Neural Local Search Enhanced with Knowledge Compilation

Alexandre Dubray[1]
DTAI, KU Leuven, Belgium

Hélène Verhaeghe
ICTEAM, UCLouvain, Belgium

Quentin Cappart
ICTEAM, UCLouvain, Belgium
Polytechnique Montréal, Canada
Mila - Québec AI Institute, Montréal, Canada

Siegfried Nijssen
DTAI, KU Leuven, Belgium

Abstract

Constraint satisfaction, the task of finding an assignment of variables that respects a set of constraints, is a key area in artificial intelligence, with many industrial applications. In general, many constraint satisfaction problems are NP-hard; hence, exhaustive searches are intractable. Local search is a popular paradigm for solving such problems: instead of exploring the entire search space, local search solvers transition, for a limited number of steps, from one assignment to another, guided by hand-crafted, problem-dependent heuristics. Recently, neural networks have been studied as a means of automatically learning heuristics for transitioning between assignments. A particular class of neural local search solvers uses neural networks as generative models to produce joint probability distributions over the problem's variables, conditioned on the current assignment. Then, transitioning to the next assignment is done by sampling from the generated probability distribution. However, these solvers do not integrate constraint reasoning; hence, any assignment can be generated during the local search. In this work, we propose to integrate knowledge compilation into neural local search solvers. By first compiling constraint satisfaction problems into multi-valued decision diagrams (MDDs) and sampling assignments in them, constraints can be applied to reduce the number of unsatisfying assignments in the sampling space. Our experiments show that using MDDs yields a sampling space that can be orders of magnitude smaller while incurring negligible overhead. Moreover, our MDD-based sampling procedure significantly improves the number of instances solved on Sudoku and graph colouring problems.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Constraint satisfaction, the task of finding an assignment to discrete variables that respects a set of constraints, and its optimisation version, are fundamental fields in artificial intelligence with numerous industrial applications, including scheduling, planning, resource management, and vehicle routing. Solving such problems often requires exploring an exponentially large solution space; hence, exhaustive searches may not finish within a reasonable time. Local search (LS) solvers alleviate this problem by avoiding classical search and instead transitioning from one assignment to another over a finite number of steps. Modifying an assignment

[1] Corresponding author

can be viewed as two processes: a destruction process that determines which variables of the current assignment to modify, and a repair process that assigns new values to the destroyed variables. The performance of LS solvers heavily depends on the quality of these two processes, which are often based on hand-crafted, problem-dependent heuristics.

In recent years, there has been a growing interest in leveraging machine learning, particularly neural networks, to solve combinatorial optimisation problems [4]. Neural local search (NLS) solvers are LS solvers that rely on a heuristic learned by a neural network to transition between assignments [20, 22]. They operate in two stages. First, during the training stage, the neural network parameters are learned on small or random instances using classical machine learning techniques. Then, at inference time (i.e., when solving a new instance), the neural network is used to transition from one assignment to another, starting from a randomly chosen one. NLS solvers considered in this work generate, from an assignment, a joint probability distribution on the problem's variables, and the local move consists of sampling new values from this distribution [20, 22]. To make sampling efficient, these solvers assume independence between the variables; hence, the neural network produces a distribution over their domain, and they are sampled independently.

These neural solvers present various advantages. First, they automatically learn heuristics during the training phase to explore the search space. This eases modelling optimisation problems, as there is no need to design the heuristic by hand. Moreover, it has been shown that heuristics learned by neural networks can outperform classical heuristics (i.e., they reduce the search space more in classical depth-first search) [5]. Moreover, by refining the assignment in a local-search-style manner, they will produce, with enough iterations, a satisfiable solution, in contrast to one-step neural predictors. Finally, modern methods employ neural architectures that, in theory, can operate on problems with arbitrary constraints [20, 22].

However, the main limitation of purely NLS solvers is that they do not integrate constraint reasoning and instead rely solely on the neural network to produce consistent distributions. For example, let us consider a problem with two binary variables linked by an inequality constraint, and the network produces uniform distributions on their domain. Then, sampling these distributions independently yields unsatisfiable assignments with probability 0.5. On the contrary, sampling one variable after another and integrating constraint reasoning always yields a satisfiable assignment. Moreover, although modern NLS solvers are designed to handle arbitrary constraints, they have only been evaluated on problems with simple constraints. For example, the `ConsFormer` solver has been benchmarked on problems containing only inequality constraints [22]. Learning to find solutions to problems with semantically more complex constraints (e.g., cardinality constraints, grammar constraints) seems currently out of reach for purely neural methods.

Constraint reasoning is a well-studied field in symbolic AI. *Knowledge compilation* is the task of transforming a knowledge base (e.g., a set of constraints) into a target language that supports various forms of reasoning. In this work, we consider the compilation of constraint satisfaction problems into *multi-valued decision diagrams* (MDDs). In this context, an MDD compactly encodes assignments to the variables of a CSP as a directed, layered, acyclic graph. During the compilation of such a data structure, reasoning is already applied, thereby removing infeasible solutions. Compiling an exact MDD yields the exact set of solutions; however, for NP-hard problems, this is often intractable. *Relaxed* MDDs encode a superset of the solutions; hence, they are a relaxation of exact MDDs. For instance, they can be obtained by bounding the width of the layers in an exact MDD and merging nodes that would otherwise be distinct, introducing non-solutions in the MDD [1, 6, 10].

The contribution of this work is the integration of constraint reasoning, via knowledge

compilation and MDDs, into neural local search solvers. We focus on such an integration at inference time, when solving a new, unseen instance. We propose to integrate constraint reasoning as a pre-processing step in the inference process of NLS solvers. First, the CSP is compiled into a relaxed MDD using knowledge compilation techniques. During this compilation process, constraints are applied, reducing the number of unsatisfying assignments stored in the MDD. During the neural local search, instead of sampling independently from the generated distributions, we sample an assignment in the MDD using these distributions. By sampling in a relaxed MDD with fewer infeasible assignments, we expect to produce better solutions, ultimately leading to finding feasible solutions in fewer iterations. We implemented this framework as an extension of `ConsFormer`, a recently proposed transformer-based NLS solver [22]. Our experiments show that integrating knowledge compilation significantly increases the number of solved instances on the Sudoku and the graph colouring problems while incurring a reasonable overhead.

