Journal of Machine Learning Research 23 (2022) 1-40 Submitted 12/21; Published 8/22

Constraint Reasoning Embedded Structured Prediction

Nan Jiang JIANG631@PURDUE.EDU
Department of Computer Science
Purdue University
West Lafayette, Indiana, USA.

Maosen Zhang ZHANGMAOSEN.PKU@BYTEDANCE.COM
ByteDance
Beijing, China.

Willem-Jan van Hoeve VANHOEVE@ANDREW.CMU.EDU
Tepper School of Business
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA.

Yexiang Xue YEXIANG@PURDUE.EDU
Department of Computer Science
Purdue University
West Lafayette, Indiana, USA.

Editor: Maya Gupta

Abstract

Many real-world structured prediction problems need machine learning to capture data distribution and constraint reasoning to ensure structure validity. Nevertheless, constrained structured prediction is still limited in real-world applications because of the lack of tools to bridge constraint satisfaction and machine learning. In this paper, we propose **CO**nstraint **RE**asoning embedded **S**tructured **P**rediction (CORE-SP), a scalable constraint reasoning and machine learning integrated approach for learning over structured domains. We propose to embed decision diagrams, a popular constraint reasoning tool, as a fully-differentiable module into deep neural networks for structured prediction. We also propose an iterative search algorithm to automate the searching process of the best CORE-SP structure. We evaluate CORE-SP on three applications: vehicle dispatching service planning, if-then program synthesis, and text2SQL generation. The proposed CORE-SP module demonstrates superior performance over state-of-the-art approaches in all three applications. The structures generated with CORE-SP satisfy 100% of the constraints when using exact decision diagrams. In addition, CORE-SP boosts learning performance by reducing the modeling space via constraint satisfaction.

Keywords: Constraint Reasoning, Decision Diagrams, Structured Prediction.

1. Introduction

The emergence of large-scale constraint reasoning and machine learning technologies have impacted virtually all application domains, including marketing, linguistics, operations, retail, robotics, and health care. Constraint reasoning has traditionally been applied to

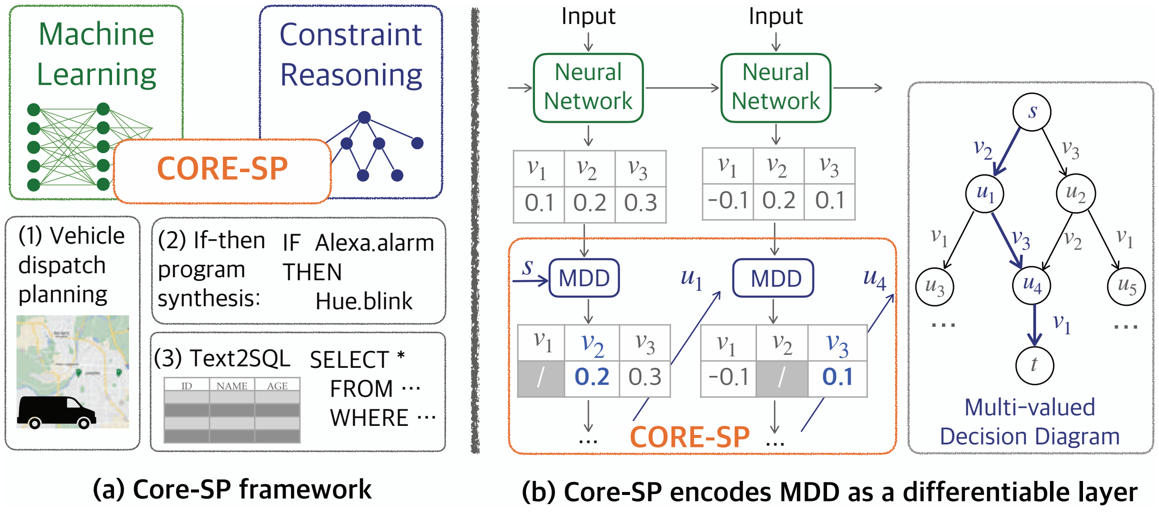


Figure 1: **(a)** Our proposed Core-Sp framework embeds constraint reasoning in machine learning for structured prediction. We demonstrate the effectiveness of Core-Sp on vehicle dispatching service, if-then program synthesis, and Text2SQL generation tasks. **(b)** At a high level, Core-Sp (in orange colored box) is a fully differentiable layer that simulates a path descending in the corresponding decision diagram. Core-Sp filters out the infeasible output from the structured output to ensure constraint satisfaction.

building *prescriptive* models that generate solutions for strategic, tactical, or operational use (Choi et al., 2012). It requires a precise problem description and is usually difficult to be made flexible to the evolving data distributions. Machine learning, on the other hand, has been applied primarily to build *predictive* models, such as classifications or regressions (Michalski and Anderson, 1984; Bishop, 2007). While the structure of a machine learning model (like a neural network) must be designed, the actual model parameters are learned automatically via gradient descent algorithms. This gives machine learning models the flexibility to adapt to the evolving data distributions. Nevertheless, it is difficult to enforce constraints on the output of machine learning models. Many real-world applications are beyond the reach of constraint reasoning or machine learning alone.

In this paper, we focus on structured prediction problems, which is a class of learning problems requiring both constraint reasoning and machine learning. It expands the output space of classification problems into high-dimensional structured space. Structured prediction has diverse application domains, ranging from natural language processing (Socher et al., 2013), social network analysis (Xiang and Neville, 2013), and ecological modeling (Tang et al., 2018; Chen et al., 2018). The applications we consider in this paper all require tight integration of constraint reasoning and machine learning. Our first application *vehicle dispatching service planning* is to recommend a route that satisfies the daily service needs as well as meeting the drivers' preferences. Historical data may reveal that the drivers do not follow common stylized objectives such as minimizing distance or time. Therefore standard constraint reasoning tools, *e.g.*, solvers for the traveling salesman problem, cannot be applied. While we need machine learning to capture the drivers' objective functions, pure machine learning-based approaches are insufficient because they often generate routes that violate delivery requests. Our second and third applications are *program synthesis from natural language*, which clearly requires machine learning to generate structured programs. Nevertheless, a pure learning approach cannot enforce the syntactic and semantic rules of those programs.

We propose **Co**nstraint **Re**asoning embedded **S**tructured **P**rediction (Core-Sp), a scalable constraint reasoning and machine learning integrated approach for learning over the structured domains. The main idea is to augment structured predictive models with a constraint reasoning module that represents physical and operational requirements. Specifically, we propose to embed decision diagrams (Akers, 1978; Bryant, 1986), a popular constraint reasoning tool, as a fully-differentiable module into deep neural networks. A decision diagram is a compact graphical representation of the constraints. It encodes each solution (an assignment of values to variables satisfying the constraints) as a path from the root to the terminal in the diagram. Core-Sp regards the neural network predictions as the simulation of descending along a path in the decision diagram. To ensure constraint satisfaction, Core-Sp filters out variable assignments from the neural network predictions that violate constraints. With the integration of Core-Sp, we provide structured prediction models with constraint satisfaction assurances. Moreover, structured prediction models with the Core-Sp layer enjoy a smaller prediction space than traditional structured prediction approaches, allowing our approach to learn faster in training and generalize better in testing. See Figure 1(a) for our proposed Core-Sp model which integrates constraint reasoning and machine learning for the three application domains. The high-level idea of Core-Sp is illustrated in Figure 1(b).

Previous approaches have considered regularizing machine learning with constraint reasoning in various application domains. Within the broader context of learning constrained models, the work of Coletta et al. (2003); Lallouet et al. (2010); Beldiceanu and Simonis (2012); Bessiere et al. (2017); Addi et al. (2018) have studied automating the constraint acquisition process from historic data or (user-)generated queries. These approaches use partial or complete examples to identify the constraints that can be added to the model. The type of constraints that can be learned depends on the formulation. Several works (Punyakanok et al., 2004; Roth and Yih, 2005; Amos and Kolter, 2017; Ferber et al., 2020) enable learning in a constrained domain via encoding mathematical programming, such as quadratic programming or mixed integer linear programming, as a neural network layer. Deutsch et al. (2019) propose to formulate the output space as an automata. They use the constraints to prune all the invalid transitions in the automata to ensure the validity of the structured outputs. In addition, constraints imposed by a knowledge graph have been embedded into the neural network as differentiable layers (Peters et al., 2019; Wu et al., 2017). Zeng et al. (2021) and Heim (2019) enforce physical constraints or expert inputs as soft constraints. We will illustrate the difference between our approach and these methods in Section 3.2. A different approach is to embed a machine learning model into optimization, *e.g.*, by extending a constraint system with appropriate global constraints. For example, Lallouet and Legtchenko (2007) integrate neural networks and decision trees with constraint programming, while Lombardi et al. (2017) and Lombardi and Gualandi (2016) introduce a "Neuron" global constraint that represents a pre-trained neural network. Another series of approaches based on grammar variational autoencoders (Kusner et al., 2017; Dai et al., 2018; Jin et al., 2018) use neural networks to encode and decode from the parse-tree of a context-free grammar to generate discrete structures. Such approaches are used to generate chemical molecule expressions, which represent a structured domain. Machine learning approaches have also been used to solve constraint reasoning and optimization problems. This includes the works of Galassi et al. (2018) and Vinyals et al. (2015), which use neural networks to extend partial solutions to complete ones. Bello et al. (2017) handle the traveling salesman problem by framing it as reinforcement learning. Selsam et al. (2019) proposes to learn an SAT solver from single-bit supervision. Approaches based on neural Turing machines (Graves et al., 2016) employ neural networks with external memory for discrete structure generation. More recently, Khalil et al. (2017) tackle the combinatorial optimization problems in graphs, by employing neural networks to learn the heuristics in the backtrack-free search. There is also a recent trend to synthesize programs using machine learning (Guu et al., 2017; Shi et al., 2019).

In experimental analysis, we demonstrate the effectiveness of Core-Sp on the following three applications: (1) *Vehicle Dispatching Service Planning*: a route planning problem that recommends routes to drivers to meet the service needs while satisfying the drivers' preferences. The implicit preferences of drivers are learned from the historical traveling data. The input of this problem is the daily service requests. The output is the permutations of the service locations, representing the sequential order that the locations should be visited by the drivers. This task requires machine learning models to capture drivers' preferences from the traveling data, and constraint reasoning to ensure the satisfaction of service requests. (2) *If-then Program Synthesis*: the task is to automatically synthesize conditional programs from the natural language. Automatic program synthesis tools are useful to streamline the program of a few online services such as IFTTT and Zapier. The if-then program is in the form of: if `trigger function` happens in the `trigger service`, then take the `action function` from the `action service`. The machine learning task, therefore, is to predict the quadruple (`trigger service`, `trigger function`, `action service`, `action function`). This application again requires machine learning to understand the semantics of the natural language, as well as constraint reasoning to satisfy the syntactic rules of the programs. (3) *Text2SQL Generation*: our last application is to automatically generate SQL queries that extract information from a database to answer a question posed in natural language. The neural model is used to understand the user's queries in natural language while the constraint reasoning tool is applied to ensure the model generates grammatically-valid SQL queries.

Our proposed Core-Sp framework demonstrates superior performance against the state-of-the-art approaches in all three applications. First, the structures generated by Core-Sp are better in constraint satisfaction. In vehicle service dispatching, all Core-Sp generated routes are valid, while a conditional generative adversarial network (cGAN) without Core-Sp generates on average less than 1% of valid routes when handling medium-sized delivery requests. We also apply a post-processing step (Deudon et al., 2018) to boost cGAN's performance, but it cannot handle the complexity brought by the large combinatorial space of the routing problem. Its performance quickly defaults to the case without post-processing as the number of delivery locations increases. For if-then program synthesis, the percentage of valid programs produced increased from 88% to 100% with the Core-Sp module incorporated into the state-of-the-art LatentAttention model (Liu et al., 2016). For Text2SQL, the percentage of valid SQL queries increased from 83.7% to 100% with Core-Sp incorporated into the state-of-the-art SQLNova model (Hwang et al., 2019) on a hard testing set. Core-Sp also improves the learning performance of structured prediction models. We show that the routes generated by Core-Sp better fulfill drivers' preferences than cGAN without Core-Sp. In if-then program synthesis, Core-Sp module leads to approximately 2.0% improvement in accuracy compared with the state-of-the-art LatentAttention model and converges to models with higher accuracy in fewer training epochs. In Text2SQL generation, the Core-Sp module improves around 4.2% in execution accuracy and 1.9% in logical accuracy against SQLNova on a challenging test set.

2. Preliminaries

In this section, we first cover the structured prediction and then brief the decision diagrams.

2.1 Structured Prediction

Structured prediction expands the output space of classification problems into a high-dimensional combinatorial space (Bakır et al., 2007). Specifically, given a set of input-output samples $\mathcal{D}^{tr} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$ drawn *i.i.d.* from some unknown distribution over the space $\mathcal{X} \times \mathcal{Y}$, a structured prediction model learns a conditional distribution $p_\theta(y|x)$, for all $(x, y) \in \mathcal{X} \times \mathcal{Y}$ from data $\mathcal{D}^{tr}$, where θ denotes the parameters of the structured prediction model. Note that the output space $\mathcal{Y} = \{0, 1\}^l$ is a high dimensional space of combinatorial structures. The three applications we consider in this paper are all structured prediction problems. In vehicle dispatching service planning, the structured outputs are the delivery routes on a map. In if-then program synthesis, the structured outputs are the programs that complete web-service tasks. In Text2SQL generation, the structured outputs are the SQL queries that follow the SQL grammar.

