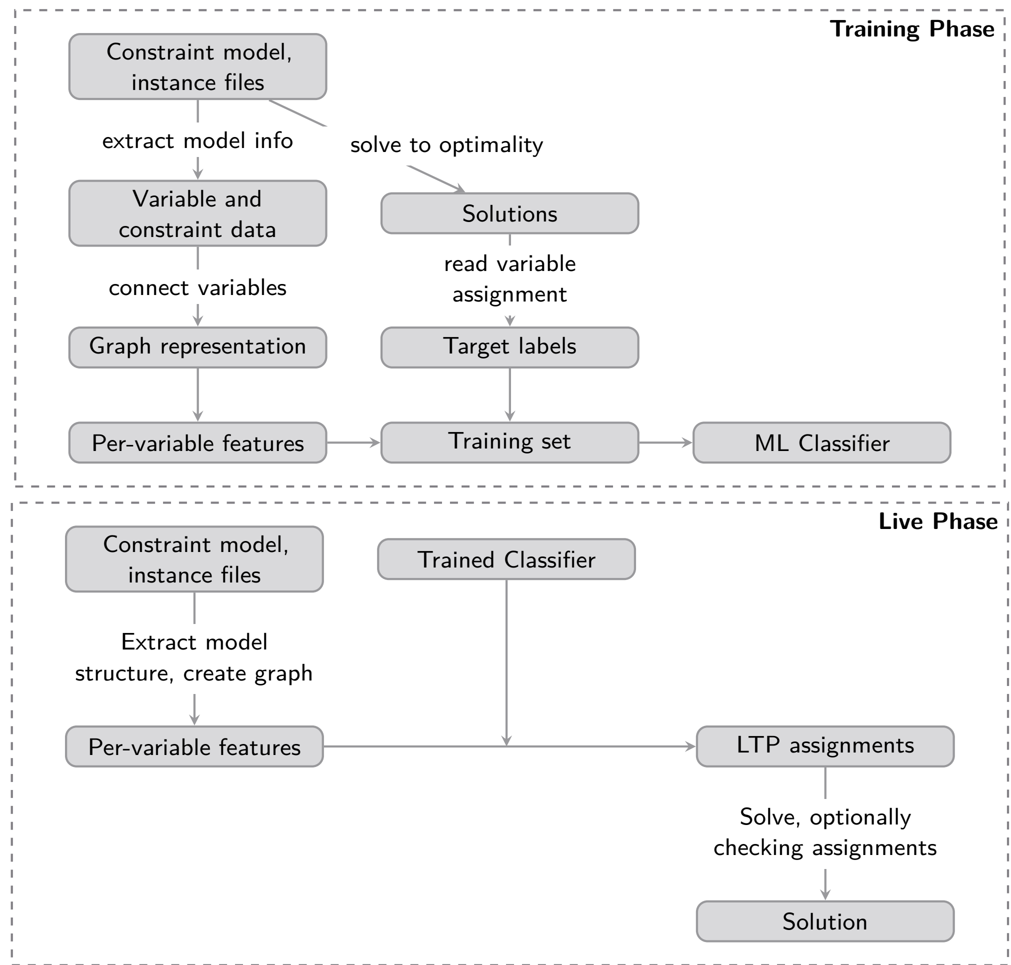


Figure 1 An outline of the steps involved in learning-to-prune for constraint programming.

Suppose for example that we wish to prune (assign) 10% of the variables, we would apply the classifier to all the relevant variables, sort into decreasing probability order, then take the first 10% in the list.

An additional step is implemented in Savile Row to check the assignments supplied by LTP. The assignments are applied one-by-one in the order they are supplied by LTP (highest probability predictions first) and at each step partial evaluation is applied to the set of constraints. After making an assignment A, if a constraint evaluates to *false* then A is reverted and the process continues with the next assignment in the sorted list. The loop ends when the required proportion of variables have been assigned (not including the assignments that were reverted). In the experiments below we report results without the additional check (named *force*) and with the extra check (named *sr-check*).