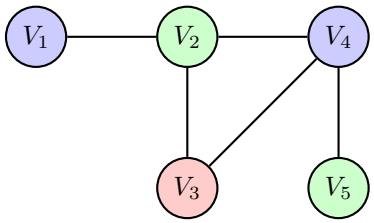
2 Problem Definition and Formalisation

In this section, we give the necessary information regarding constraint satisfaction problems, multi-valued decision diagram compilation, and neural local search solvers.

2.1 Constraint Satisfaction with Multi-valued Decision Diagrams

Let $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ be a constraint satisfaction problem over variables $\mathbf{X} = \{X_1, \dots, X_n\}$ with domain $\mathbf{D} = \{D_1, \dots, D_n\}$ and constraints $\mathbf{C} = \{C_1, \dots, C_m\}$. We denote $\mathbf{x} = \langle x_1, \dots, x_n \rangle$ an assignment to the variables, with $x_i \in D_i$. A constraint $C_j \in \mathbf{C}$ can be seen as a function over its scope $\mathbf{Y} \subseteq \mathbf{X}$, indicating which assignment is valid or not. For example, the `AllDifferent` $(\mathbf{Y})$ returns true if and only if all variables in $\mathbf{Y}$ take different values. A solution to $\mathcal{P}$ is an assignment respecting all constraints.

▶ **Example 1** (Graph Colouring). As a running example, let us consider the graph colouring problem. This problem consists of assigning colours from a limited set to the nodes of a graph such that no two adjacent nodes share the same colour. Below is an example of a graph with a valid colouring of its nodes, assuming the maximum number of colours is 3.



More formally, let $G(\mathbf{V}, \mathbf{E})$ be an undirected graph with $n = |\mathbf{V}|$ nodes and edges $\mathbf{E}$. To encode the graph colouring problem as a CSP, we model each node with a variable whose domain is the set of colours. That is, we have $\mathbf{X} = \{X_1, \dots, X_n\}$ with $D_1 = \dots, = D_n = \{1, \dots, k\}$. There is one constraint C per edge $(V_i, V_j) \in \mathbf{E}$ defined as $C \equiv X_i \neq X_j$.

In the context of constraint satisfaction problems, a multi-valued decision diagram is a rooted, directed, acyclic graph that encodes the solutions to $\mathcal{P}$. In such a representation, each path from the root to the sink encodes a solution; nodes correspond to variables, and outgoing edges correspond to domains' values. In this work, we consider layered-organised MDDs in which all nodes of a given layer correspond to the same variable.

▶ **Example 2** (Graph Colouring (cont.))**.** Let us consider again the graph colouring problem on the graph shown in Example 1. In the previous example, we showed a possible valid assignment, but many more exist. The MDD below encodes all feasible colour assignments. The decision variable associated with each layer is shown on the left, and labels on the edges are values from its domain.

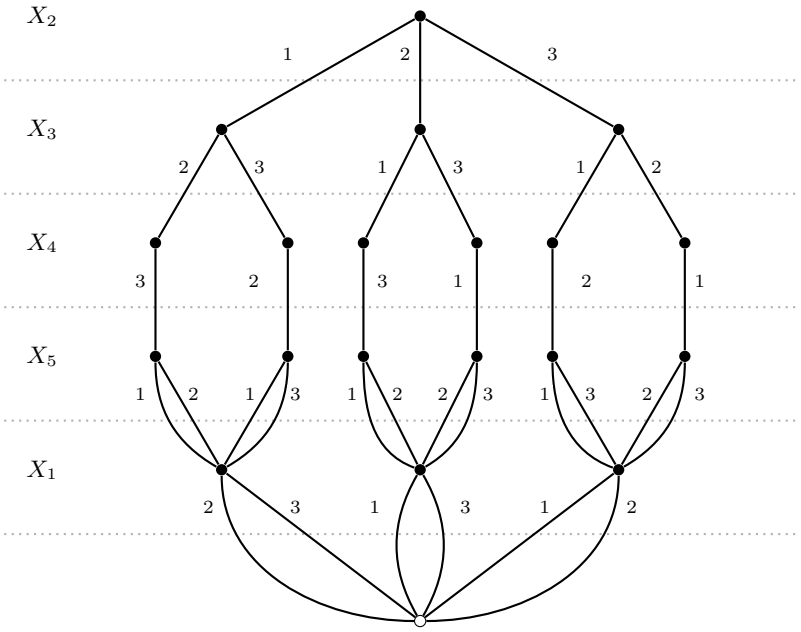


Figure 1 An exact MDD encoding all solutions to the graph colouring problem in Example 1

Each path from the root to the sink (the white dot) is a full assignment to the CSP's variables. It can be seen that constraints have been applied; for example, in the second layer, each node has two outgoing edges rather than three, ensuring that the assignments are consistent with the constraint $X_2 \neq X_3$.

Creating an MDD that represents the exact set of solutions is at least as hard as solving the initial problem; hence, it is not always possible to construct one. Relaxed MDDs, i.e., MDDs representing a superset of the solutions, are designed to be easier to construct, at the cost of not always providing a feasible solution. Typically, to construct such relaxed diagrams, a constraint is imposed on the width of the MDD (i.e., the maximum number of nodes in a layer). This implies merging nodes, which makes some assignments indistinguishable; hence, unsatisfying assignments are included in relaxed MDDs.

The framework we use in this work to construct (relaxed) MDDs is the *node splitting and refinement* process [1, 6, 9]. In this framework, the compilation process starts with a fully relaxed MDD of width 1 that encodes all assignments. In such an MDD, all paths go through the same nodes, and assignments are indistinguishable. Then, a node is selected for *splitting*: it is duplicated, and its incoming edges are redirected to the new nodes, while its outgoing edges are copied. Doing so increases the number of paths that are distinct in the succession of nodes they traverse, thereby enabling the distinction between assignments. Hence, constraint propagation can be applied to remove some unfeasible paths; this is the *refinement* step. Constraint propagation removes edges in the MDD and nodes that become unreachable from the root or cannot reach the sink. After this refinement step, the number of nodes in a layer can exceed the maximum authorised width, in which case some nodes are merged together.