In the literature, various approaches have been proposed for structured prediction problems. The classifier chain approach (Read et al., 2015) decomposes the joint likelihood into a product of conditionals and reduces the structured prediction problem into a series of binary prediction problems. In this approach, the error tends to propagate along the classifier chain, which limits its effectiveness (Dembczynski et al., 2010). Energy-based modeling, such as conditional random fields (Lafferty et al., 2001; Geman and Geman, 1984) and structured prediction energy networks (Belanger and McCallum, 2016) learn to assign a high likelihood to structures that exist in the training data set while keeping the likelihood low for unseen structures. Constraints can be incorporated into these models as prior terms in the energy function but approximated inference is required to compute the intractable partition function, which often hinders their scalability. Another line of research uses structured support vector machines (Tsochantaridis et al., 2005), which apply hinge loss and row generation approaches for structured prediction; however, these were superseded in performance by later neural-network-based approaches. Recently, generative models, such as conditional generative adversarial networks (Mirza and Osindero, 2014; Goodfellow et al., 2014), flow models (Rezende and Mohamed, 2015), and sequence-to-sequence models (Sutskever et al., 2014) have become increasingly popular for structured prediction. These models use highly flexible neural networks to increase model capability. The over-parameterized networks with gradient descent-based optimization can learn better representation for the structures than the classic shallow models. However, it is not straightforward to enforce constraints into the neural network-based models.

Constraints in Structured Prediction. Often the structured output space $\mathcal{Y}$ is subject to additional constraints $\mathcal{C}$. The conditional probability that y takes values that violate the (physical) constraints $\mathcal{C}$ given the input x is zero. Such information is known prior to the training of the machine learning model. Formally, we have:

$$p(y|x)\begin{cases} > 0 & \text{if } y \text{ satisfies } \mathcal{C}, \\ = 0, & \text{if } y \text{ violates } \mathcal{C}. \end{cases} \tag{1}$$

Take the first task discussed in this paper as an example. A valid delivery route should cover all the requested locations and should only visit each location once. Thus, the machine learning model should assign zero probability to those invalid routes. Notice that the constraints are often intricate and the inference problem of finding a valid structure satisfying constraints cannot be decomposed into independent small problems. After learning, the *inference* problem is to predict the structured output y given the input x. Such inference problems can be solved by either Maximum A Posteriori (MAP) inference, *e.g.*, computing $max_y\ p(y|x)$ or marginal inference, *e.g.*, computing $\mathbb{E}_y[p(y|x)]$. Learning structured prediction models involves solving the inference problems within the learning loop, hence having an even higher complexity.

Combinatorial constraints render both the inference and the learning problems highly intractable. Indeed, much effort has been made to improve the efficiency of both the inference and learning problems (Pan and Srikumar, 2018; Bello et al., 2020). For example, Niculae et al. (2018) propose the sparseMAP function which solves the inference problem by returning a few sparse structures that attain high likelihoods. This inference method sits between the MAP and marginal inference. In their problem setup, sparseMAP can be solved via quadratic programming. However, combinatorial constraints considered in this paper make the inference problem non-convex, even for a fixed structured prediction model, let alone the more challenging learning problem. Overall, constrained structured prediction presents two main challenges. The first is the *sample complexity*, since massive data is needed to learn an accurate model in an exponentially large space. The second is the *computational complexity*, since it is combinatorially intractable (unless P=NP) to generate structured outputs subject to complicated constraints.

Sequence-to-sequence Structured Prediction. Our proposed Core-Sp method is designed to extend *sequence-to-sequence* models, which are recently proposed popular structured prediction models (Sutskever et al., 2014). The sequence-to-sequence model uses the re-parameterization trick to model the conditional probability $p_\theta(y|x)$, where $x \in \mathcal{X}$ denotes the input variables and $y \in \mathcal{Y}$ is the structured output. Here θ denotes the parameters of the neural model. Instead of modeling the probability $p_\theta(y|x)$ directly, the model introduces an additional random variable z and models it as a deterministic transformation from random variable z and evidence x to the output y. In other words, the conditional probability $p_\theta(y|x)$ is an integral over random variable z in the following way:

$$\begin{aligned} p_\theta(y|x) &= \int p_\theta(y|x,z)p(z)\,dz, \\ p_\theta(y|x,z) &= \mathbb{1}\{y = f_\theta(x,z)\}, \end{aligned} \tag{2}$$

where we assume z is from a known prior probability distribution $p(z)$. As a result, we only need to model $p_\theta(y|x, z)$ for the overall model $p_\theta(y|x)$. We further assume that $p_\theta(y|x, z)$ is given in the form of a deterministic function. We let $f_\theta(x, z) \in \mathcal{Y}$ be a deterministic mapping from inputs (x, z) to an output in the structured space $\mathcal{Y}$. The indicator function $\mathbb{1}\{\cdot\}$ evaluates to 1 if and only if $y = f_\theta(x, z)$. This formulation is closely related to the generative adversarial network and gives us high flexibility to model multi-modal distributions. Take the vehicle dispatching service planning as an example. The input x is the daily service requests and y is the suggested dispatching route. There can be several routes that meet the service demands and satisfy the driver's underlying preference function. In this case, the conditional probability $p_\theta(y|x)$ may have multiple modes, one for each good route. This formulation allows us to represent the multi-modal distribution effectively. The variable z decides which route to pick. The function $f_\theta(x, z)$ returns one route that meets the demand of input x and is randomly selected by z. If $p_\theta(y|x)$ has k modes, the space of z will be split into k regions where variable z in every region will be mapped to one mode in $p_\theta(y|x)$.

We use a sequence-to-sequence neural network to model the function $f_\theta(x, z)$. Assume the input variables x, z, and the output y are all represented in sequential forms $x = (x_1, x_2, \dots, x_T)$, $z = (z_1, z_2, \dots, z_T)$ and $y = (y_1, y_2, \dots, y_T)$. The sequence-to-sequence model is made of an encoder and a decoder. The sequential encoder receives x and outputs a representation vector for input x. The sequential decoder receives the output of the encoder as well as z and outputs y in T steps, where T refers to the maximum length for variable y. In the k-th step ($1 \le k \le T$), the decoder network takes z_k, and the hidden vector h_{k-1} from the previous step as inputs, and outputs a score vector $o_k = (o_{k1}, o_{k2}, \dots, o_{kD_k})$ of length $D_k = |D(y_k)|$. Here, o_k corresponds to the un-normalized likelihoods of each value that variable y_k can take. The softmax function is then applied to get the normalized probability:

$$p_{kj} = p\left(y_k = v_j | x, h_{k-1}\right) = \frac{\exp(o_{kj})}{\sum_{j'=1}^{D_k} \exp(o_{kj'})}, \qquad \text{for } j = 1, 2, \dots, D_k.$$

p_{kj} is the probability that variable y_k takes the j-th value v_j. Assume the prior distribution $p(z_k)$ is the uniform distribution in $(0, 1)$, denoted by $\mathcal{U}(0, 1)$. Variable z_k is sampled from $\mathcal{U}(0, 1)$ and is used to determine the value for y_k according to the probability distribution vector $p_k = (p_{k1}, p_{k2}, \dots, p_{kD_k})$. Let $P_{k1}, P_{k2}, \dots, P_{k(D_k+1)}$ be the cumulative probabilities:

$$P_{kj} = \begin{cases} 0 & \text{for } j = 1, \\ \sum_{j'=1}^{j-1} p_{kj'} & \text{for } j = 2, 3, \dots, D_k, \\ 1 & \text{for } j = D_k + 1. \end{cases}$$

y_k is set to the value v_j if and only if $z_k \in \left[P_{kj}, P_{k(j+1)}\right)$. Notice that because z_k is sampled from $\mathcal{U}(0, 1)$, the probability that y_k takes the j-th value v_j is exactly p_{kj}. Aside from producing the value for y_k in the k-th step, the sequence-to-sequence neural net also produces the hidden-state vector h_k at the k-th step, which is used by the neural net again in the subsequent $(k + 1)$-th step. The overall architecture of the sequence-to-sequence model can be seen in Figure 4.

The training process of the sequence-to-sequence model is to minimize a pre-defined loss function, or an additional discriminator neural net, which penalizes the differences of

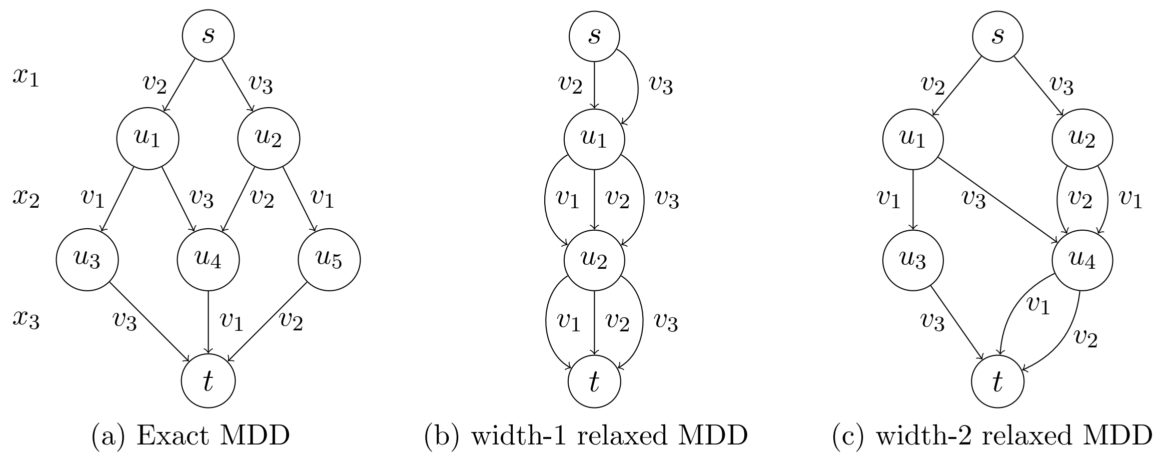


Figure 2: Illustration of Multi-valued Decision Diagrams (MDDs) for decision variables x_1, x_2, x_3. **(a)** An exact MDD with all variable assignments satisfying two constraints: `all-diff`(x_1, x_2, x_3) and $x_1 \neq v_1$. **(b)** A width-1 relaxed MDD for the exact MDD in (a). **(c)** A width-2 relaxed MDD, which is formed by combining nodes u_4 and u_5 of the MDD in (a).

the predicted structure $f_\theta(x, z)$ and the observed structure y. Here $f_\theta(x, z)$ is a predicted sequence obtained from the above process. Given a training data set $\mathcal{D}^{tr} = \{(x^{(i)}, y^{(i)})\}_{i=1}^N$, the learning objective is to minimize the loss function:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_{z^{(i)}} \left[\ell \left(f_\theta \left(x^{(i)}, z^{(i)} \right), y^{(i)} \right) \right]. \tag{3}$$

Here $\ell(\cdot, \cdot)$ can be a predefined loss function that measures the mismatch between the predicted and observed structures. Function $\ell(\cdot, \cdot)$ can also be represented as a discriminator network, which leads to the training of a generative adversarial network. The parameters θ are updated via gradient descent, *i.e.*, $\theta^{t+1} = \theta^t - \eta \nabla \mathcal{L}(\theta)$, where η denotes the learning rate.

2.2 Decision Diagrams

Decision diagrams were originally introduced to compactly represent Boolean functions in a graphical form (Akers, 1978; Bryant, 1986). Since then, they have been widely used in the context of verification and configuration problems (Wegener, 2000). More recently, they have been used successfully as an optimization tool, by representing the set of solutions to combinatorial optimization problems (Bergman et al., 2016b; van Hoeve, 2022).