3.1 Feature Extraction and Graph Representation

In each of the models we consider for this work, the decision variables are represented by a single matrix of Booleans. Savile Row divides the matrices into individual boolean variables, but it is possible to retain information about where each variable fitted into the original matrix. We traverse the internal abstract syntax tree generated by Savile Row

to represent an instance and extract the following information for each (decision variable, constraint) pair:

Size of Matrix The size of the containing matrix, one integer for each dimension.
Position in Matrix How far along each dimension the variable occurs, as a proportion of the size of the matrix in each dimension.
Constraint Type An integer representing the type of constraint. Each distinct AST structure is given a unique integer. See Section 3.2 for more detail.
Position in Constraint Which position a variable occupies in the canonical representation of a constraint, e.g. if the constraint is $a < b$ then the position of variables a and b are 0 and 1 respectively.
Contribution to Objective If a variable is in the objective function, we record the coefficient associated with the decision variable.

Having extracted the above information for each (variable, constraint) pair in the model, we can construct two graph structures to represent the model.

Primal Graph Each decision variable is represented by a node; edges link any two variables which co-occur in the scope of a constraint. We annotate nodes with any relevant variable features as described above.
Bi(tri)-partite Graph We represent both variables and constraints as nodes, adding edges between them if a variable occurs in a constraint. Instead of a direct edge, we add two edges via an intermediate node whose attributes describe how a variable takes part in a constraint, i.e. the *Position in Constraint* describe above. The variable and constraint nodes have the relevant features from above as node attributes.

We can use either of these graphs to obtain node embeddings for each variable, capturing the position of the variable in question within the context of the constraint problem as represented in the graph (bipartite or primal). We make use of the following sets of features, which are concatenated in different combinations for different problem types.

SR-AST These are the variable-related attributes extracted directly from Savile Row's abstract syntax tree, as described above.
GRASSIA Grassia et al. [8] define 10 features (F1-F10) of nodes in a simple graph, considering a variety of graph-theoretic and statistical aspects of each node. We can apply these to either the primal or bipartite graph.
GLEE Node embedding in n-dimensional space considering only the structure of the graph [24], applicable to both primal and bipartite graphs.
MUSAE Node embeddings based on random walks which separately aggregate node features from different distances [21]. The node attributes are essentially what is extracted by the SR-AST featureset. Once again, these embeddings can be extracted from our primal or bipartite graphs.

3.2 Constraint Type

In constraint modelling languages, two constraints can be written (and represented in the AST) in different ways and yet be essentially the same constraint (on different variables). Before generating the type number for a constraint, we first apply partial evaluation, then sort all commutative-associative k-ary operators (such as sum) and all commutative binary operators (e.g. $=$). The alphabetical order is used because it will group together references to the same matrix and place them in a consistent order w.r.t. their indices in the matrix.

Then the AST is traversed in a defined order to obtain a duplicate-free list of variables. The variables are renamed to a canonical sequence of names. The AST is then converted to an Essence Prime string. We use a hash table to store a mapping from strings to distinct integers, allowing a new number to be assigned when a new constraint type is found.

This approach to labelling constraints with an integer type has some advantages. It supports the entire Essence Prime language (and will continue to do so as new operators or global constraints are added, without any changes to the LTP implementation). However it does not recognise degrees of similarity or difference between constraint types. For example, two sum equality constraints that differ only in the number of terms in the sum would simply be given different type numbers – there is no indication that the two constraints are very similar. Improving the type labelling of constraints could be an interesting future direction.

3.3 Machine Learning Model

In preliminary experiments we evaluated random forest classifiers and gradient boosted trees, both of which can train quickly on relatively small datasets. An added advantage of tree-based methods is they can learn from different types of features. Also, numerical features can take very different ranges of values without the need for normalisation. In our case we have a mixture of discrete features such as the degree of a node in the primal graph and continuous features such as the relative position of a variable within its matrix.