▶ **Example 3** (Graph Colouring (cont.)). Let us illustrate the compilation process on our graph colouring example. The figure below shows a relaxed MDD at different stages of compilation, with a maximum width of 2. For clarity, we represent multiple edges with the same source and target as a single edge with multiple assignments on its label.

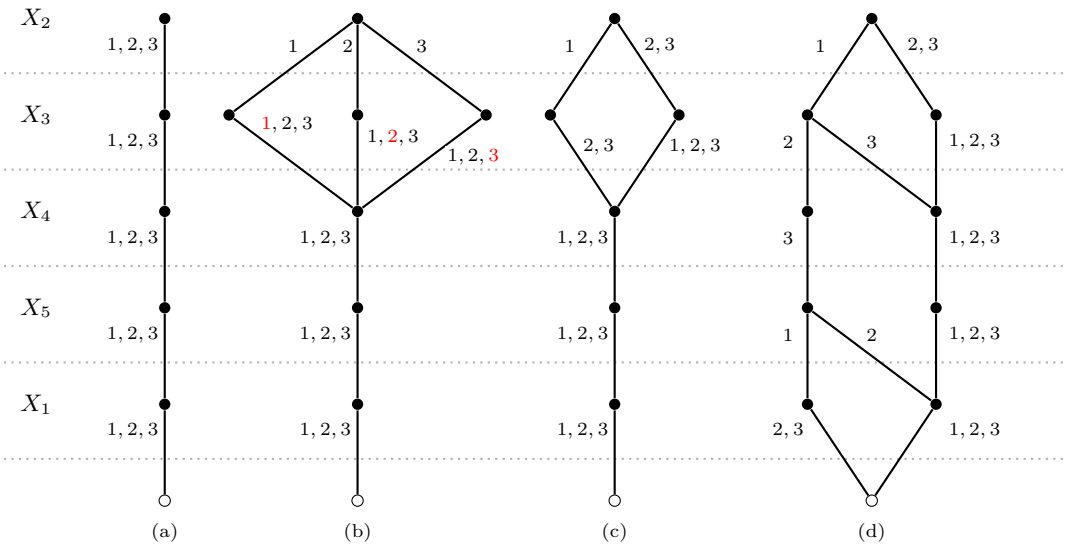


Figure 2 Example of compilation process, by splitting and refinement, of a relaxed MDD for the graph colouring problem of Example 1

In (a), the MDD is completely relaxed with one node per layer; this is the initial MDD that encodes all assignments. Then, in (b), the node in the second layer, associated with X_3, is selected for splitting and refinement. It can be seen that, in that case, different assignments emerge, reflecting the possible choices for X_2. The constraint $X_2 \neq X_3$ can be applied to remove invalid assignments in the second layer, which are highlighted in red. For example, if $X_2 = 1$ (leftmost outgoing edge), then X_3 can not be 1, and this edge is removed. At this step, the number of nodes in the second layer exceeds the maximum width constraint. Let us assume that the heuristic selects the rightmost nodes for merging, resulting in the relaxed MDD in (c). This process can be repeated for each layer, in order, resulting, for example, in the final relaxed MDD shown in (d). It can be seen that this MDD is smaller than the exact MDD shown in Figure 1, but includes invalid assignments.

2.2 Neural Local Search Solvers

Local search solvers work by producing a sequence of assignments $\langle \mathbf{x}^1, \ldots, \mathbf{x}^t \rangle$ to a CSP $\mathcal{P}$ until either $\mathbf{x}^t$ satisfies all constraints $\mathbf{C}$, or the solvers reach a limit T (e.g., a number of steps, or a time limit). LS solvers implement various heuristics to detect which part to modify to go from an assignment $\mathbf{x}^i$ to an assignment $\mathbf{x}^{i+1}$. NLS solvers, on the other hand, guide exploration of the search space using neural networks.

NLS solvers considered in this work proceed as follows [20, 22]. Intuitively, these solvers treat the problem's variables as random, and the neural network acts as a generative model that outputs a joint probability distribution over these variables, conditional on the current assignment and the constraints. Then, a new assignment can be sampled from this distribution. To make the sampling of this distribution tractable, independence between

the variables is assumed, so that each variable can be sampled independently in parallel, leveraging modern GPU architectures.

Formally, let Φ be a trained neural network and $\mathbf{x}^i$ an assignment to the variables $\mathbf{X}$ of a CSP $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$. Evaluating Φ results in n probability distributions, one over the domain of each variable. Hence, the joint probability distribution to sample from can be factorised as follows

$$P(X_1, \ldots, X_n \mid \mathbf{x}^i, \mathcal{P}) = \prod_{j=1}^{n} P(X_j \mid \mathbf{x}^i, \mathcal{P}).$$

For clarity, in the rest of this paper, we denote simply by $P(X_i)$ the conditional probability distribution $P(X_i \mid \mathbf{x}^{i-1}, \mathcal{P})$ when $\mathbf{x}^{i-1}$ and $\mathcal{P}$ are clear from the context.

3 Related Work

On the symbolic side, multi-valued decision diagrams have emerged as a practical approach for efficiently solving satisfiability and optimisation problems. For example, it has been used in standalone solvers based on depth-first search [3, 7, 14, 13], branch-and-bound [18], or in iterative solvers, refining a relaxed MDD with conflict analysis [11, 21]. Decision diagrams have also been integrated into constraint programming [1, 6, 9, 10]. In particular, some *global constraints* (i.e., constraints defining complex semantics over a subset of variables), have been adapted to work on MDDs [8, 10]: given a relaxed MDD, specialised algorithms can be applied on it to remove inconsistent arcs.