Decision diagrams are defined with respect to a sequence of decision variables $x_1, \ldots, x_n$. Variable x_i has a domain of possible values $D(x_i)$, for $i = 1, 2, \ldots, n$. A decision diagram is a directed acyclic graph, with $n + 1$ layers of nodes. Layer 1 contains a single node s, called the root. Layer $n + 1$ also contains a single node t, called the terminal. An arc from a node in layer i to a node in layer $i + 1$ represents a possible assignment of variable x_i to a value in its domain and is therefore associated with a label in $D(x_i)$. For an arc $e(v, u)$,

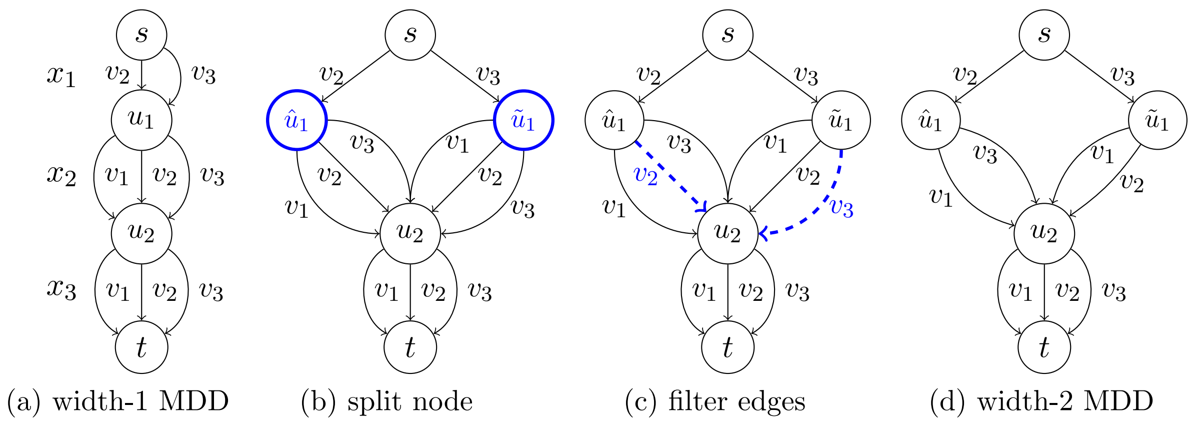


Figure 3: Node splitting and arc filtering for MDDs for variables x_1, x_2, x_3. **(a)** A width-1 relaxed MDD as in Figure 2(b). **(b)** Split node u_1 into $\hat{u}_1$ and $\tilde{u}_1$. **(c)** Filter arcs $e(\hat{u}_1, u_2) = v_2, e(\tilde{u}_1, u_2) = v_3$ that violate the constraint $\texttt{all-diff}(x_1, x_2, x_3)$. The arcs in dashed lines are removed. **(d)** A width-2 relaxed MDD after one iteration of node splitting and arc filtering.

we use $\texttt{val}(v, u) \in D(x_i)$ to represent the assigned label for variable x_i. For a node v in layer i, we use $\texttt{val}(v) \subseteq D(x_i)$ to represent the union of the values of each arc starting from node v, *i.e.*, $\texttt{val}(v) = \cup_{e(v,u)}\{\texttt{val}(v, u)\}$. In other words, $\texttt{val}(v)$ represents the possible value assignments for the decision variable x_i at node v. Each path from the root s to the terminal t represents a solution, *i.e.*, a complete variable assignment. In this paper, we consider variables with domains of categorical values, which result in so-called multi-valued decision diagrams (MDDs) (Wegener, 2000). See Figure 2 for an example.

Exact Decision Diagrams. Given a set of constraints $\mathcal{C}$, the MDD $\mathcal{M}$ is said to be *exact* with respect to $\mathcal{C}$ if and only if every path that leads from the root node s to the terminal node t in $\mathcal{M}$ is a variable assignment satisfying all constraints in $\mathcal{C}$. Conversely, every valid variable assignment can be found as a path from s to t in $\mathcal{M}$.

Relaxed Decision Diagrams. Since exact decision diagrams can grow exponentially large, *relaxed* decision diagrams were introduced to limit their size (Andersen et al., 2007). The set of paths in a relaxed decision diagram forms a superset of the paths in the associated exact decision diagram. Relaxed MDDs are often defined with respect to the maximum layer width, which is the number of nodes in its largest layer.

Variable Ordering. In general, the size of an exact decision diagram is known to strongly depend on the variable ordering (Friedman and Supowit, 1990). In our applications, however, we consider *sequential* decision processes which follow a natural prescribed ordering. Our approach can also be applied to more general decision problems, in which case the variable ordering needs to be considered when compiling the MDD.

Example 1 *Figure 2 demonstrates several MDDs. Let x_1, x_2, x_3 be a sequence of decision variables with domain $D(x_1) = D(x_2) = D(x_3) = \{v_1, v_2, v_3\}$. The constraint $\texttt{all-diff}(x_1, x_2, x_3)$ restricts the values of x_1, x_2 and x_3 to be all different, i.e., they form a permutation. The other constraint is $x_1 \neq v_1$. (1) Exact MDD. The set of feasible permuta-*

tions is $\{(v_2, v_1, v_3), (v_2, v_3, v_1), (v_3, v_2, v_1), (v_3, v_1, v_2)\}$. *Figure 2(a) depicts the exact MDD that encodes all permutations satisfying the two constraints. (2) Relaxed MDD. Figure 2(b) is a width-1 relaxed MDD and Figure 2(c) is a width-2 relaxed MDD. The set of paths in the relaxed MDD forms a superset of all feasible permutations. As an illustration, Figure 2(c) contains two infeasible solutions* $\{(v_3, v_1, v_1), (v_2, v_2, v_2)\}$. *(3) Variable ordering. All the MDDs in Figure 2 have the same variable ordering of* $\pi = (1, 2, 3)$, *meaning that the MDD first expands on variable* x_1, *then* x_2, *finally* x_3.

Decision Diagram Compilation. Decision diagrams can be compiled via a repeated process of *node splitting* and *arc filtering* from a width-1 relaxed MDD (Andersen et al., 2007; Bergman et al., 2016a). Arc filtering removes arcs that lead to infeasible solutions, while node splitting increases the size of the decision diagram by splitting one node into two or more nodes. In practice, one can reach an exact MDD by repeatedly going through the splitting and filtering processes from a width-1 MDD. We refer to Ciré and van Hoeve (2013) for the detailed process of MDD compilation for sequential decision problems.

Example 2 *Figure 3 demonstrates one possible process of applying the node splitting and arc filtering steps. We re-use the example in Figure 2(b) as the initial MDD in Figure 3(a), which depicts a width-1 relaxed MDD before compilation. The constraint to be applied is* ***`all-diff`***(x_1, x_2, x_3), *i.e., the assignments of variables* x_1, x_2, x_3 *should be pairwise different. The node* u_1 *in Figure 3(a) is split into two nodes* $\hat{u}_1, \tilde{u}_1$ *in Figure 3(b). The incoming arc* $e(s, u_1)$ *with labe* v_2 *is assigned to node* $\hat{u}_1$ *and the other incoming arc* $e(s, u_1)$ *with label* v_3 *is assigned to node* $\tilde{u}_1$. *The outgoing arcs of node* u_1 *are copied for the two nodes. In Figure 3(c), the arc filtering process checks if certain variable assignments violate constraints for the two nodes. Arc* $e(\hat{u}_1, u_2) = v_2$ *is not compatible with the previous arc* $e(s, \hat{u}_1)$ *with label* v_2 *because it violates* ***`all-diff`***(x_1, x_2, x_3). *Thus it is removed. For the same reason, arc* $e(\tilde{u}_1, u_2) = v_3$ *is also removed. (d) We get a width-2 relaxed MDD after splitting node* u_1 *and filtering the arcs.*

3. Constraint Reasoning Embedded Structured Prediction

Core-Sp is motivated by the lack of constraint satisfaction in sequence-to-sequence structured prediction models. The key idea of Core-Sp is the correspondence between the predicted outcomes of a sequence-to-sequence model and a path in a multi-valued decision diagram (MDD). Figure 4 provides an example. In this example, the sequence-to-sequence model outputs a sequence of variable assignments $y_1 = v_2$, $y_2 = v_3$, $y_3 = v_1$ in Figure 4(a), which exactly corresponds to the highlighted blue path in the MDD in Figure 4(b). However, the sequence-to-sequence model is also likely to output a variable assignment with no correspondence to the MDD. For example, if the neural model in Figure 4(a) outputs $y_1 = v_2$, $y_2 = v_3$, $y_3 = v_2$, there is no corresponding path in the MDD in Figure 4(b). This illustrates the case where the output of the sequence-to-sequence model violates the `all-diff` constraint. Indeed, neural network-based models for structured prediction problems are not guaranteed to satisfy constraints as defined in Equation (1), which forms a key limitation of state-of-the-art structured prediction models.

Core-Sp ensures constraint satisfaction of the neural network prediction by limiting the values that each variable can take following the flow of the MDD. Suppose we set $y_1 = v_2$

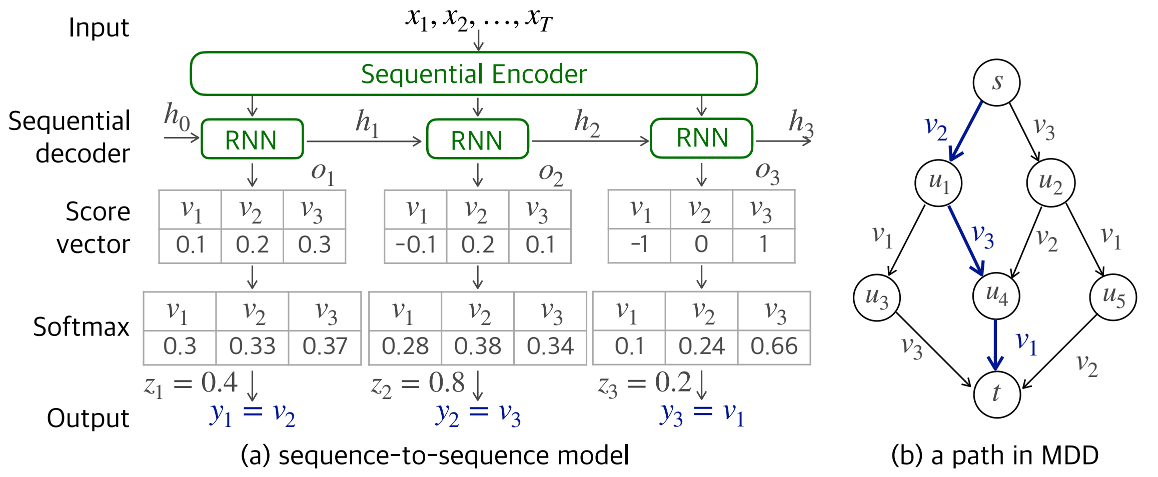


Figure 4: Illustration of (a) a sequence-to-sequence model which generates an output corresponding to (b) a path in the multi-valued decision diagram. **(a)** A sequence-to-sequence model receives input x and random variables z, and outputs $y_1 = v_2$, $y_2 = v_3$ and $y_3 = v_1$ in three steps. **(b)** The assignment $(y_1, y_2, y_3) = (v_2, v_3, v_1)$ corresponds to path $s \xrightarrow{v_2} u_1 \xrightarrow{v_3} u_4 \xrightarrow{v_1} t$ in the multi-valued decision diagram.

and $y_2 = v_3$ in Figure 4(b) and arrive at node u_4, the only valid option for y_3 is to set $y_3 = v_1$. The other options $y_3 = v_2$ or $y_3 = v_3$ lead to constraint violations. Hence Core-Sp masks out the choices of $y_3 = v_2$ and $y_3 = v_3$ for the sequence-to-sequence model. In this way, Core-Sp addresses a key limitation of structured prediction models. We next provide the details of Core-Sp.

3.1 Embed Constraint Reasoning in Structured Prediction

Our proposed Core-Sp framework creates an additional layer to a sequence-to-sequence model to enforce constraint satisfaction for structured problems. It can be integrated into various structured prediction neural networks for different tasks. In this paper, we demonstrate the Core-Sp layer on the sequence-to-sequence structured prediction network. Core-Sp works by masking out the output that violates constraints, thereby providing correctness guarantees. Following the discussions of Section 2.1, the sequence-to-sequence structured prediction neural network receives input $x = (x_1, x_2, \ldots, x_T)$ and $z = (z_1, z_2, \ldots, z_T)$ in sequential format and outputs $y = (y_1, y_2, \ldots, y_T)$. In the k-th step, the score vector $o_k = (o_{k1}, o_{k2}, \ldots, o_{kD_k})$ is produced by the sequence-to-sequence network, where o_{kj} represents the un-normalized likelihood that y_k takes the value v_j. Vector $p_k = (p_{k1}, p_{k2}, \ldots, p_{kD_k})$ is the result after normalizing o_k, where p_{kj} is the probability for variable y_k to take the value v_j. Without the addition of the Core-Sp layer, the probability p_{kj} may be assigned a positive value for certain variable assignments $y_k = v_j$ that lead to a constraint violation.