We found that gradient boosted trees perform better with our LTP setup. Tree boosting creates a sequence of trees where each new tree concentrates on predicting entries which have been previously misclassified. We use the implementation in scikit-learn [18], which is based on LightGBM [10]. We run 50 rounds of randomised search with 5-fold cross validation to tune the hyperparameters, targeting: the learning rate (or shrinkage), the maximum number of boosting iterations, maximum tree depth, and class weighting policy.

4 Experimental Evaluation

4.1 Models

Our focus in this paper is on optimisation problems where the main decision variables are Boolean and the optimization function is linear.

4.1.1 Maximum Clique

We include this problem because our work is in part inspired by the success of LTP on the maximum clique enumeration (MCE) problem in [8]. CP would not necessarily be a good choice for solving the maximum clique problem (much faster dedicated solvers exist), but it is a very simple example of a problem that our approach can be applied to. The model is shown in Listing 1 and consists of one type of constraint, which excludes any pair of nodes from both being in the maximum clique if there is no edge between them.

A large collection of graphs is available from `networkrepository.com` [20]. We have experimented on a number of graph collections from the DIMACS, miscellaneous and social networking categories. We present results for the *brock* collection which includes four graphs each of 200, 400 and 800 nodes and high density (0.5 to 0.75). We found that Savile Row and WMaxCDCL could find optimal solutions within a reasonable time only for the 200-node graphs.

Listing 1 Constraint model for the Maximum Clique Problem in Essence Prime

```
given graph_am : matrix indexed by [int(1..n),int(1..n)] of int(0,1)
letting NODES be domain int(1..n)

find inMaxClique : matrix indexed by [NODES] of bool
maximising sum(inMaxClique)

such that

forall n1 : int(1..n-1) .
  forall n2 : int(n1+1..n) . (graph_am[n1,n2]=0) ->
                              ( !inMaxClique[n1] \/ !inMaxClique[n2] )
```

Listing 2 Constraint model for the Combinatorial Auction Problem in Essence Prime

```
given nBids : int
given profit : matrix indexed by [int(1..nBids)] of int
given clashes : matrix indexed by [int(1..nClashes),int(1..2)]
                               of int(0..nBids)

find sold : matrix indexed by [int(1..nBids)] of bool

maximising (sum b : int(1..nBids). sold[b] * profit[b])

such that
forAll c : int(1..nClashes) . (!sold[clashes[c,1]] \/ !sold[clashes[c,2]])
```

4.1.2 Combinatorial Auction

In the Combinatorial Auction (also called Winner Determination) problem [19] a set of items are to be sold, and a set of bids have been received. Each bid has a monetary value and is for a subset of the items. A bid can be accepted in its entirety or rejected. Therefore any pair of bids that both bid on the same item cannot both be accepted. It can be written very simply in Essence Prime. The model is shown in Listing 2. There is only one type of constraint, and it says that two conflicting bids cannot both be accepted. The objective is to maximise the total value of accepted bids.

We obtain instances of this problem from the Combinatorial Auction Test Suite [14], which is able to generate sets of instances according to a variety of algorithms and underlying distributions of values. We have experimented with a number of these collections, including the *legacy* distributions L4 to L6, as well as the *scheduling*, *matching*, and *arbitrary* distributions. Below we present results for the L6 distribution which was challenging but still allowed us to obtain optimal objective values within 30 minutes.