On the neural side, our work aligns with two recently proposed neural local search solvers. `ANYCSP` uses a graph neural network as a generative model to guide exploration of the search space [20]. They represent the CSP as a constraint-value graph with nodes for variables, domain values, and constraints. To guide the generation of the distribution, they label value-to-constraint edges with a binary indicator indicating whether a value will help satisfy the constraint. On the other hand, the solver we consider in this work, referred to as `ConsFormer`, is based on a transformer architecture[22]. The primal graph of the CSP is used in the relative positional encoding to guide the generative model. A key aspect of `ConsFormer` is that it changes only a subset of the assignment at each local search step. While in the original paper, this subset is randomly selected, the authors recently proposed an extension of `ConsFormer` that employs more advanced techniques [23].

4 MDD-Based Sampling for Neural Local Search Solvers

The intuition behind our method is as follows. NLS solvers such as `ConsFormer` [22] or `AnyCSP` [20] generate a probability distribution over all possible assignments and sample each variable independently. In the problem described in Example 1, all $3^5 = 243$ assignments can be sampled when using these solvers, but there are only 24 solutions. For more challenging CSPs, the proportion of solutions can be significantly lower. The main idea behind our approach is to leverage relaxed MDDs to reduce the number of unfeasible assignments that can be sampled. In Figure 2, the final relaxed MDD encodes 194 assignments, a reduction of roughly 20%. Hence, for a set of distributions $P(X_1), \ldots, P(X_n)$, the probability of sampling a feasible assignment is higher when sampling a path in a relaxed MDD, since there are fewer unfeasible assignments. Our new sampling procedure traverses the MDD from the root to the sink and, at each layer associated with variable X_i, samples an edge according to the neurally-generated probability distribution $P(X_i)$. Note that the generation of the distribution (i.e., the neural architecture) is not changed in this work.

Algorithm 1 `RelaxedMDDSampling`$(P(X_1), \dots, P(X_n), \mathcal{M})$

input : Probability distributions $P(X_1), \dots, P(X_n)$
input : A relaxed MDD $\mathcal{M}$ with root r and sink s
output : An assignment $\mathbf{x} = \langle x_1, \dots, x_n \rangle$ such that its path exists in $\mathcal{M}$ and is sampled proportionally to $\prod_{i=1}^{n} P(X_i)$

`/* Empty assignment of size` n `*/`
1 $\mathbf{x} \leftarrow \langle \_ \rangle^n$
2 $v \leftarrow r$
3 **while** $v \neq s$ **do**
4 $\quad X_i \leftarrow$ decision variable associated with node v
$\quad$ `/* Sample an edge from` v `to` u `with label` l `*/`
5 $\quad (v, u, l) \leftarrow$ `Sample(`$P(X_i), v$`)`
6 $\quad \mathbf{x}_i \leftarrow l$
7 $\quad v \leftarrow u$
8 **end**
9 **return x**

Algorithm 1 shows our new sampling procedure. It starts from an empty assignment (line 1) and sets the current node as the root of the MDD (line 2). Then, it traverses the MDD until the sink is reached (lines 3-8). At each step, an edge (i.e., a labelled pair or nodes) is sampled according to the distribution associated with the current layer's decision variable (line 5), and its label (l) is assigned to the decision variables (line 6). Then, the current node is updated by following the edge (line 7).

This sampling procedure can be integrated into a NLS solver using the loop shown in Algorithm 2. It starts by compiling a relaxed MDD of maximum width W, encoding assignments of the input CSP (line 1). If the MDD is empty (i.e., all nodes have been removed during constraint propagation), then the problem is unsatisfiable (e.g., it is impossible to colour a graph containing a clique of size 4 with 3 colours), and `UNSAT` is returned (line 2). If the MDD contains a path passing only through non-relaxed nodes, this path is assured to be a solution to the CSP, and the corresponding assignment is returned (line 3). For example, in Figure 2, the leftmost path of the final MDD contains only nodes that have not been merged; hence, it is guaranteed to be a valid colouring. If no exact path exists and the MDD is not empty, the local search (lines 6-15) starts from a random assignment (line 4). While a solution has not been found, or the maximum number of steps is not reached, distributions are generated over the variables' domain (line 7) similarly to classical NLS solvers, and a new assignment is sampled from the relaxed MDD, using Algorithm 1 (line 8).

Key aspects of Algorithm 2 are the `UNSAT` check as well as the extraction of a solution directly from the relaxed MDD, when it contains an exact path. During the initial compilation, we heuristically merge nodes linked to the most relaxed nodes; hence, this heuristic aims to preserve as many exact paths as possible. However, if this fails, and the NLS is started, it may be beneficial to leverage the neural network to guide a recompilation process. Hence, we propose to recompile the MDD periodically in a neurally guided manner (lines 9-13). During this recompilation, all solutions sampled since the last compilation are used to guide the merging heuristic. We implemented a simple neural-based heuristic to guide the merging process: we merge nodes whose outgoing edges are labelled with values that were least selected in the last assignments.

Algorithm 2 `MDD-NLS`$(\mathcal{P}, \Phi, T, W, r)$

input : A CSP $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$
input : A trained neural network Φ
input : A maximum number of steps T
input : A maximum width W for the relaxed MDD and a recompilation frequency r
output : `UNSAT`, a solution to the CSP, or the last found assignment in the local search

$\mathcal{M} \leftarrow$ `CompileRelaxMDD`$(\mathcal{P}, W, \langle\rangle)$
if $\mathcal{M}$ *is empty* **then return** UNSAT
if $\mathcal{M}$ *has an exact path with assignment* $\mathbf{x}^s$ **then return** $\mathbf{x}^s$
$\mathbf{x}^0 \leftarrow$ random assignment
$i \leftarrow 1$
while $\mathbf{x}^{i-1}$ *is not a solution of* $\mathcal{P}$ ***and*** $i \leq T$ **do**
  $P(X_1), \ldots, P(X_n) \leftarrow \Phi(\mathbf{x}, \mathcal{P})$
  $\mathbf{x}^i \leftarrow$ `RelaxedMDDSampling`$(P(X_1), \ldots, P(X_n), \mathcal{M})$
  if $(i \bmod r) = 0$ **then**
    $\mathcal{M} \leftarrow$ `CompileRelaxMDD`$(\mathcal{P}, W, \langle x^{i-r}, \ldots, x^i \rangle)$
    if $\mathcal{M}$ *is empty* **then return** UNSAT
    if $\mathcal{M}$ *has an exact path with assignment* $\mathbf{x}^s$ **then return** $\mathbf{x}^s$
  end
  $i \leftarrow i + 1$
end
return $\mathbf{x}^{i-1}$

5 Experimental Results

To showcase the effectiveness of our approach, we implemented our MDD-based neural local search solver as a modification of `ConsFormer` [22]. We evaluate our approach against `ConsFormer` on two satisfaction problems: Sudoku and graph colouring.