The Core-Sp module enforces constraints by masking out certain entries p_{kj} of the vector p_k whose associated assignment $y_k = v_j$ is not allowed by the MDD. It does this by tracking a 'pivot node' in the MDD. Initially, the pivot node starts at the source node, and it descends along a path determined by the output of Core-Sp in a sequential way. In the example in Figure 4, the pivot node starts at node s, descends along nodes u_1, u_4, and arrives at t, following the output of the sequence-to-sequence model. In each step, Core-Sp maintains a mask vector $c_k = (c_{k1}, c_{k2}, \dots, c_{kD_k})$ based on the current pivot node. Vector c_k is used to mask out entries in p_k that will lead to constraint violation. If there is no path labeled with v_j leaving the current pivot node, c_{kj} is set to 0. Otherwise, c_{kj} is set to 1. Suppose the pivot node is at u_1 in the example shown in Figure 4, c_{22} is set to 0, and c_{21}, c_{23} are set to 1 because the two outgoing edges from u_1 are labeled with v_1 and v_3. The next step of Core-Sp is the element-wise multiplication of p_k and c_k, resulting in p'_k. Denote $\odot$ as the element-wise vector-vector product, the masking step is computed as $p'_k = p_k \odot c_k$. Those entries that lead to constraint violation in p'_k are zeroed out. To make sure that the probabilities sum up to 1, p'_k further goes through a re-normalization step. The re-normalized probability vector is computed as: $\tilde{p}_{kj} = \frac{p'_{kj}}{\sum_{j'} p'_{kj'}}$. Finally, z_k is sampled uniformly at random from $\mathcal{U}(0,1)$ and the output y_k is decided based on the cumulative probabilities $P_{k1}, P_{k2}, \dots, P_{k(D_k+1)}$ computed from $\tilde{p}_k$: $P_{k1} = 0$, and $P_{kj} = \sum_{j'=1}^{j-1} \tilde{p}_{kj'}$, for $j = 2, 3, \dots, D_k$ and $P_{k(D_k+1)} = 1$. y_k is set to the value of v_j if and only if $z_k \in \left[P_{kj}, P_{k(j+1)}\right)$. Denote assignment indicator vector $q_k = (q_{k1}, q_{k2}, \dots, q_{kD_k})$, where q_{kj} is an indicator variable for $y_k = v_j$. This implies q_{kj} is 1 if and only if $y_k = v_j$, otherwise $q_{kj} = 0$. After setting the value of y_k, the pivot node descends to a new node along the corresponding arc in the MDD. To conclude, the computational pipeline at the k-th step is reflected in the following equations:

$$p_{kj} = \frac{\exp(o_{kj})}{\sum_{j'=1}^{D_k} \exp(o_{kj'})}, \tag{4}$$

$$p'_k = p_k \odot c_k, \tag{5}$$

$$\tilde{p}_{kj} = \frac{p'_{kj}}{\sum_{j'=1}^{D_k} p'_{kj'}}, \tag{6}$$

$$P_{kj} = \begin{cases} 0 & \text{for } j = 1, \\ \sum_{j'=1}^{j-1} \tilde{p}_{kj'} & \text{for } j = 2, 3, \dots, D_k, \\ 1 & \text{for } j = D_k + 1. \end{cases} \tag{7}$$

$$q_{kj} = \begin{cases} 1 & \text{if } z_k \in \left[P_{kj}, P_{k(j+1)}\right), \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

$$y_k = v_j, \qquad \text{if } q_{kj} = 1, \text{ for } 1 \le j \le D_k. \tag{9}$$

Here $\odot$ denotes the element-wise product between two vectors and $z_k \sim \mathcal{U}(0,1)$. We illustrate how Core-Sp works using the following Example 3.

Example 3 *We illustrate the procedure of* Core-Sp *using the example in Figure 5. Initially, the pivot node for tracking the MDD is the root node s. The first step is to set the value for the first variable y_1. Here, the neural network outputs an un-normalized likelihood vector*

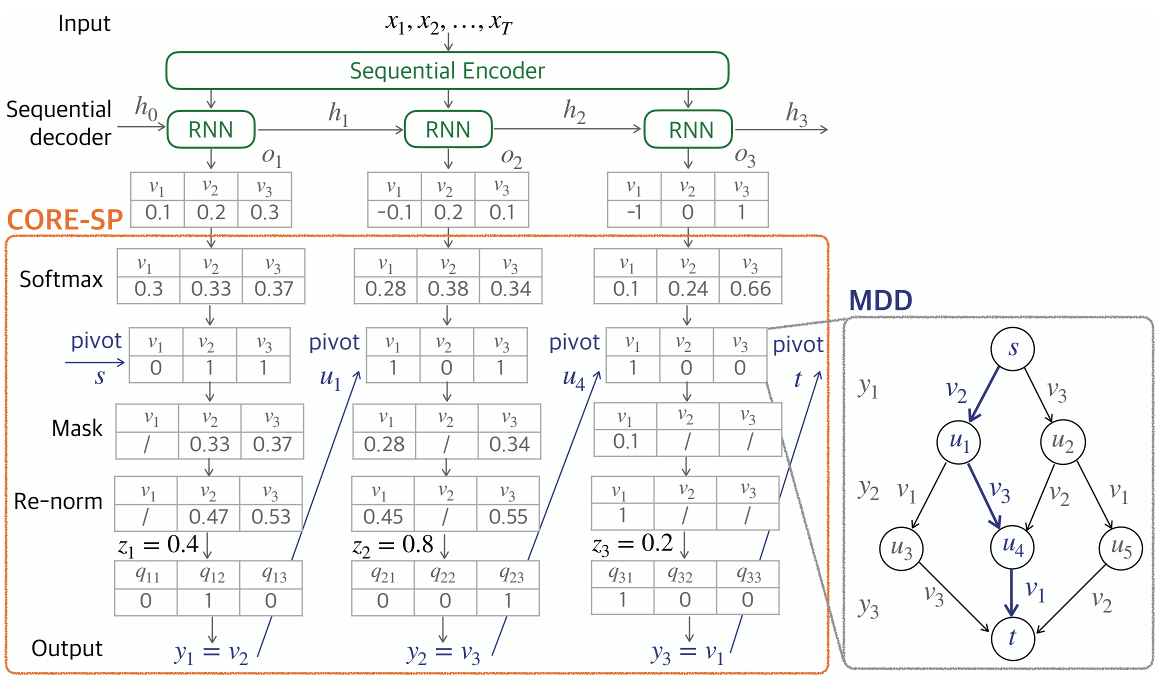

Input
$x_1, x_2, \dots, x_T$
Sequential Encoder
Sequential decoder
h_0 RNN h_1 RNN h_2 RNN h_3
o_1 o_2 o_3
v_1 v_2 v_3: 0.1 0.2 0.3 | -0.1 0.2 0.1 | -1 0 1
CORE-SP
Softmax: 0.3 0.33 0.37 | 0.28 0.38 0.34 | 0.1 0.24 0.66
pivot s: 0 1 1 | pivot u_1: 1 0 1 | pivot u_4: 1 0 0 | pivot t
Mask: / 0.33 0.37 | 0.28 / 0.34 | 0.1 / /
Re-norm: / 0.47 0.53 | 0.45 / 0.55 | 1 / /
$z_1 = 0.4$ $z_2 = 0.8$ $z_3 = 0.2$
q_{11} q_{12} q_{13}: 0 1 0 | q_{21} q_{22} q_{23}: 0 0 1 | q_{31} q_{32} q_{33}: 1 0 0
Output: $y_1 = v_2$ $y_2 = v_3$ $y_3 = v_1$
MDD
y_1 y_2 y_3
s u_1 u_2 u_3 u_4 u_5 t
v_1 v_2 v_3

Figure 5: Architecture of embedding CORE-SP into a sequence-to-sequence model for the decision variables y_1, y_2, y_3, where the highlighted CORE-SP module encodes the exact MDD in Figure 2(a). CORE-SP descends layer-by-layer in the MDD. Initially, the pivot node of CORE-SP is at root s. The node s limits the value of y_1 to be $y_1 \in \texttt{val}(s) = \{v_2, v_3\}$. If the model picks $y_1 = v_2$, then the pivot node moves to node u_1 following the arc $e(s, u_1) = v_2$. Next, the node u_1 limits the value of y_2 to be $y_2 \in \texttt{val}(u_1) = \{v_1, v_3\}$. If the neural model picks $y_2 = v_3$, then the pivot node shifts to node u_4 following the arc $e(u_1, u_4) = v_3$. Finally, the pivot node descends to u_4 following the single outgoing arc: $e(u_4, t) = v_1$. Hence the assignment for variable y_3 becomes $y_3 = v_1$.

$o_1 = (0.1, 0.2, 0.3)$. *The next softmax layer receives* o_1 *and outputs the normalized probability vector* $p_1 = \left(\frac{\exp(0.1)}{\exp(0.1)+\exp(0.2)+\exp(0.3)}, \frac{\exp(0.2)}{\exp(0.1)+\exp(0.2)+\exp(0.3)}, \frac{\exp(0.3)}{\exp(0.1)+\exp(0.2)+\exp(0.3)}\right) \approx (0.30, 0.33, 0.37)$. *From the MDD on the right-hand side,* y_1 *has only two valid assignments,* v_2 *or* v_3. *Therefore,* CORE-SP *produces a mask vector* $c_1 = (0, 1, 1)$, *which forbids* y_1 *taking the value* v_1. *As in Equation (5), multiplying* p_1 *with* c_1 *elementwisely gives us an unnormalized probability vector* $p'_1 = (0, 0.33, 0.37)$. *After the re-normalization operation in Equation (6), we obtain* $\tilde{p}_1 = (0, \frac{0.33}{0.33+0.37}, \frac{0.37}{0.33+0.37}) \approx (0, 0.47, 0.53)$. *According to Equation (7), the cumulative probability vector would be:* $P_1 = (0, 0, 0.47, 1)$. *We then uniformly sample* z_1 *at random between 0 and 1. In this example,* $z_1 = 0.4 \in [0, 0.47)$, *hence we get vector* $q_1 = (0, 1, 0)$ *and set* $y_1 = v_2$, *that corresponds to Equations (8) and (9). After setting* y_1*'s value,* CORE-SP *sets the pivot node to* u_1 *following the arc* $e(s, u_1) = v_2$. *It continues the same process of setting values for* y_2 *and* y_3. *This example sets* y_2 *to* v_3 *and* y_3 *to* v_1, *which corresponds to the blue path in the decision diagram on the right-hand side.*

Proposition 1 *Let $\mathcal{M}$ be a non-empty exact MDD that is compiled from the constraint set $\mathcal{C}$. The sequence-to-sequence model with the addition of* Core-Sp *is guaranteed to generate structured outputs satisfying all constraints in $\mathcal{C}$.*

Proof Because $\mathcal{M}$ is exact, it represents all solutions to $\mathcal{C}$ with respect to the domains of the decision variables. At a specific pivot node u in layer k, Core-Sp only masks values for y_k that do not belong to $\texttt{val}(u)$. For all remaining values v, an edge $e(u, u')$ with label v exists. Moreover, at least one path from u' to the terminal node t must exist. Hence, unless $\mathcal{M}$ is empty, the process generates a complete variable assignment satisfying $\mathcal{C}$. ■

Implementation. Core-Sp allows for efficient back-propagation of the gradient of the neural network's parameters. In model training, all computations are differentiable during the gradient backward pass except for the setting of the q_{kj} values. We set q_{kj} to 1 if and only if $z_k \in [P_{kj}, P_{k(j+1)})$. In other words, the value for q_{kj} is determined by $q_{kj} = \mathbb{1}\{z_k \geq P_{kj}\}\mathbb{1}\{z_k < P_{k(j+1)}\}$. When computing $\frac{\partial q_{kj}}{\partial P_{kj}}$, we use the sigmoid function with a huge constant to replace the indicator function $\mathbb{1}\{\cdot\}$. This operation allows for gradient propagation and improves the numeric stability of gradient computation, and avoids producing NaN or Infinity gradients. For the cases where the loss function can be directly defined on $\tilde{p}_k = (\tilde{p}_{k1}, \tilde{p}_{k2}, \ldots, \tilde{p}_{kD_k})$, such as the cross-entropy loss, we do not use z variables to sample variables y_k during training. The z variables are used during testing, as will be detailed for the applications in Section 5. The MDD inside Core-Sp is implemented with two key-value dictionaries. One dictionary memorizes all the mask vectors. It uses the nodes in the MDD as keys and returns the corresponding mask vectors. The other dictionary saves the connectivity of the MDD, it uses the current node in the MDD as the key and returns all of its following nodes in the next layer. This dictionary allows for the pivot node to descend along the path and is also used for the node split and arc filtering procedure discussed in Section 4.

3.2 Connection to Existing Works

There are several existing works that also use reasoning tools to enforce constraints in neural network-based models. OptNet (Amos and Kolter, 2017) and MIPaaL (Ferber et al., 2020) propose to encoding quadratic programming (QP) or mixed integer programming (MIP) to enforce constraints, these methods backpropagate the gradients through their optimality conditions. Both approaches require solving a linear programming problem (or MIP problem) in the forward pass. In contrast, our approach pre-computes the feasible set and therefore can be integrated "as is" into the neural net. No LP or MIP solver is needed.

Another line of work imposes sparsity on the structured output. The sparseMAP method (Niculae et al., 2018) models the probability distribution using a combination of a few sparse structured outputs. Their sparsity assumption implicitly enforces constraints by assigning invalid solutions zero probabilities. Because their overall formulation needs to be convex, the types of combinatorial constraints they can handle are limited. The authors generalize their approaches to handle more general logic constraints in their follow-up work LP-sparseMAP (Niculae and Martins, 2020). Their approach is to decompose the problem in the factor graph, and uses the alternating direction method of multipliers (ADMM) to enforce the consistent value assignments towards variables. This approach indeed provides a good way to handle constraints in structured prediction. However, ADMM only ascends towards the maximum of the dual problem, although the primal-dual gap can be large for non-convex problems. Our approach provides an alternative way to handle constraints beyond problem decomposition and harnessing the primal-dual gap.

Deutsch et al. (2019) propose a strategy to formalize the constraints as automata. During inference, the outputs are generated by walking step-by-step in the automata. Compared to this work, the MDD we use is similar to the automata since both of them only use valid paths as valid solutions. However, we enforce Core-Sp during both learning and inference stages while their space-optimized automata can only be applied in inference. We will show in Section 6 that constraint satisfaction during learning actually leads to improvement in learning performance because of the reduced modeling space. In addition, in Section 4 we propose a relaxed search algorithm for MDD structures to automatically find the sweet spot that balances model complexity and learning performance.