4.1.3 Nurse Scheduling Problem

We use a simple version of the Nurse Scheduling problem [26] with preferences. The number of days in the schedule, the number of nurses, and the number of shift types are all given as parameters. For each nurse, day, and shift type, a preference value is given (typically in the range 1 to 4) where small values indicate preferred shifts. We are given a minimum number of nurses required for each day and shift type. The model is shown in Listing 3. The assignment of a nurse to a shift on a particular day is represented with a Boolean variable. The same variable is also included in the objective function, weighted by the corresponding preference value. Constraints state that a nurse must have at least two days off per week,

Listing 3 Constraint model for the Nurse Scheduling Problem in Essence Prime

```
given n_days :  int
given n_nurses : int
given n_shift_types : int
letting DAYS be domain int(1..n_days)
letting NURSES be domain int(1..n_nurses)
letting SH_TYPES be domain int(1..n_shift_types)
given covers : matrix indexed by [DAYS,SH_TYPES] of int
given prefs : matrix indexed by [NURSES,DAYS,SH_TYPES] of int

where (n_days % 7) = 0 $ assume full weeks

find sched : matrix indexed by [NURSES,DAYS,SH_TYPES] of bool

minimising (
  sum n : NURSES .
    sum d : DAYS .
      sum s : SH_TYPES .
        sched[n,d,s] * prefs[n,d,s]
)

such that

$ the shift coverage requirements are met
forAll d : DAYS .
  forAll s : SH_TYPES .
    sum( sched[..,d,s] ) >= covers[d,s],

$ each nurse only works one shift type in a day
forAll n : NURSES .
  forAll d : DAYS . sum(sched[n,d,..])=1,

$ each nurse should have 2/7 "off" shifts
forAll n : NURSES .
  sum( sched[n,..,n_shift_types] ) = ((2*n_days)/7)
```

and must do one shift type per day (where the shift types include the dummy ‘day off’ type). Also, each day and shift type must have enough nurses assigned to it. We use the same model as in Ansótegui et al. [1].

We draw instances from NSP-LIB [26], which provides 6 groups of problems, each for a fixed number of nurses – we have experimented with all the distributions and present in this paper the results for a random sample of instances from the N25 and N50 collections (both scheduling over 7 days with 25 and 50 nurses respectively).

4.2 Results

Our experiments were carried out on the University of York’s Viking compute cluster, with AMD EPYC3 7643 CPUs. We allowed 16GB RAM for solving runs and 64GB for training runs. We set a 30 minute time-out for the solving runs. Table 1 shows which featuresets and graph representations we used with each set of problems for which we present results in this section.

4.2.1 Maximum Clique

First we consider the maximum clique problem. As mentioned above, we use the *brock* graphs. In this case we were not able to train and test on the same instance size because

Table 1 Featuresets and graph type used for our experiments

Problem	Graph	Featureset			
		SR-AST	GRASSIA	GLEE	MUSAE
Maximum Clique	primal		✓	✓	
Combinatorial Auction	primal	✓	✓		
Combinatorial Auction	bipartite	✓	✓		✓
Nurse Scheduling	bipartite	✓		✓	✓

there were not enough instances of size 200 available. Instead we trained LTP using the four instances of size 200 where we were able to find an optimal solution.

The *brock* graphs of size 400 are not solved within 10 hours with our setup. However, if we apply LTP with 45% pruning then we can find solutions within 10 minutes for each of the four graphs. In `networkrepository.com` [20] the lower bound for maximum cliques given is 22; the maximum clique size found with LTP is 21 for three of the graphs and 20 for the other. It is possible that the lower bound of 22 is in fact the optimal value. If this is the case, then we have an optimality gap of 4.5% for three of the four instances, and a speedup of more than 60 times for all four.

4.2.2 Combinatorial Auctions

The results for combinatorial auction are presented in Figure 2. Each pane plots the set of instances as points on a scatterplot with speedup on the x-axis and the optimality gap percentage (i.e. percentage loss when using LTP) on the y-axis. Each pane is labelled with the percentage of variables that LTP assigned, which is a parameter of the framework. Rows of panes are labelled with the graph type (primal or bipartite), and whether *sr-check* is used.