5.1 Experimental setup

We based our implementation on `ConsFormer`'s source code, which is freely available online[2]. We trained the neural networks using the hyperparameters reported by `ConsFormer`'s authors [22], and our baseline's performance matches theirs. We also use the same train and test datasets as the ones used by `ConsFormer` [22].

For the Sudoku, training instances consist of 9×9 grids with between 39 and 50 missing cells. For conciseness, we omit benchmarks on the test instances, as they are easily solved by all methods, and we focus on out-of-distribution instances that are 9×9 grids containing between 47 and 64 missing cells. For graph colouring, training instances consist of random graphs with 50 nodes that must be coloured with 5 colours. Out-of-distribution instances are graphs with 100 nodes that must be coloured with 5 colours. We set the maximum number of steps to $20,000$ for Sudoku and $10,000$ for graph colouring instances.

Regarding the MDD compilation, we used a modified version of the `MaxiCP` solver,

[2] At the time of the writing, only the source code for the initial `ConsFormer` is available; we leave experiments with the advanced subset selection strategy for further work.

implemented in Java, that integrates MDD compilation using the split and refine framework, as well as constraint propagation for `AllDifferent` and inequality constraints on MDDs [8, 19]. We limit the available memory to 16 GB for the compilation process. We use the following heuristic to order the variables in the MDDs. For Sudoku, the variables in the first layer correspond to prefilled values; for graph colouring, the first variable corresponds to the node of highest degree. Then we use the same heuristic for both problems: the variable selected at layer i is the one most constrained by variables in layers $j < i$. We use this heuristic because it has been shown to perform well for solving graph colouring problems with MDDs [21].

We implemented our sampling method in `PyTorch` [2], so that it is integrated with the current `ConsFormer` codebase and allows for solving multiple instances at the same time. Given a batch of instances, the MDDs are compiled using the `MaxiCP` Java implementation and are then stored in tensors. An MDD can be stored in two tensors, using a flattened representation. The N nodes of the MDD are implicitly stored using indexes ranging from 0 to $N-1$. The first tensor is a (N, D)-shaped tensor (D being the number of values in the variables' domain) encoding the structure of the MDD. It stores, for each node v and possible label l, the index of the node reached by following the outgoing edge of v labelled with l. If an edge does not exist, -1 is used, as it does not correspond to any node. The second tensor is a $(N, |\mathbf{X}|)$-shaped tensor, mapping each node to its decision variable. Using these two tensors, it is possible to traverse the MDD and assign values to the decision variables.

We evaluate two versions of our method. First, the complete implementation of our method, denoted `ConsFormer-MDD`. Then, to evaluate the impact of sampling paths in the MDD, compared to just detecting `UNSAT` and exact paths, we evaluate a version of our method in which the MDD is compiled, but the distributions are sampled independently, as in the initial `ConsFormer`, denoted `ConsFormer-MDD-mix`.

5.2 Sudoku

We implemented Sudoku using the classical encoding with one `AllDifferent` constraint per row, column, and block. Figure 3 shows the proportion of solved instances over the number of iterations. We show one cactus plot per configuration, where $r = 0$ indicates no recompilation. We also show a virtual best solver (VBS) across all configurations. When the maximum width for MDD compilation is 1, we do not recompile because no splitting or merging is performed; hence, recompiling would result in the same MDD.

It can be seen that, overall, integrating knowledge compilation results in a higher number of solved instances, even when the maximum width is 1. In this configuration, only the `AllDifferent` propagator is launched (no nodes are split), but infeasible values are still removed (e.g., if a cell is prefilled with 8, this value is removed from all other cells in its row, column, and block). Due to the number of missing values and the strong propagation power of the `AllDifferent` constraint, the number of unfeasible assignments removed is high. On average, in our benchmark, `ConsFormer` samples from 7×10^{59} assignments while the average number of paths encoded in the relaxed MDDs is 6×10^{40}. Applying constraint propagation drastically reduced the number of assignments to sample from.

Increasing the maximum width for compilation does not significantly increase the overall number of solved instances, but it increases the number of instances for which an exact path is found during compilation. Moreover, periodically recompiling MDDs helps with the solving process, as small bumps are observed at recompilation iterations, especially the first.

Finally, it can be seen that our mixed approach `ConsFormer-MDD-mix` performs better than the basic `ConsFormer`, but worse than our full approach `ConsFormer-MDD`. Hence, on this problem, sampling paths in the MDD is beneficial.