4. Searching for the Optimal CORE-SP Structure

The exact MDDs for real-world problems could be arbitrarily large, so the exact MDD may consume too much memory overhead. Because of the large space complexity of exact MDDs, it is not practical to deploy the Core-Sp with the exact MDD on several real-world problems. Also, a large MDD implicitly implies a complex output space, which requires more data to learn an accurate model. In problems where exact MDDs are not practical, relaxed MDDs can be used to reduce the memory requirement. In this section, we explore the trade-off between space complexity and learning performance by exploring the usage of relaxed MDDs.

To find the optimal MDD structure which balances memory consumption and learning performance, we propose an iterative search procedure, presented in Algorithm 1. We tune the width parameter to find a relaxed Core-Sp that achieves the optimal performance. The algorithm starts increasing the width from 1 to the given hyper-parameter maximum layer with $\omega_{\max}$, iteratively learning Core-Sp model with new MDD model $\mathcal{M}$ on the training set, validating their learning performance on a separated validation data set until finding a good MDD structure. The inputs to Algorithm 1 are training and validation data sets $(\mathcal{D}^{tr}, \mathcal{D}^{val})$, parameters of the sequence-to-sequence neural network θ, a set of constraints $\mathcal{C}$ and the maximum width $\omega_{\max}$ of MDD. In the beginning, the `initMDD` function initializes a width-1 MDD. At every iteration, the `train` function trains the neural network with the constraints enforced in the relaxed MDD via gradient descent on the training data set. This is detailed in Section 3.1. Then the `validation` function evaluates the performance on a separate validation data set. In line $6-9$ of Algorithm 1, we evaluate if the current MDD has a better performance than the previous one. If the loss on the validation set is decreasing, the algorithm would continue the relaxation; otherwise, the algorithm terminates and returns the current Core-Sp as well as the learned parameters.

For the `relaxMDD` procedure, it takes a relaxed MDD as input and relaxes all its layers from top to bottom to the given width. For each layer, the algorithm repeatedly picks a node and then splits it into two nodes, which corresponds to the `nodeSplit` function until the width of the layer reaches the given width. Every node split is followed by an arc

Algorithm 1: Iterative algorithm for searching optimal performance of Core-Sp.

```
Input: Training set $\mathcal{D}^{tr}$, validation set $\mathcal{D}^{val}$; Parameters $\theta$ of sequence-to-sequence
       neural network; Constraints $\mathcal{C}$; Maximum layer width $\omega_{\max}$.
Output: The best relaxed MDD $\mathcal{M}$ and the learned parameters of the network $\theta$.
1  $\mathcal{L}_{prev} = +\infty$;
2  $\mathcal{M}$ = initMDD($\mathcal{C}, w = 1$);                  /* initialize the width-1 MDD */
3  for $w = 2, \dots, \omega_{\max}$ do
4      $\theta$ = train($\theta$, $\mathcal{M}$, $\mathcal{D}^{tr}$);        /* learn $\theta$ with MDD on training data */
5      $\mathcal{L}$ = validation($\theta$, $\mathcal{M}$, $\mathcal{D}^{val}$);      /* evaluate on validation data */
6      if $\mathcal{L} > \mathcal{L}_{prev}$ then
7          break;
8      else
9          $\mathcal{L}_{prev} = \mathcal{L}$;
10     $\mathcal{M}$ = relaxMDD($\mathcal{M}$, $\mathcal{C}$, $w$);       /* relax MDD with a larger width */
11 return $\mathcal{M}$, $\theta$;                                  /* find the optimal Core-Sp */
12 Procedure relaxMDD($\mathcal{M}$, $\mathcal{C}$, $w$):
13     for layer $= 1, \dots,$ layerSize($\mathcal{M}$) do
14         while |$\mathcal{M}$[layer]| $< w$ do
15             do
16                 u = $\mathcal{M}$[layer].pop();              /* pick node u to split */
17             while |u.in| $\leq 1$;
18             vi, wi = nodeSplit(u.in); /* split incoming arcs of node u */
19             v = node(vi, u.out); w = node(wi, u.out);
20             arcFilter(v, w, $\mathcal{C}$);  /* filter invalid arcs by constraints */
21             $\mathcal{M}$[layer].add(v, w);      /* update the MDD with new nodes */
22             $\mathcal{M}$[layer].delete(u);
23     return $\mathcal{M}$;
```

filtering process, denoted by `arcFilter`, to enforce constraints. Hoda et al. (2010) present MDD arc filtering procedures for various constraint types. Figure 3 gives an example of the `nodeSplit` and `arcFilter` processes on a relaxed MDD.

Note that in node splitting (line 14 − 17 of Algorithm 1), those nodes with only one incoming arc are skipped for the splitting process, *i.e.*, $u.in \geq 1$. In our paper, the heuristic for splitting is: the set of incoming arcs (noted as *.in*) of the original node is randomly assigned to the two newly created nodes (v, w) and then the outgoing arcs (noted as *.out*) are copied to nodes v, w. There are other heuristic methods for node splitting. We refer the readers to Bergman et al. (2016a) for details. The `arcFilter` function is applied to remove those outgoing arcs that lead to constraint violations.

Limitations. While the advantages of Core-Sp will be demonstrated by realistic applications in Section 5, the framework has several limitations. First, the current constraint reasoning module based on MDDs cannot enforce continuous-valued constraints. Indeed, it is possible to apply discretization to transform continuous constraints into discrete ones and then use Core-Sp. However, such discretization may result in very large decision diagrams. Second, the decision diagram is based on a sequence-to-sequence structured prediction model. Other encoding-decoding structures, for example, encoding-decoding on a graph, may require exploring other types of structured prediction models. We leave the study of these limitations as future works.

5. Applications

In this section, we describe the problem settings, the neural network configurations, and the construction of MDDs for three applications.

5.1 Vehicle Dispatching Service Planning

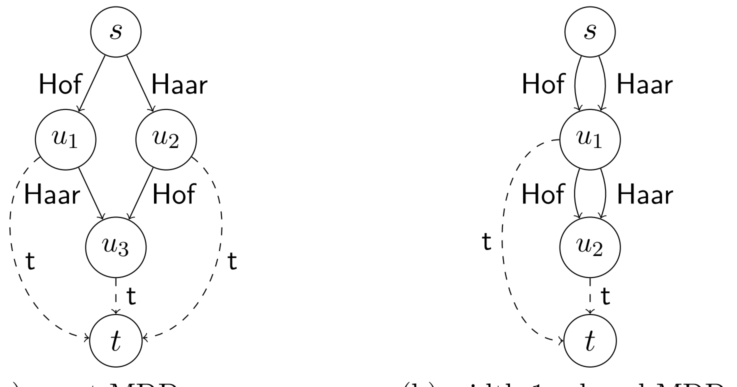
Task Definition. Consider a routing problem in which one needs to dispatch a service vehicle to perform maintenance at a set of locations. The sets of locations differ per day and are rarely the same. Previous routes indicate that the driver does not follow a clear objective, such as minimizing the distance or time. Instead, historical data suggest that the driver has an underlying route preference, such as visiting a shopping mall after leaving a restaurant. Our task is: given the historic routes and a set of requested locations, determine a path that visits all the locations once and only once while capturing the hidden trends embedded in the historical data. To be specific, given a request to visit a set of locations $x = \{x_1, x_2, \dots, x_{T_i}\}$ in the i-th day, determine $y = (y_1, y_2, \dots, y_{T_i})$, which forms a permutation of x and captures the driver's preferences. For this application, we assume an upper bound T on the number of sites to visit per day. In other words, for all i, $T_i \leq T$.

Traditional optimization methods such as integer programming or constraint programming do not work well in this context since they are unable to represent an appropriate objective function for the latent route preference (Braekers et al., 2016). Machine learning models on the other hand can be used to learn the underlying pattern from the historical routes (Junior Mele et al., 2021). Nevertheless, the routes generated from pure machine learning models cannot satisfy key operational constraints. They may visit some locations multiple times, or fail to visit all locations. Post-processing steps, such as removing redundant locations and randomly appending unvisited locations, have been tried to fix the output of machine learning models (Deudon et al., 2018). However, their performance is limited in our experiments (see Section 6.1 for details).

Definition of Constraints. The input of this application is a set of locations to visit for day i: $x = \{x_1, x_2, \dots, x_{T_i}\}$. The goal is to generate a schedule y to represent the order of visiting the locations, where y is a permutation of x. The route y needs to satisfy the following constraints:

- `full-cover` constraint. The delivery route should visit all and only the locations in x. In other words, the set of locations in y is the same as the set in x.
- `all-diff` constraint. The route should not visit one place twice. In other words, $y_j \neq y_k$ for all $y_j, y_k \in y$ and $j \neq k$.

We note that our MDDs can potentially incorporate other constraints such as time windows or precedence constraints (Ciré and van Hoeve, 2013).



(a) exact MDD (b) width-1 relaxed MDD

Figure 6: The MDDs for the vehicle dispatching service planning task. **(a)** An exact MDD that models the visit to "Hof", or "Haar", or both of them. All arcs of solid lines are of the first type and arcs of dashed lines are of the second type, which directs the delivery agent to the stop location t. **(b)** A width-1 relaxed MDD, which is formed by combining nodes u_1 and u_2 of the exact MDD.

MDD Construction. The delivery routes have at most T locations in the data set, so the MDD graph $\mathcal{M}$ would contain $T+2$ layers. There is a single source node s in the first layer and a single sink node t at the last layer. There are two types of arcs in the MDD. For the first type, an arc $e(u,u')$ with label v_i (where $u' \neq t$) in the j-th layer represents that we visit v_i as the j-th location in the schedule. The second type of arcs $e(u,t)$ with label t connects every node to the sink node t, allowing the delivery agent to travel to the ending location at any time.

Figure 6(a) shows an example MDD. Here, the delivery agent needs to visit "Hof" or "Haar" (two towns in Bavaria) or both of them. The arcs of the first type are shown in solid lines and the arcs of the second type are shown in dashed lines. The two arcs $e(s,u_1) = Hof, e(s,u_2) = Haar$ leaving the root node s denote the first location that the delivery agent visit, which can be either "Hof" or "Haar". The three arcs $e(u_1,t) = e(u_2,t) = e(u_3,t) = t$ are of second type. The purpose is to bring the delivery agent to the ending location t. In practice, an MDD that represents all valid paths can be of exponential size with respect to the number of maximum locations. Figure 6(b) shows a width-1 relaxed MDD, formed by combining nodes u_1 and u_2 in Figure 6(a). The paths in relaxed MDDs form a superset of all valid paths. As we can see, "Hof $\rightarrow$ Hof $\rightarrow$ t" is a path in the relaxed MDD, but it violates the `all-diff` constraint.

MDD Arc Filtering. The previously constructed MDD contains routes up to length T considering all possible locations. To enforce the `all-diff` constraint, we apply the arc filtering rules from Ciré and van Hoeve (2013). To enforce the `full-cover` constraint for a specific day with request x, we apply the following steps:

1. Remove all arcs whose label does not belong to x (these are arcs of the first type). This ensures that only locations in x are considered.

2. Remove all arcs with label t, except for arcs $e(u,t)$ where u belongs to layer $|x|+1$. Remove all nodes and associated arcs from the layers between layer $|x|+1$ and the last layer. This ensures that the routes in the MDD have the appropriate length.

3. Remove all nodes and arcs that do not belong to an s-t path.

The first two steps are implemented in a top-down pass of the MDD, while the last step requires an additional bottom-up pass. The total time complexity is therefore linear in the size of the MDD.

Model Structure. We employ the Core-Sp module on a conditional Generative Adversarial Network (cGAN) to generate routes that capture the implicit preferences of the drivers while preserving the operational constraints. In the generative adversarial structure, the generator network G is trained to generate routes to mimic the pattern in the training data set. The discriminator network D is trained to separate the generated routes from the actual ones in the training data set. When training converges, the discriminator should not be able to tell the difference between the true outputs and the structures generated by the generator. In return, the generator generates structures that closely look like the ones in the data set. The Core-Sp module is embedded in the generator and filters out those routes that violate the operational constraints. As a result, the generated routes would satisfy all operational constraints. We employ the conditional GAN model structure because the element-wise loss function is not ideal to measure the distance between the predicted route and the ground truth route. Namely, suppose one route $(y_T, y_{T-1}, \dots, y_1)$ is the reverse of the optimal route $(y_1, y_2, \dots, y_T)$. Both of them may be equally good to fit the delivery constraints as well as the driver's underlying preference. However, an element-wise loss function penalizes the shifted route heavily because it is different from the optimal route in every location.