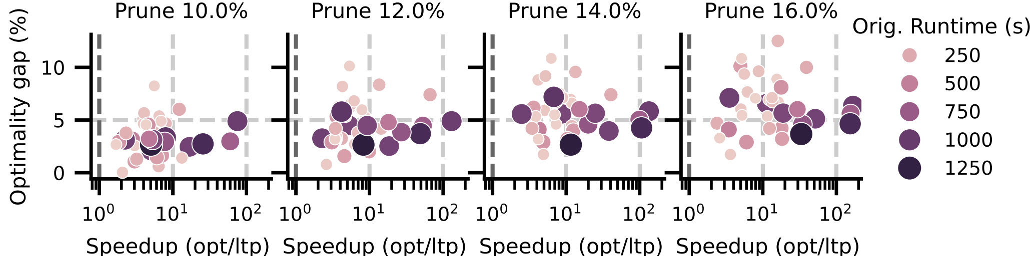
Unsurprisingly, LTP is somewhat more effective when given the primal graph. All constraints are of the same type, and all are binary, meaning that the set of constraints can be perfectly represented in the simpler and smaller primal graph.

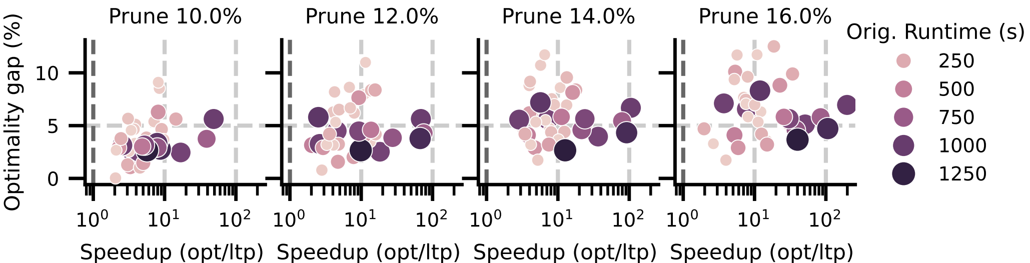
Speedups can be in excess of 100 times with an optimality gap of approximately 5% when 16% of variables are pruned. However, at 16% LTP is quite aggressive and we are seeing optimality gaps of more than 10% for some instances. The less aggressive setting of 10% pruning can still provide some strong speedups on hard instances, while a large majority of instances have an optimality gap well within 5%. We also observe that the larger speedups tend to occur for the harder instances.

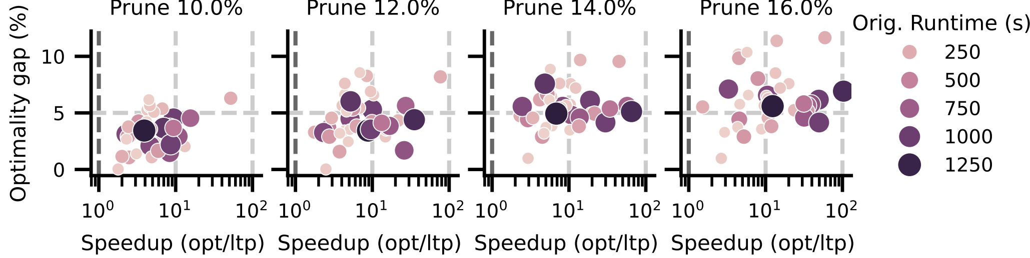
4.2.3 Nurse Scheduling

Results for nurse scheduling are presented in Figure 3. In this case we use the bipartite graph because the primal graph would lose most of the details of the constraint network. This problem class seems more challenging for LTP, however for the instance set N25 we found that LTP can learn enough to do some effective pruning with a low percentage pruned. As shown in the figure, in some cases we have a speedup of approximately 10 times with an optimality gap of less than 1%, however LTP is slowing down some of the instances. The *sr-check* setting is useful here, allowing higher pruning percentages to be used.