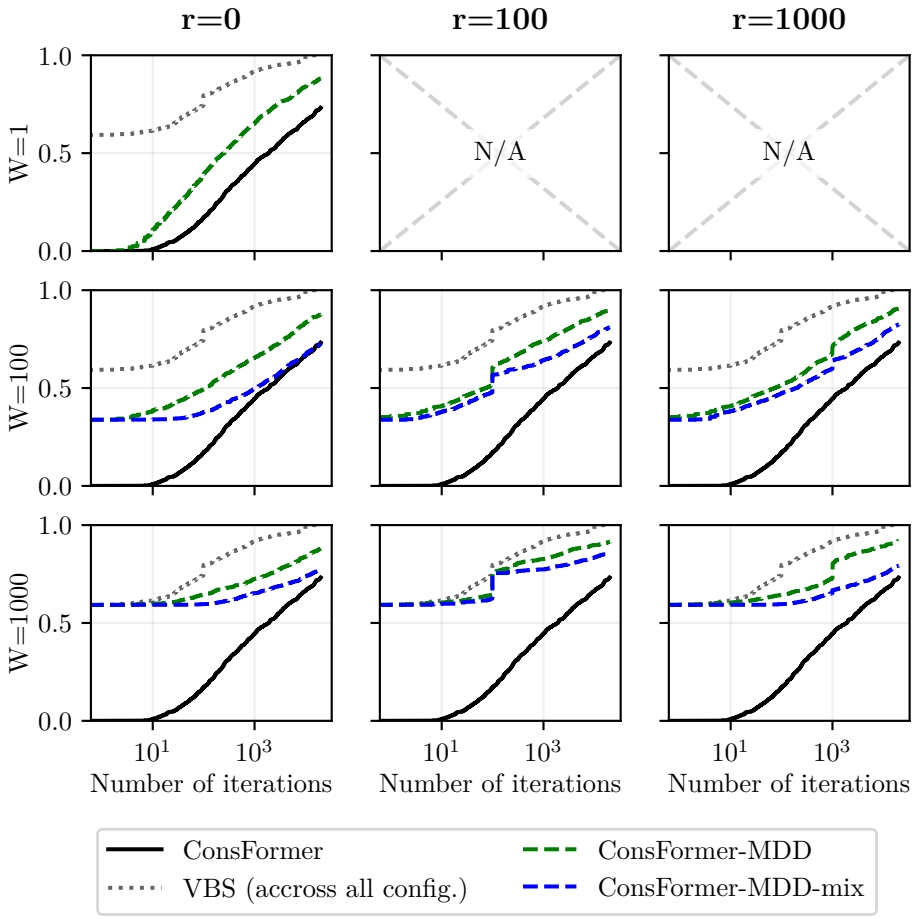


Figure 3 Proportion of instances solved over the number of iterations on the out-of-distribution dataset for the Sudoku (47 to 64 missing cells) for various configurations for the MDD (re-)compilation.

5.3 Graph Colouring

Now, we consider the graph colouring problem, as described in Example 1. We encode this problem with one variable per node and inequality constraints between adjacent nodes. To be efficient, we must implement two optimisations. First, we detect symmetrical nodes using `Nauty` [12]. Such nodes can be swapped without changing the graph's structure; hence, given variables $X_1, \ldots, X_m$ representing m symmetrical nodes, we impose $X_1 \leq \ldots, \leq X_m$. Moreover, we detect cliques in the graph to strengthen constraint propagation. Indeed, nodes in a clique must take different values, which can be encoded using an `AllDifferent` constraint, whose propagation is stronger than the pairwise inequality constraint [16, 17]. However, finding all cliques is computationally intractable; hence, given a graph colouring problem with k colours, we find all cliques of size at most $k+1$ using `Cliquer` [15]. Finding larger cliques is unnecessary, since finding a clique of size $k+1$ makes the instance unsatisfiable.

Figures 4 and 5 show the proportion of solved instances over the number of iterations for, respectively, the in-distributions and out-of-distribution datasets. It can be seen that, while MDD-based sampling helps solve more instances, the improvement is less marked than on the Sudoku. This can be explained by considering how much the search space is

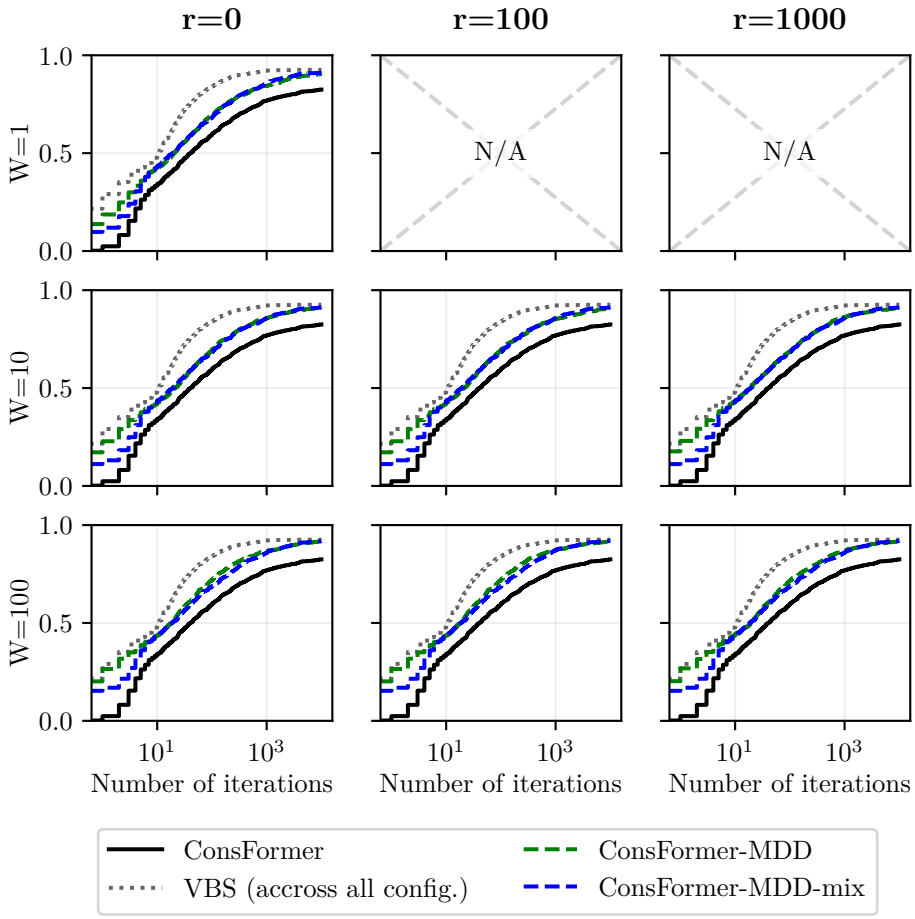


Figure 4 Proportion of instances solved over the number of iterations on the in-distribution dataset for the graph colouring (50 vertices and 5 colours) for various configurations for the MDD (re-)compilation.

reduced when applying constraint propagation (i.e., MDD of width 1). While for Sudoku, the number of encoded paths was 10^{19} times smaller, for graph colouring, a 1-width relaxed MDD encodes between 1% and 2% of the initial number of assignments. Due to this weaker pruning capability, the maximum width allowed during compilation is also smaller. In our experiments, instances with 100 nodes reach an out-of-memory state when compiled with a maximum width $W = 100$.