The overall conditional GAN with the Core-Sp architecture is shown in Figure 7, which is composed of the generator G and the discriminator D. The generator G takes the set of locations x as input and outputs the un-normalized score vectors $(o_1, o_2, \dots)$, where vector o_j denotes the un-normalized likelihood of visiting each location at the j-th step. Core-Sp receives these score vectors and the random values $(z_1, z_2, \dots)$ as inputs, and outputs a valid route $(q_1, q_2, \dots)$. Here, $q_j = (q_{j1}, \dots, q_{jN})$ is a vector, where q_{jk} is an indicator variable representing whether location k is visited in the j-th step. Finally, the discriminator function D tries to separate the predicted route $(q_1, q_2, \cdots)$ and the ground-truth route $(y_1, y_2, \cdots)$. Here, each $y_j = (y_{j1}, \dots, y_{jN})$ is again represented as a vector, where y_{jk} indicates whether to visit location k in the j-th step. The generator G uses an encoder to learn a representation vector for the input and uses a sequential decoder to generate the schedule. In each step, the daily request x is fed as an input vector, where the locations in the requested set are marked as 1. G uses the following LSTM structure to generate the schedule:

$$\begin{aligned} h_j &= \texttt{LSTM}(x, h_{j-1}),\\ o_j &= W h_j, \end{aligned}$$

where the score vector o_j represents the likelihood of picking the next location in the j-th step. The score vector o_j and the random variable z_j are then fed into the Core-Sp module. The Core-Sp module removes invalid locations and produces q_j according to the

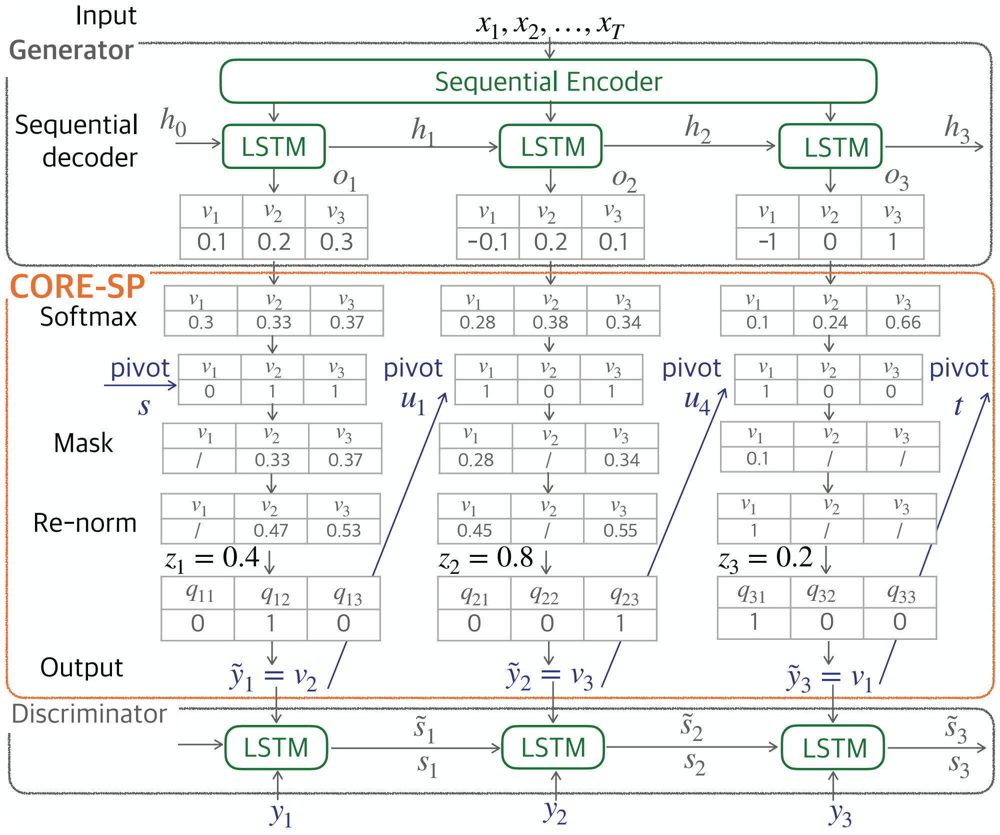
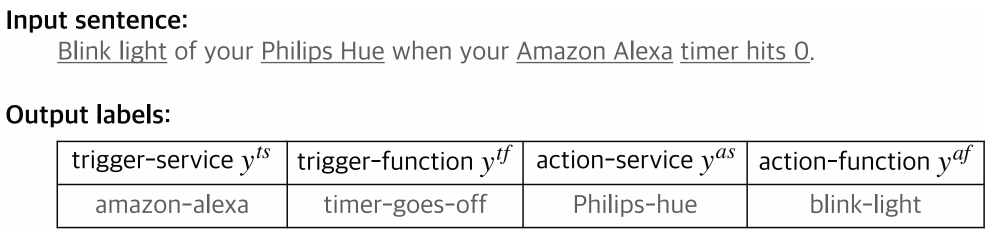


Figure 7: The conditional GAN with Core-Sp module for the vehicle dispatching service planning problem. Vector x represents the requested delivery locations in day i. $(\tilde{y}_1, \tilde{y}_2, \ldots)$ represents the generated path from Core-Sp, represented using indicator vectors $(q_1, q_2, \ldots)$. The generator G takes the set of locations x as input and uses a sequential encoder to learn a representational vector. Then it outputs a sequence of score vector $(o_1, o_2, \ldots)$ using a sequential decoder, where o_j denotes the likelihood of picking the next locations at the j-th step. The Core-Sp module removes invalid locations. The Discriminator D is used to separate the real path $(y_1, y_2, \ldots)$ from the generated path $(\tilde{y}_1, \tilde{y}_2, \ldots)$.

random value of z_j. The detailed equations of Core-Sp were presented in Section 3.1. The discriminator D is trained to separate the generated schedule $q = (q_1, q_2, \ldots, q_{T_i})$ from the real schedule $y = (y_1, y_2, \ldots, y_{T_i})$. It uses the following LSTM structure:

$$\tilde{s}_j = \texttt{LSTM}(q_j, \tilde{s}_{j-1}),$$
$$s_j = \texttt{LSTM}(y_j, s_{j-1}),$$

where $\tilde{s}_j$ denotes the hidden vector after encoding the first j generated locations and s_j denotes the hidden vector after encoding the first j locations in the real data. The output of the discriminator D is $\sigma(Us)$, where U is a linear transformation matrix and s is either s_{T_i} or $\tilde{s}_{T_i}$. The sigmoid activation function is $\sigma(s) = 1/(1 + \exp(-s))$. Overall, D and G are



Input sentence:

Blink light of your Philips Hue when your Amazon Alexa timer hits 0.

Output labels:

trigger-service y^{ts}	trigger-function y^{tf}	action-service y^{as}	action-function y^{af}
amazon-alexa	timer-goes-off	Philips-hue	blink-light

Figure 8: An example of if-then program synthesis task. The input is a natural language description of the program. The output are four labels: `trigger-service`, `trigger-function`, `action-service` and `action-function`. The semantics of the synthesized if-then program are: if `trigger-function` happened at `trigger-service`, then take `action-function` at the `action-service`.

trained by minimizing the loss function in a competing manner using stochastic gradient descent. The loss function $\mathcal{L}$ is:

$$\min_G \max_D \quad \mathbb{E}_{x,y}\left[\log D\left(y,x\right)\right] + \mathbb{E}_{z,x,y}\left[\log\left(1 - D\left(G\left(x,z\right),y\right)\right)\right].$$

5.2 If-Then Program Synthesis

Task Definition. Many internet applications provide automatic services to meet user needs, including daily weather reports and video streaming services. Connectivity platforms such as IFTTT[1] and Zapier[2] can streamline services from different providers by connecting simple services into more complex ones, in the form of if-then programs. For instance, the smart device Philips Hue can automatically blink lights when commands are sent from a cellphone. As another example, Amazon Alexa can be programmed as a timer via voice commands. Given these two services, users can set up an if-then program on the IFTTT platform for more complicated tasks. For example, an if-then program can command the Philips Hue to blink lights when the timer in Amazon Alexa reaches zero. However, it may take several hours for inexperienced users to learn the IFTTT website's interface before they can implement the program above. If such if-then programs can be automatically synthesized from natural language to provide suggestions for inexperienced users, this will help to reduce the overhead in using such platforms and boost the users' efficiency.

We consider the task of generating if-then programs from natural language as a structured prediction task. In our setup, an if-then program is made up of four components: `trigger-service`, `trigger-function`, `action-service`, and `action-function`. The logic is "if `trigger-function` happens in the `trigger-service`, then take the `action-function` from the `action-service`". Such programs can be represented using this pseudo-code:

```
IF trigger-service.trigger-function THEN
    action-service.action-function
```

1. `https://ifttt.com/`
2. `https://zapier.com/`

Figure 8 shows an example of this task. We would like to transform a user's text description: *"Blink light of the Philips Hue when the Amazon Alexa timer hits 0"* into the following if-then program:

```
IF Alexa.timer-go-off  THEN
   Hue.blink-light
```

The challenge in if-then program synthesis is to enforce the constraints between the services and the associated functions. Without enforcing constraints, the output of the structured prediction model may be invalid. For instance, the model can predict "Hue" for `trigger-service`, but perhaps assigns "report rain" to the `trigger-function`, while we know before training that the smart device "Hue" does not provide any weather reporting services.

Definition of Constraints. We introduce the `Functionality` constraint for the if-then program synthesis as follows. Let s be a service. We define a mapping $F(s)$ to be the set of functions that can be associated with s. For example, if s is "weather service", then the output of $F(s)$ is a set that contains functions such as "hourly report", "tomorrow forecast", "severe weather alarms", etc. The `Functionality` constraints for all the four components are:

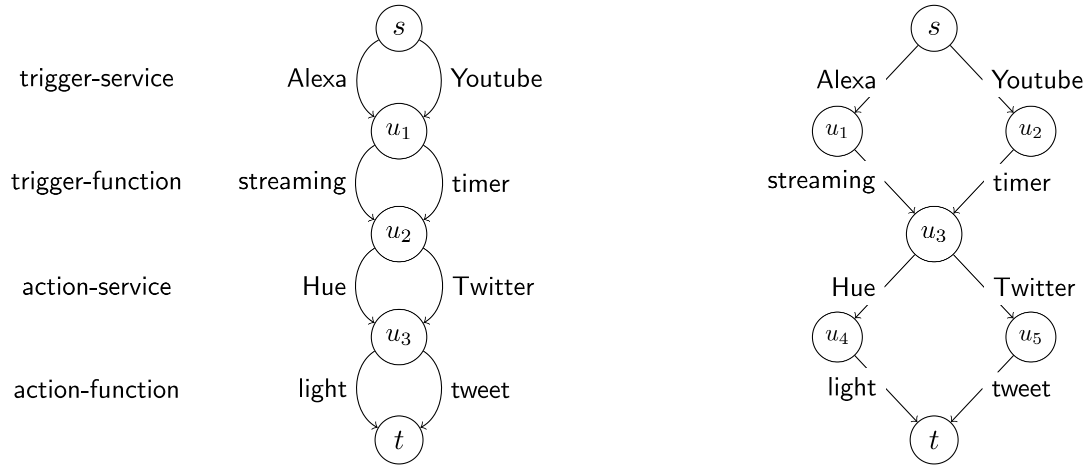
$$\texttt{trigger-service} = s \Rightarrow \texttt{trigger-function} \in F(s),$$
$$\texttt{action-service} = s \Rightarrow \texttt{action-function} \in F(s).$$

For each service s, the mapping $F(s)$ is collected from the associated reference page and provided as prior information.

MDD Construction. The `Functionality` constraints can be represented using an MDD with five layers, where the first layer has one source node s and the last layer has one sink node t. Each arc between the first and second layer corresponds to a value assignment to the variable `trigger-service`. Each arc between the second and third, third and fourth, and fourth and fifth layers corresponds to a value assignment to variable `trigger-function`, `action-service`, and `action-function`, respectively. Figures 9(a) and (b) represent a width-1 and width-2 MDD for if-then program synthesis.

In the MDD, multiple arcs from the first layer can be connected to a single node u in the second layer. The node u hence represents the set of if-then programs that have a given subset of trigger services. The set of arcs leaving from u represents the union of all the trigger functions that are associated with the trigger services connecting u. For example, Figure 9(a) demonstrates that both Alexa and Youtube can be associated with the streaming and timer services. Notice that this is a relaxed MDD. In practice, only Alexa has the timer service and only Youtube has the streaming service. Similar semantic meaning holds for the arcs between the 3rd and 4th layers, representing action services and action functions.

The width of the MDD can be expanded to enforce constraints more precisely. Figure 9(b) shows an example of the width expansion of MDD. Here, the nodes u_1 in (a) is split into two nodes $\tilde{u}_1$, $\hat{u}_1$ in (b). After arc filtering, $\tilde{u}_1$ is connected to "timer" only while $\hat{u}_1$ is connected to "streaming" only because only "Alexa" has the service "timer" and only "Youtube" has the service "streaming". Similar node splitting and arc filtering are applied



(a) width-1 relaxed MDD (b) exact MDD

Figure 9: Examples of a relaxed and an exact MDD for the if-then program synthesis task. The exact MDD in (b) models the constraints that only the timer service is provided by Alexa, and only the streaming service is provided by Youtube. Similarly, only Hue provides the light service, and only Twitter provides the tweet service. (a) is a relaxed MDD, where both trigger services provide both trigger functions, and both action services provide both action functions.

for action services and functions as well. In a nutshell, the relaxed MDD can be expanded into an exact one using repeated node splitting and arc filtering.