The larger N50 instances were more challenging for LTP, with speedups of less than 6 times (and in some cases less than 1, i.e. LTP slowed down solving). However the optimality







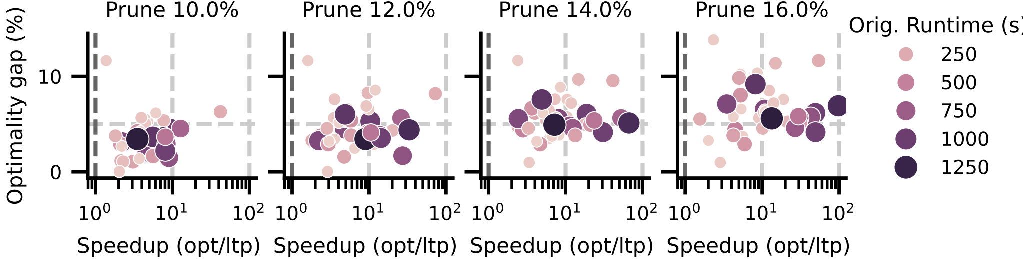


Figure 2 The results of applying Learning-to-Prune to the combinatorial auction problem (distribution L6). The plots show the optimality gap as a percentage on the vertical axis and speedup achieved for each instance in a test set on the horizontal axis. The colour and size of each point represents the hardness of the problem as measured by the total runtime for the initial non-pruned optimisation call.

Table 2 The runtime overhead of SAVILE ROW's assignment check. The table shows the median time spent by SAVILE ROW in processing an instance before calling the backend solver. †The *can* graphs are in the Miscellaneous section of the network repository [20] – in this test set we used 5 graphs of size 229, 256, 268, 292, and 445.

Problem	SR median time (s)	
	forced	checked
Combinatorial Auction (L6)	2.24	2.41
Nurse Scheduling (N25)	1.64	1.85
Nurse Scheduling (N50)	2.44	3.07
Maximum Clique (*can*†)	104.54	104.91

gaps are very small, suggesting that if LTP can be trained to avoid breaking constraints then perhaps larger speedups could be obtained with a larger optimality gap.

4.2.4 Overhead of the Savile Row Check

The basic assignment check carried out in SAVILE ROW incurs a runtime overhead. However, this overhead is in fact quite small in our experiments. Table 2 shows the difference in the time SAVILE ROW spends before calling the backend solver. For most sets of problems the difference in time is negligible. In the third row, the problem is much harder, with a median solving time of over 2 minutes - the extra half a second spent in SAVILE ROW checking the assignments is reasonable if a considerable solving speedup can be achieved.