However, a key advantage of our method is its ability to detect `UNSAT` instances. While `ConsFormer` does not check for unsatisfiable instances, our method handles that seamlessly. `AllDifferent` constraints on cliques detect roughly 10% of UNSAT instances in both in-distribution and out-of-distribution datasets.

5.4 Runtime Overhead

Finally, we analyse the overhead induced by Algorithm 2 compared to `ConsFormer`. There are two sources of overhead, one from compiling relaxed MDDs and one from the sequential sampling of an assignment, compared to the parallel sampling of `ConsFormer`. Table 1 shows

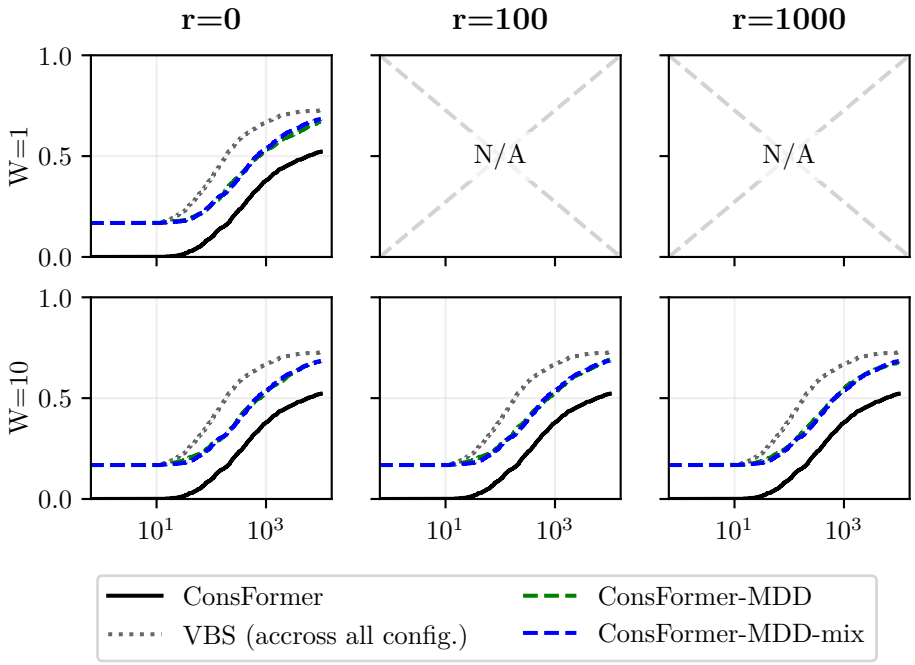


Figure 5 Proportion of instances solved over the number of iterations on the out-of-distribution dataset for the graph colouring (100 vertices and 5 colours) for various configurations for the MDD (re-)compilation.

the average overhead of our method. For each problem, we show the average time to compile the relaxed MDDs for various maximum widths and the slowing factor of the sequential sampling. We also report the average time required by `ConsFormer` to solve a problem instance.

The cost of compiling a relaxed MDD can be controlled with the maximum width, allowing a balance between search space reduction and runtime. For example, for the graph colouring instances we evaluated, many instances can be detected as infeasible by compiling a relaxed MDD of width 1, which takes one second for the larger instances.

The overhead induced by our sequential sampling is also limited. While the number of sampling steps increases linearly with the number of variables, the slowing factor on the iteration's runtime increases more slowly. For example, for the out-of-distribution graph colouring instances, our sampling strategy traverses a MDD with 100 layers, but the iterations' runtime is less than twice the `ConsFormer`'s iteration runtime. For the smaller graphs, the overhead is more significant, but our method is still less than an order of magnitude slower than the initial `ConsFormer`. Further development is needed to optimise our MDD tensor implementation to reduce this overhead.

Overall, there is a tradeoff between compilation time and epoch overhead. For instance, in Sudoku, there is no overhead to running MDD-based sampling, which improves the number of solved instances. On the other hand, for graph colouring, our current implementation incurs overhead and does not scale well with MDD width, without increasing the solver's capabilities. In this case, a mix strategy such as `ConsFormer-MDD-mix` is preferable.

Problem	Baseline avg. runtime (s)	Metric	W=1	W=10	W=100	W=1000
Sudoku	275.23	Compilation time (s)	0.22 ± 0.01	-	0.68 ± 0.33	2.20 ± 2.41
		Iteration runtime increase factor	1.00	-	1.02	1.19
GCOL ($\|V\| = 50$)	24.60	Compilation time (s)	0.49 ± 0.14	1.71 ± 1.00	7.61 ± 5.55	-
		Iteration runtime increase factor	3.40	3.37	3.34	-
GCOL ($\|V\| = 100$)	92.88	Compilation time (s)	1.17 ± 0.42	4.82 ± 1.76	-	-
		Iteration runtime increase factor	1.70	1.67	-	-

Table 1 Runtime overhead of MDD-based sampling for different maximum widths. For each problem, the average runtime needed by ConsFormer to solve one instance is indicated (second column). Dashed entries indicate configurations that have not been run in our experiments. The top line of each problem shows the average runtime (in seconds) for compiling the relaxed MDD, along with the standard deviation. The bottom line for each problem shows the slowing factor of MDD-based sampling.

6 Conclusion

In this work, we proposed a new methodology for solving constraint satisfaction problems using neural local search solvers. While the neural search solvers we studied in this work do not integrate constraint reasoning into their local search, we propose integrating knowledge compilation to guide the search towards higher-quality assignments. In a first phase, our method compiles the CSP into a relaxed multi-valued decision diagram, enabling the application of constraints and thereby removing infeasible assignments. Then, during the local search, generated assignments are restricted to those that appear in the relaxed MDD. We implemented our framework as an extension of the `ConsFormer` solver, using `PyTorch`. Our experiment shows that integrating constraint reasoning into neural local search solvers significantly improves their performance while incurring a reasonable overhead.