Model Structure. We employ Core-Sp on the LatentAttention model proposed by Liu et al. (2016), which achieved the state-of-the-art result on if-then program synthesis. The LatentAttention model is a bidirectional LSTM with residual connection, followed by the self-attention mechanism. To be specific, the bidirectional LSTM (Bi-LSTM) encodes a natural sentence input of length T: $x = (x_1, x_2, \dots, x_T)$ into T latent vectors $(h_1, h_2, \dots, h_T)$. Here, x_j is a one-hot vector representing the j-th word in the sentence. Each vector h_j is a concatenation of a forward vector $\overrightarrow{h_j}$ and a backward vector $\overleftarrow{h_j}$. Suppose vector $\overrightarrow{h_j}$ and $\overleftarrow{h_j}$ are of length m, vector h_j will be of length $2m$. The forward vector $\overrightarrow{h_j}$ is the result of encoding input words $x_1, x_2, \dots, x_j$ from the left through an LSTM, and the backward vector $\overleftarrow{h_j}$ is the result of encoding input words $x_T, \dots, x_{T-j}$ from the right. More precisely, in mathematical form:

$$\begin{aligned}
\overrightarrow{h_j} &= \texttt{LSTM}(x_j, \overrightarrow{h_{j-1}}),\\
\overleftarrow{h_j} &= \texttt{LSTM}(x_{T-j+1}, \overleftarrow{h_{j+1}}),\\
h_j &= \left[\overrightarrow{h_j}; \overleftarrow{h_j}\right].
\end{aligned}$$

where [;] denotes concatenation of two vectors. The detailed equations for the `LSTM` neural network with a residual connection can be found in Greff et al. (2017). In the second step, we encode the sequence of vectors $(h_1, h_2, \dots, h_T)$ into a single vector g through an attention

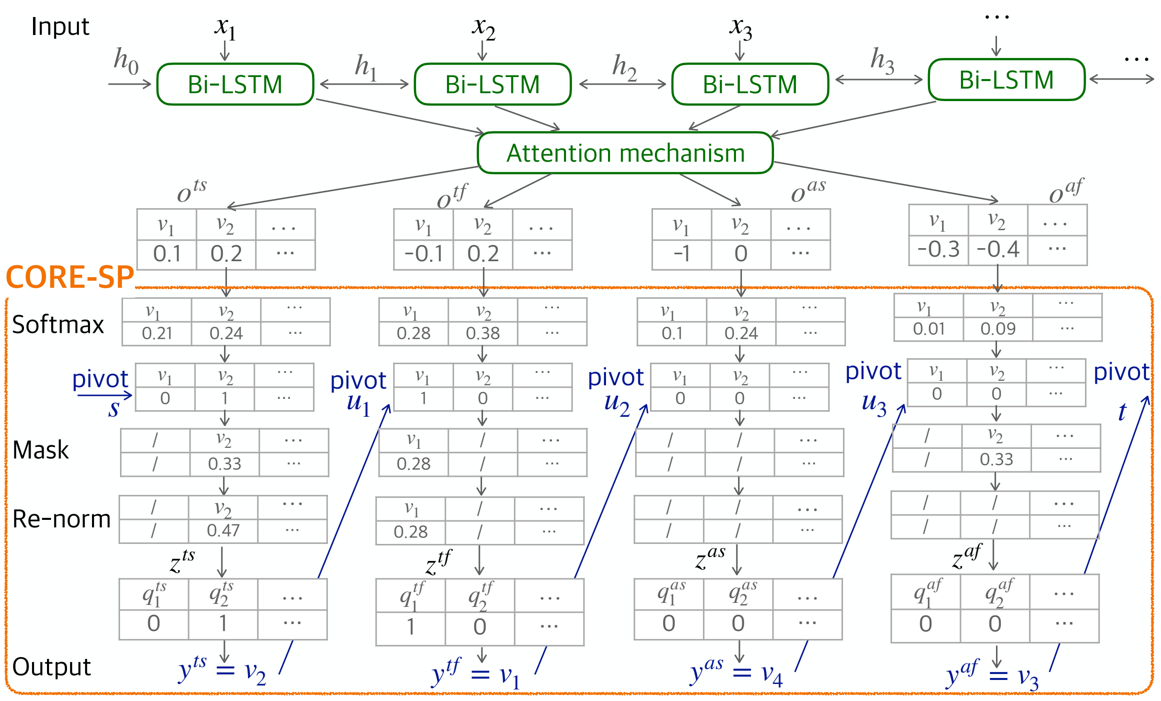


Figure 10: Model structure of If-then program synthesis. Text input $x_1, x_2, \dots, x_T$ is fed into bi-directional LSTM with self attention mechanism. The un-normalized likelihood vectors $o^{\text{ts}}, o^{\text{tf}}, o^{\text{as}}, o^{\text{af}}$ are fed into the Core-Sp module for constraint satisfaction.

mechanism, which is similar to Shaw et al. (2018):

$$\alpha_{jk} = \frac{\exp({h_k}^\top h_j)}{\sum_{k'=1}^{T} \exp({h_{k'}}^\top h_j)}, \qquad \text{for all } j,k \in \{1,\dots,T\},$$
$$g = \sum_{j=1}^{T}\sum_{k=1}^{T} \alpha_{jk} h_k.$$

This Bi-LSTM with attention neural network structure encodes the entire sentence into one latent vector g of size $2m$. Let U^{ts} be a matrix of size $|\#\text{service}| \times 2m$, where $|\#\text{service}|$ is the number of `trigger-service` options. We similarly define matrices U^{tf}, U^{as}, U^{af} for `trigger-function`, `action-service`, `action-function` respectively with their corresponding shapes. The un-normalized likelihoods for `trigger-service` o^{ts}, for `trigger-function` o^{tf}, for `action-service` o^{as}, and for `action-function` o^{af} are defined as

$$o^{\text{ts}} = U^{\text{ts}} g, \quad o^{\text{tf}} = U^{\text{tf}} g, \quad o^{\text{as}} = U^{\text{as}} g, \quad o^{\text{af}} = U^{\text{af}} g.$$

The vectors $o^{\text{ts}}, o^{\text{tf}}, o^{\text{as}}, o^{\text{af}}$ are fed into the Core-Sp module. During training, we use cross-entropy loss as the loss function $\mathcal{L}$ that minimizes the difference between the ground-truth prediction and the probabilities $\tilde{p}^{\text{ts}}$, $\tilde{p}^{\text{tf}}$, $\tilde{p}^{\text{as}}$, $\tilde{p}^{\text{af}}$ produced from Core-Sp (definition of these probabilities are in Equation 6). This training procedure is similar to the teacher-forcing approach used in Sutskever et al. (2014) to accelerate the learning speed. Variables z are used to sample particular (`trigger-service`, `trigger-function`, `action-service`, `action-function`) quadruples from the probability distribution given by $\tilde{p}^{\text{ts}}$, $\tilde{p}^{\text{tf}}$, $\tilde{p}^{\text{as}}$, $\tilde{p}^{\text{af}}$ during testing.

5.3 Text2SQL Generation

Task Definition. Formatted data such as travel records and stock market transactions are stored in relational databases. Currently, accessing the database requires a data scientist who masters the SQL query language. Our task is to automatically synthesize SQL queries from natural language sentences using machine learning. Compared with the data expert approach, SQL query generation requires deeper reasoning across the structure of the database, the semantics of the structured query language, and the understanding of natural language. As shown in Figure 11, the input of the text2SQL generation is a sentence that describes the query in natural language and the table headers in the relational database. The output is a SQL query with the following structure:

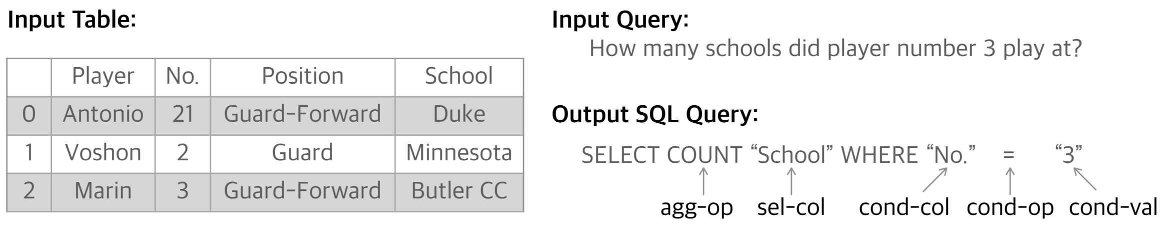
```
SELECT agg-op sel-col
WHERE (cond-col cond-op cond-val) AND ...
```

Here, `SELECT` and `WHERE` are keywords in the SQL language. What we need to predict are: (1) the aggregation operator `agg-op`, which chooses among the set {`empty, COUNT, MIN, MAX, SUM, AVG`}; (2) the column name in selection `sel-col` and (3) the column name in condition `cond-col`, both of which are chosen from the table headers; (4) the conditional operator `cond-op`, which is in $\{=, <, >\}$; (5) the conditional value `cond-val`, which is assumed to be a sub-sequence of the given query. Here, one bracket pair `()` represents one conditional statement. The SQL query may have multiple conditions, which are denoted above by "`...`". Figure 11 displays this SQL query:

```
SELECT COUNT "School"
WHERE "No." = "3"
```

Here `agg-op` is `COUNT`; `sel-col` is "school", which is a column name from the table headers. One `cond-col` is "No.", which also comes from the table headers. The `cond-op` is "=". The `cond-val` is "3", which we assume is from the input query. This example has one condition but multiple conditions are allowed.

Definition of Constraints. Existing generative neural models for this task are not guaranteed to generate a query that follows the grammar of a SQL query. To avoid grammar violations, we compile a set of common SQL grammars as constraints into the Core-Sp module. The Core-Sp module will ensure that all the generated SQL queries follow the grammatical constraints. Our constraints are defined on the operators, namely the conditional operator `cond-op` and the aggregation operator `agg-op`. The domains of these operators are dependent upon the data types of the entities (namely, `cond-col` and `sel-col`) they operate on. Consider the previous example. The `agg-op` can only take values between {`empty, COUNT`}, because the `sel-col` is "school", which is of the string type. More precisely, let s be a column header (the value of `sel-col` or `cond-col`). We define $F_a(s)$ as


Input Table:
Input Query:
How many schools did player number 3 play at?
Output SQL Query:
SELECT COUNT "School" WHERE "No." = "3"
agg-op
sel-col
cond-col
cond-op
cond-val

	Player	No.	Position	School
0	Antonio	21	Guard-Forward	Duke
1	Voshon	2	Guard	Minnesota
2	Marin	3	Guard-Forward	Butler CC

Figure 11: An example for the Text2SQL generation task. The input is the text query "How many schools did player number 3 play at?" and the table header "`Player, No., Position, School`" from the relational database. The output should be the SQL query: `SELECT COUNT "School" WHERE "No." = "3"`.

the set of aggregation operators `agg-op` that can be associated with s, and $F_c(s)$ as the set of condition operators `cond-op` that can be associated with s. That is:

$$F_a(s) = \begin{cases} \{\texttt{empty, COUNT, MIN, MAX, SUM, AVG}\} & \text{if } s \text{ of is numeric type} \\ \{\texttt{empty, COUNT}\} & \text{if } s \text{ of is string type} \end{cases}$$

$$F_c(s) = \begin{cases} \{\texttt{=, >, <}\} & \text{if } s \text{ is of numeric type} \\ \{\texttt{=}\} & \text{if } s \text{ is of string type} \end{cases}$$

We also introduce `dataype` constraints, which are defined as:

$$\texttt{sel-col} = s \Rightarrow \texttt{agg-op} \in F_a(s),$$
$$\texttt{cond-col} = s \Rightarrow \texttt{cond-op} \in F_c(s).$$

Model Structure. We embed the Core-Sp module to SQLova (Hwang et al., 2019), the state-of-the-art neural network for text2SQL generation. SQLova has a sequence-to-sequence architecture. It first encodes a natural language sentence and the table headers into a high-dimensional vector. Then the decoder of SQLova decodes the hidden representation into the predictions of various entities in the SQL query. SQLova first determines the number of conditions in the SQL query and then fills in the (`cond-col`, `cond-op`, `cond-val`) for each condition. The operators `agg-op, cond-op` are predicted as a classification task from a fixed set of operators. Column names `cond-col, sel-col` are predicted from the set of table headers in the relational database. The `cond-val` is predicted by a pointer neural network which points at a span of the input natural language sentence. The selected span of the query is used as the `cond-val` (Dong and Lapata, 2018).

MDD Construction. The associated MDD that encodes the constraints for text2SQL generation is similar to the MDD for if-then program synthesis. The MDD is split into layers and every two layers form a group. One two-layer group is used to enforce constraints on an operator-column name pair. The operator-column name pair can be `agg-op` and `sel-col`, or can be `cond-op` and `cond-col`. Note that there can be only one group of `agg-op` and `sel-col` and more than one group of `cond-op` and `cond-col`. In the first layer of the group, the column name is determined. In the second layer, the invalid operators are ruled out based on the type of the column name selected in the first layer. The two-layer group is copied several times because the SQL query can contain multiple conditions.

6. Results and Analysis

We demonstrate the effectiveness of the Core-Sp module on the three applications from Section 5. We mainly focus on two metrics: (1) the percentage of valid structures generated; and (2) the learning performance. Metric (1) evaluates whether Core-Sp is able to improve constraint satisfaction for the structures generated by neural network models, while metric (2) considers whether Core-Sp improves the overall performance of neural network models in pattern detection from data. For the task of if-then program synthesis and text2SQL generation, we use accuracy as the metric for learning performance. It measures the percentage that the predicted structures match exactly with the ground-truth structures in the testing set. For the vehicle dispatching service planning, we introduce a quantitative metric that measures how close the generated routes resemble those in the training set. The quantitative metric will be discussed in later text. We also demonstrate the effect of the MDD structures, especially the change of the layer width, on the overall performance of the Core-Sp module.