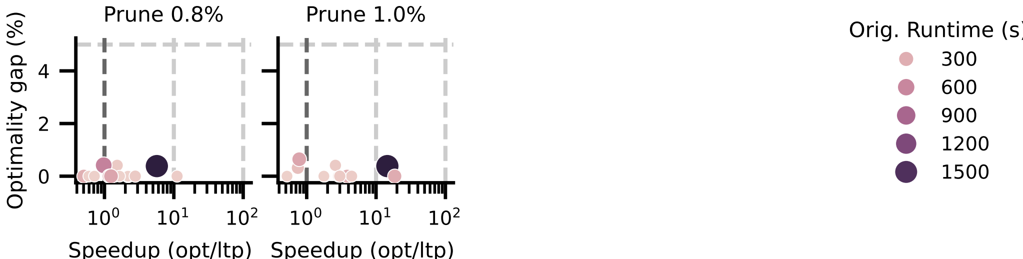
5 Conclusions

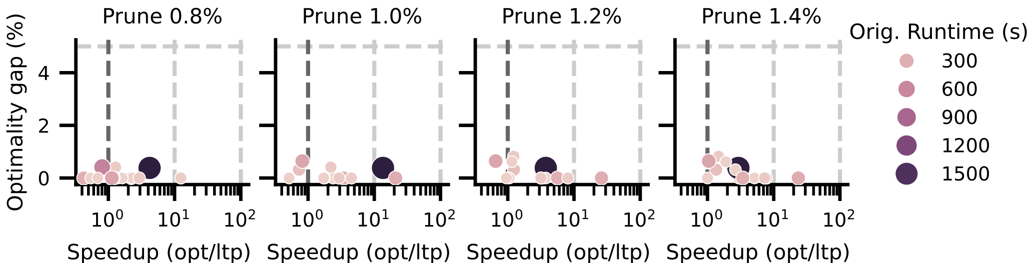
We have described early work on generalizing the learning-to-prune method for CP models written in a CP modelling language. Our overall goal is to learn from the distribution of instances how to solve future instances from the same distribution more efficiently, and we believe that LTP is a very promising approach. It can be trained with relatively little data because each training problem instance contributes many training examples, and also the models produced are relatively simple compared to other approaches. The advantages of LTP stem from answering a simpler question than end-to-end ML approaches. We presented promising results on three problem classes, showing that LTP deserves to be investigated further in CP.

Future Work

There are many ways in which this work could be extended, for example:

- Generating a better embedding of the variable-constraint graph by training a Graph Neural Network (GNN) to generate the embedding, alongside training the LTP classifier.
- Adapt *soft pruning* from the earlier work on problem-class-specific LTP, where (instead of directly assigning some of the variables) a constraint is placed on a subset of the variables stating that at least k are assigned as predicted by the LTP model. Soft pruning has been observed to reduce the optimality gap.
- Applying LTP pruning during search, where the solver allows custom propagators.
- A more thorough evaluation using multiple solver types, to investigate how LTP interacts with the choice of solver.





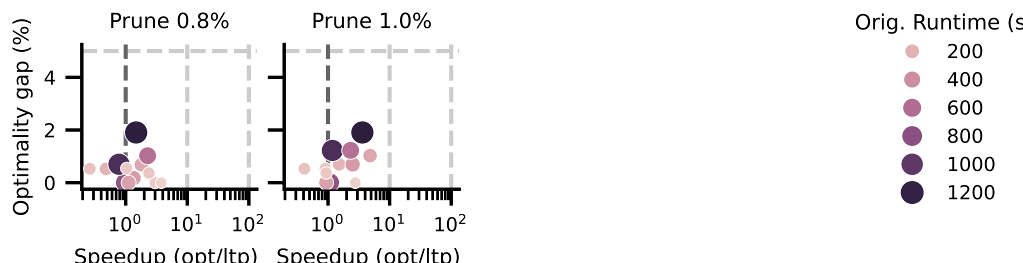


Figure 3 The results of applying Learning-to-Prune to the nurse scheduling problem (NSPLIB distributions N25 and N50). The plots show the optimality gap as a percentage on the vertical axis and speedup achieved for each instance in a test set on the horizontal axis. The colour and size of each point represents the hardness of the problem as measured by the total runtime for the initial non-pruned optimisation call.

References

1 Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z. Salamon, Josep Suy, and Mateu Villaret. Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-boolean constraints. In *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming*, pages 20–36, 2019.

2 Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *In International Conference on Learning Representations (ICLR), Workshop Track*, 2017.

3 Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

4 Jeremias Berg, Matti Järvisalo, Ruben Martins, and Andreas Niskanen. MaxSAT Evaluation 2023 : Solver and Benchmark Descriptions. URL: `https://helda.helsinki.fi/items/2b164fd8-ce8e-44d8-b760-6d79788d0730`.

5 Miquel Bofill, Jordi Coll, Peter Nightingale, Josep Suy, Felix Ulrich-Oltean, and Mateu Villaret. SAT encodings for pseudo-boolean constraints together with at-most-one constraints. *Artificial Intelligence*, 302, 2022. `doi:10.1016/j.artint.2021.103604`.

6 James Fitzpatrick, Deepak Ajwani, and Paula Carroll. Learning to sparsify travelling salesman problem instances. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 410–426. Springer, 2021.

7 Carla Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In *10th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, 2004.