In this paper, we implemented our framework using existing solvers, that is `ConsFormer` and `MaxiCP`. `MaxiCP` has not been designed to be integrated with neural local search solvers. Moreover, when recompiling relaxed MDDs, the compiler starts from scratch, considering only the solutions generated during the previous steps of local search. We leave as future work the design of a compiler aimed towards neural solvers. In this work, we focused on modifying the inference stage of NLS solvers. How to integrate knowledge compilation into the training pipeline for NLS solvers remains an open question. Another interesting research direction is applying our framework to optimisation problems, as they are encoded differently than CSP's when solved using MDDs. Finally, problems with semantically more challenging constraints are also left for future work.

References

1 H. R. Andersen, T. Hadzic, J. N. Hooker, and P. Tiedemann. A Constraint Store Based on Multivalued Decision Diagrams. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, volume 4741, pages 118–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74970-7_11.

2 Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard

401 Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith
402 Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Trans-
403 formation and Graph Compilation. In *Proceedings of the 29th ACM International Conference*
404 *on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages
405 929–947, La Jolla CA USA, April 2024. ACM. doi:10.1145/3620665.3640366.
406 **3** David Bergman, Andre A. Cire, Willem-Jan Van Hoeve, and John Hooker. *Decision Diagrams*
407 *for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer
408 International Publishing, Cham, 2016. doi:10.1007/978-3-319-42849-9.
409 **4** Quentin Cappart, Tias Guns, Michele Lombardi, Gilles Pesant, and Dimos Tsouros. Com-
410 bining Constraint Programming and Machine Learning: From Current Progress to Future
411 Opportunities. *Journal of Artificial Intelligence Research*, 84, 2025.
412 **5** Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. SeaPearl: A
413 Constraint Programming Solver Guided by Reinforcement Learning. In Peter J. Stuckey,
414 editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*,
415 volume 12735, pages 392–409. Springer International Publishing, Cham, 2021. doi:10.1007/
416 978-3-030-78230-6_25.
417 **6** Rebecca Gentzel, Laurent Michel, and W.-J. Van Hoeve. HADDOCK: A Language and
418 Architecture for Decision Diagram Compilation. In Helmut Simonis, editor, *Principles and*
419 *Practice of Constraint Programming*, volume 12333, pages 531–547. Springer International
420 Publishing, Cham, 2020. doi:10.1007/978-3-030-58475-7_31.
421 **7** Xavier Gillard, Vianney Coppé, and Pierre Schaus. Ddo, a generic and efficient framework
422 for mdd-based optimization. In *International Joint Conference on Artificial Intelligence*
423 *(IJCAI20)*, 2020.
424 **8** Amaury Guichard, Laurent Michel, Hélène Verhaeghe, and Pierre Schaus. Towards Bound
425 Consistency for the No-Overlap Constraint Using MDDs. *arXiv preprint arXiv:2601.14784*,
426 2026. arXiv:2601.14784.
427 **9** Tarik Hadzic, John N. Hooker, Barry O'Sullivan, and Peter Tiedemann. Approximate
428 Compilation of Constraints into Multivalued Decision Diagrams. In David Hutchison, Takeo
429 Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor,
430 Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos,
431 Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Peter J. Stuckey, editors, *Principles and*
432 *Practice of Constraint Programming*, volume 5202, pages 448–462. Springer Berlin Heidelberg,
433 Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-85958-1_30.
434 **10** Samid Hoda, Willem-Jan Van Hoeve, and J. N. Hooker. A Systematic Approach to MDD-Based
435 Constraint Programming. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg,
436 Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan,
437 Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard
438 Weikum, and David Cohen, editors, *Principles and Practice of Constraint Programming –*
439 *CP 2010*, volume 6308, pages 266–280. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
440 doi:10.1007/978-3-642-15396-9_23.
441 **11** Anthony Karahalios and Willem-Jan van Hoeve. Column Elimination: An Iterative Approach
442 to Solving Arc-Flow Formulations.
443 **12** Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of symbolic*
444 *computation*, 60:94–112, 2014.
445 **13** Quentin Meurisse, Pierre Schaus, Renaud De Landtsheer, Fabian Germeau, Roger Kameugne,
446 and Xavier Gillard. DDOLib: Librairie pour les Diagrammes de Décision en Optimisation. In
447 *ROADEF 2026*, 2026.
448 **14** Laurent Michel and Willem-Jan Van Hoeve. CODD: A Decision Diagram-Based Solver for
449 Combinatorial Optimization. In Ulle Endriss, Francisco S. Melo, Kerstin Bach, Alberto
450 Bugarín-Diz, José M. Alonso-Moral, Senén Barro, and Fredrik Heintz, editors, *Frontiers in*
451 *Artificial Intelligence and Applications*. IOS Press, October 2024. doi:10.3233/FAIA240997.
452 **15** Sampo Niskanen and Patric Östergård. Cliquer user's guide, version 1.0. 2003.

16 Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI*, volume 94, pages 362–367, 1994.

17 Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

18 Isaac Rudich, Quentin Cappart, and Louis-Martin Rousseau. Improved peel-and-bound: Methods for generating dual bounds with multivalued decision diagrams. *Journal of Artificial Intelligence Research*, 77:1489–1538, 2023.

19 Pierre Schaus, G. Derval, A. Delecluse, L. Michel, and P. V. Hentenryck. Maxicp: A constraint programming solver for scheduling and vehicle routing. *URL https://github. com/aia-uclouvain/maxicp*, 2024.

20 Jan Tönshoff, Berke Kisin, Jakob Lindner, and Martin Grohe. One Model, Any CSP: Graph Neural Networks as Fast Global Search Heuristics for Constraint Satisfaction, August 2022. `arXiv:2208.10227`, `doi:10.48550/arXiv.2208.10227`.

21 Willem-Jan van Hoeve. Graph coloring with decision diagrams. *Mathematical Programming*, 192(1):631–674, 2022.

22 Yudong W. Xu, Wenhao Li, Scott Sanner, and Elias B. Khalil. Self-Supervised Transformers as Iterative Solution Improvers for Constraint Satisfaction, June 2025. `arXiv:2502.15794`, `doi:10.48550/arXiv.2502.15794`.

23 Yudong W. Xu, Wenhao Li, Scott Sanner, and Elias B. Khalil. Large Neighborhood Search meets Iterative Neural Constraint Heuristics. *arXiv preprint arXiv:2603.20801*, 2026. `arXiv:2603.20801`.