8 Marco Grassia, Juho Lauri, Sourav Dutta, and Deepak Ajwani. Learning Multi-Stage Sparsification for Maximum Clique Enumeration, September 2019. `arXiv:1910.00517`, `doi:10.48550/arXiv.1910.00517`.

9 Nikolaos Karalias and Andreas Loukas. Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS*, 2020.

10 Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

11 Shima Khoshraftar and Aijun An. A survey on graph representation learning methods. *ACM Trans. Intell. Syst. Technol.*, 15(1), 2024.

12 Juho Lauri and Sourav Dutta. Fine-grained search space classification for hard enumeration variants of subset problems. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI*, pages 2314–2321. AAAI Press, 2019. `doi:10.1609/aaai.v33i01.33012314`.

13 Juho Lauri, Sourav Dutta, Marco Grassia, and Deepak Ajwani. Learning fine-grained search space pruning and heuristics for combinatorial optimization. *Journal of Heuristics*, 29(2):313–347, June 2023. `doi:10.1007/s10732-023-09512-z`.

14 Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM Conference on Electronic Commerce - EC '00*, pages 66–76, Minneapolis, Minnesota, United States, 2000. ACM Press. `doi:10.1145/352871.352879`.

15 Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey. *Computers & Operations Research*, 134:105400, 2021. URL: `https://www.sciencedirect.com/science/article/pii/S0305054821001660`, `doi:https://doi.org/10.1016/j.cor.2021.105400`.

16 N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, pages 529–543, 2007.

17 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, October 2017. doi:10.1016/j.artint.2017.07.001.

18 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.

19 Patrick Prosser. CSPLib problem 063: Winner determination problem (combinatorial auction). http://www.csplib.org/Problems/prob063.

20 Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29, Mar. 2015. URL: https://ojs.aaai.org/index.php/AAAI/article/view/9277, doi:10.1609/aaai.v29i1.9277.

21 Benedek Rozemberczki, Carl Allen, Rik Sarkar, and xx Thilo Gross. Multi-Scale attributed node embedding. *Journal of Complex Networks*, 9(1):1–22, April 2021. doi:10.1093/comnet/cnab014.

22 Yuan Sun, Andreas Ernst, Xiaodong Li, and Jake Weiner. Generalization of machine learning for problem reduction: A case study on travelling salesman problems. *arXiv preprint arXiv:2005.05847*, 2020.

23 Dena Tayebi, Saurabh Ray, and Deepak Ajwani. Learning to Prune Instances of k-median and Related Problems. *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 184–194, January 2022. doi:10.1137/1.9781611977042.15.

24 Leo Torres, Kevin S Chan, and Tina Eliassi-Rad. GLEE: Geometric Laplacian Eigenmap Embedding. *Journal of Complex Networks*, 8(2):cnaa007, April 2020. doi:10.1093/comnet/cnaa007.

25 Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, USA, 1999.

26 Mario Vanhoucke and Broos Maenhout. Nsplib–a nurse scheduling problem library: A tool to evaluate (meta-) heuristic procedures. In *Operational research for health policy: making better decisions, proceedings of the 31st annual meeting of the working group on operations research applied to health services*, pages 151–165, 2007.

27 Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

28 Haoyu Wang, Nan Wu, Hang Yang, Cong Hao, and Pan Li. Unsupervised learning for combinatorial optimization with principled objective relaxation. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2024.

29 L. Wu, P. Cui, J. Pei, and L. Zhao. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer, 2022.

30 Yunhao Yang and Andrew Whinston. A survey on reinforcement learning for combinatorial optimization. In *2023 IEEE World Conference on Applied Intelligence and Computing (AIC)*, pages 131–136, 2023.

31 Jiwei Zhang, Dena Tayebi, Saurabh Ray, and Deepak Ajwani. Learning to prune instances of steiner tree problem in graphs. In *Proceedings of the 11th International Network Optimization Conference*, pages 40–45. OpenProceedings.org, 2024.