Our experimental results demonstrate the efficiency of Core-Sp in boosting both the percentages of valid structures generated and the learning performance. In terms of constraint satisfaction, the percentage of valid routes generated increases from 1% to 100% for vehicle dispatching service planning with the embedding of Core-Sp on a conditional GAN model. The percentage of valid programs also increases from about 88% to 100% for if-then program synthesis when Core-Sp is added to the LatentAttention model, and from 83% to 100% for text2SQL generation when Core-Sp is added to SQLNova on a hard test set. Both the LatentAttention and SQLNova are state-of-the-art models for the corresponding tasks. Furthermore, the Core-Sp module also helps improve learning performance. For the if-then program synthesis, the accuracy is 44% for Core-Sp compared to 42% for the LatentAttention model. The neural network also converges to relatively higher accuracy with fewer training epochs. In the text2SQL task, the execution accuracy improves from 76.1% obtained from the state-of-the-art SQLNova model to 78.0% while the logical accuracy improves from 58.3% to 62.5% with the Core-Sp module embedded. The code for all the experiments is available at GitHub.[3]

6.1 Vehicle Dispatching Service Planning

Our experiments are on a data set consisting of 29 cities in Bavaria.[4] We vary the number of maximum locations T in the daily requests from 2 to 29 in generating the training and testing sets. We generate $N = 10,000$ instances for every given T. The daily requests are randomly sampled from all sets of locations of the specified size. The optimal delivery paths are generated assuming that the delivery agent is maximizing a hidden reward function:

$$R(y) = \sum_{j=1}^{T-1} \texttt{pref}\left(y_j, y_{j+1}\right). \tag{10}$$

Here the scalar value $\texttt{pref}\left(y_j, y_{j+1}\right) \in [0, 1]$ is the delivery agent's implicit preference to visit location y_{j+1} after leaving the location y_j. When generating the data set, we enumerate

3. Code summary: `https://jiangnanhugo.github.io/CORE-SP/`

4. Instance bays29.tsp from TSPLIB: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`

all valid delivery routes and select the one that maximizes this reward function R. Notice that this reward function R was fixed during the data generation step and was hidden to the machine learning algorithms. During the evaluation, the reward function was used as one quantitative measure for the quality of the routes generated. The higher the reward function values, the better the machine learning algorithm was able to capture the hidden preferences of the delivery agent.

Figure 12 (left) presents the memory requirements to represent relaxed MDDs of varying maximum width for routes with lengths $T = 6, 7, 8, 9$. Figure 12 (right) shows the percentage of valid routes generated during training by Core-Sp with relaxed MDDs of varying maximum widths, for $T = 10$. For maximum width 128, the relaxed MDD can produce over 80% valid routes, but the large memory usage is prohibitive. In particular, for length $T = 29$, the exact MDD will exhaust the memory of our machine. Therefore, we also consider an iterative algorithm that *incrementally* creates the exact MDD alongside the output of the sequential decoder. At step t, we follow the prediction made by the sequential decoder and only expand one step of the MDD starting at the node selected by the sequential decoder. This corresponds to only loading its current outgoing arcs at every predicted step of the exact MDD. Using this idea, we are able to expand the exact MDD even for a larger number of locations ($T = 29$).

Valid Routes Comparison. We evaluate the performance of Core-Sp in generating valid routes, *i.e.*, those satisfying the `all-diff` and `full-cover` constraints, over data sets of different sizes. As a baseline for comparison, we use the output of the conditional GAN without Core-Sp, to which we include the post-processing method by Deudon et al. (2018). The post-processing method uses a mask vector to enforce that the model can only visit the locations in the daily requested set, and removes all the duplicates in the output schedule. In Figure 13 (left), we compare the performance of Core-Sp using exact MDDs against the baseline. To compile the exact MDD, we use the incremental iterative algorithm mentioned above. The figure shows that the conditional GAN (cGAN) can only generate around 0.1% of valid routes, mostly due to visiting some locations more than once. Once we apply the post-processing method to the output generated by the baseline (cGAN+post process), the model's performance is improved to 50% for the data set $T = 2$. However, the post-processing method cannot handle the combinatorial complexity of the dispatching problem, as its performance quickly falls close to the baseline cGAN as the number of locations in the daily requested set increases. In contrast, the percentage of valid routes is always 100% using the exact Core-Sp. The percentage of valid routes using relaxed MDDs (width ≤ 128) is shown in Figure 13 (right). We can see that Core-Sp still produces over 80% valid routes.

Route Reward Comparison. To evaluate the neural model's capability of learning the implicit preferences, we compare the reward function value of the routes generated from the structured prediction algorithms and the ground-truth routes. Notice that the ground truth routes are those that maximize the reward function $R(\cdot)$, which is defined in Equation (10). We define the normalized reward in the following way:

$$\texttt{norm-reward} = \frac{1}{N}\sum_{i=1}^{N}\frac{R(\tilde{y})}{R(y)}, \tag{11}$$

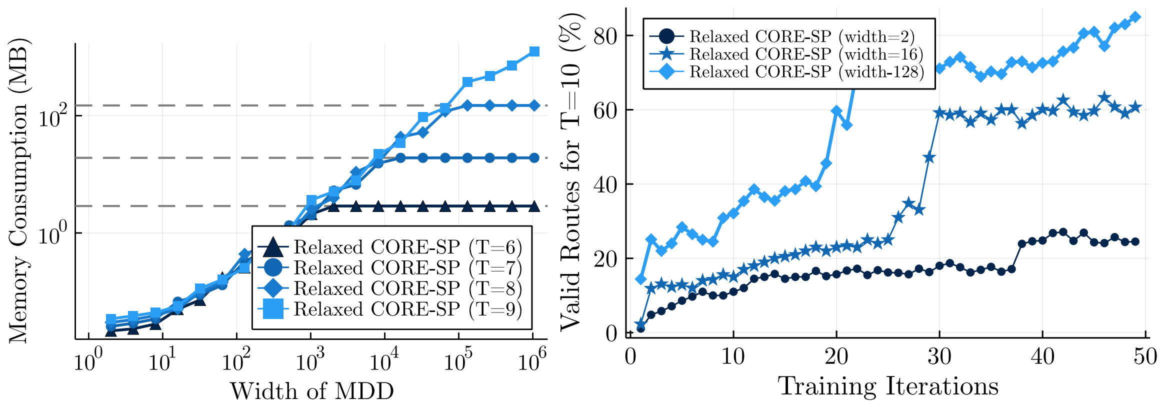


Figure 12: **(Left)** The memory usage of relaxed MDDs. **(Right)** The percentage of valid routes produced by Core-Sp using relaxed MDDs.

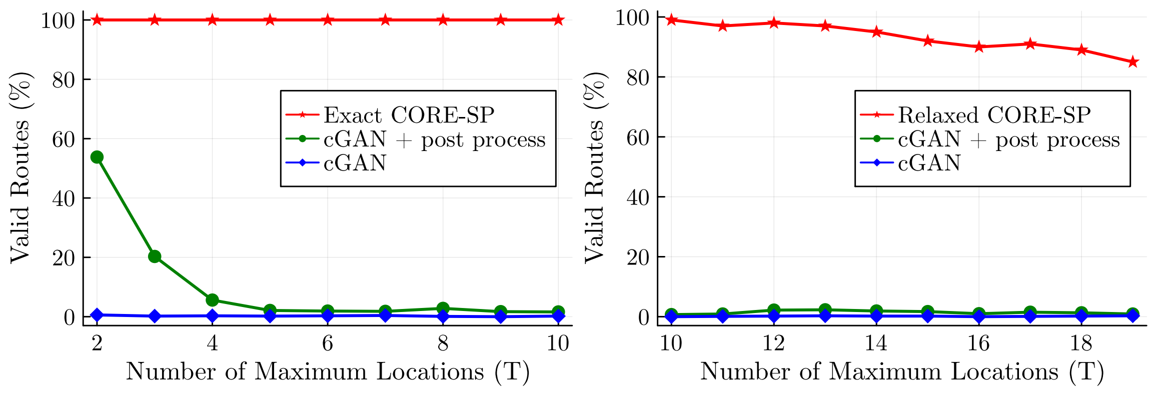


Figure 13: Our exact Core-Sp models outputs 100% valid routes in the vehicle service dispatching task, while competing approaches, namely conditional GAN (cGAN) and cGAN with post-processing cannot guarantee valid routes. Experiments are carried out with varying maximum numbers of locations in the daily requests. **(Left)** Exact MDDs are created by the incremental iterative algorithm described in the main text. **(Right)** Relaxed MDDs are generated with maximum width 2^{20} to ensure that the memory consumption is less than 1 GB.

where routes $\tilde{y}$ are predicted from the machine learning algorithms and y are the ground-truth routes that maximize the hidden reward function. According to our definition, `norm-reward` cannot be greater than 1. The closer `norm-reward` is to 1, the better the generated routes satisfy the hidden preferences of the driver. When computing this metric, we only include valid routes, and we let N be the number of valid routes generated by the algorithm in the test data.

Figure 14 demonstrates the normalized rewards (defined in Equation 11) of the valid routes generated by Core-Sp and the baseline cGAN with post-processing. We can see

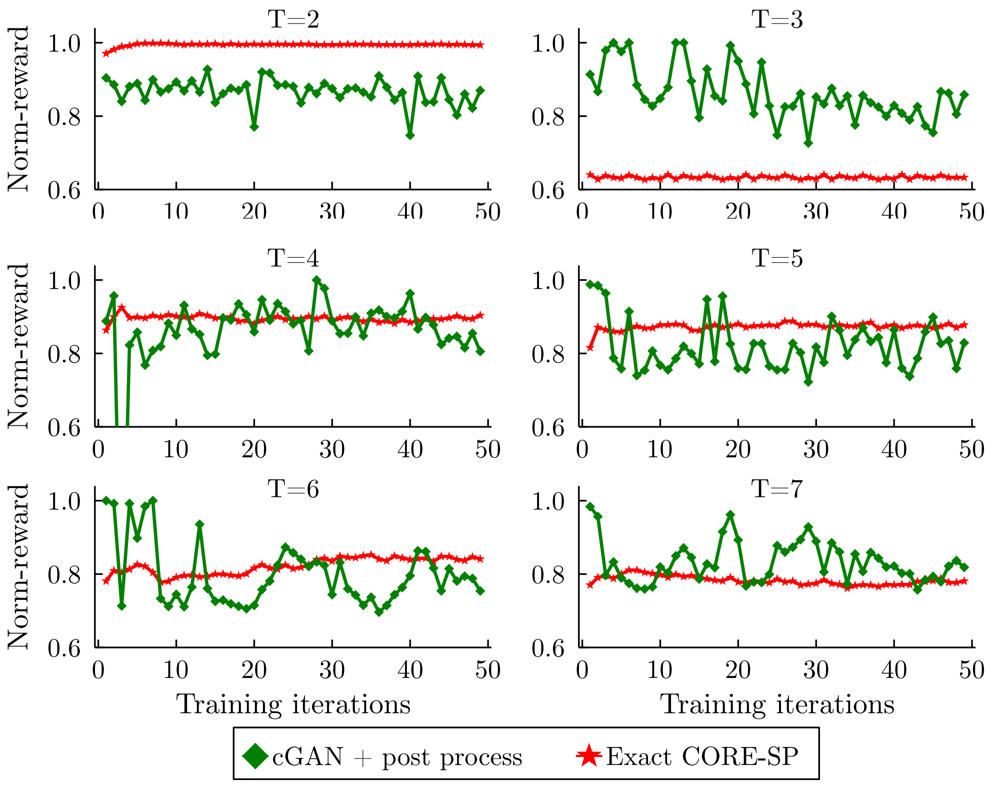


Figure 14: Comparing the normalized reward value of the model for the exact Core-Sp method and cGAN with post-processing, for the vehicle service dispatching application. The hidden driver preferences are reflected by the normalized reward (defined in Equation 11). Core-Sp and cGAN with post-processing both achieve good normalized rewards.

that both methods can generate routes with a normalized reward score between 0.6 and 1. Observe that the routes generated by our Core-Sp module have more stable normalized rewards than the cGAN model with post-processing.

6.2 If-then Program Synthesis

Datasets and Metrics. The data sets for this experiment are crawled from the IFTTT and Zapier websites.[5,6] The statistics of the two data sets are shown in Table 1. The IFTTT data set contains more data samples than the Zapier data set, while the dimensions of the four labels in the Zapier data set are several times larger than those of the IFTTT data set. The sentences in the data set are tokenized by the Spacy library.[7]

To evaluate the performance of different models on this data set, we consider two metrics: the percentage of valid if-then programs, and accuracy. A program is considered valid if

5. IFTTT data set is collected from `https://ifttt.com/`
6. Zapier data set is collected from `https://zapier.com/`
7. Sentence tokenizer: `https://spacy.io/api/tokenizer`

Dataset	#train set	#val set	#test set	#quadruple	#vocabulary
IFTTT	66761	4148	2640	(111, 443, 88, 161)	4000
Zapier	24454	4809	2576	(1353, 1755, 1333, 1466)	3782

Table 1: The statistics for the IFTTT and Zapier data sets.

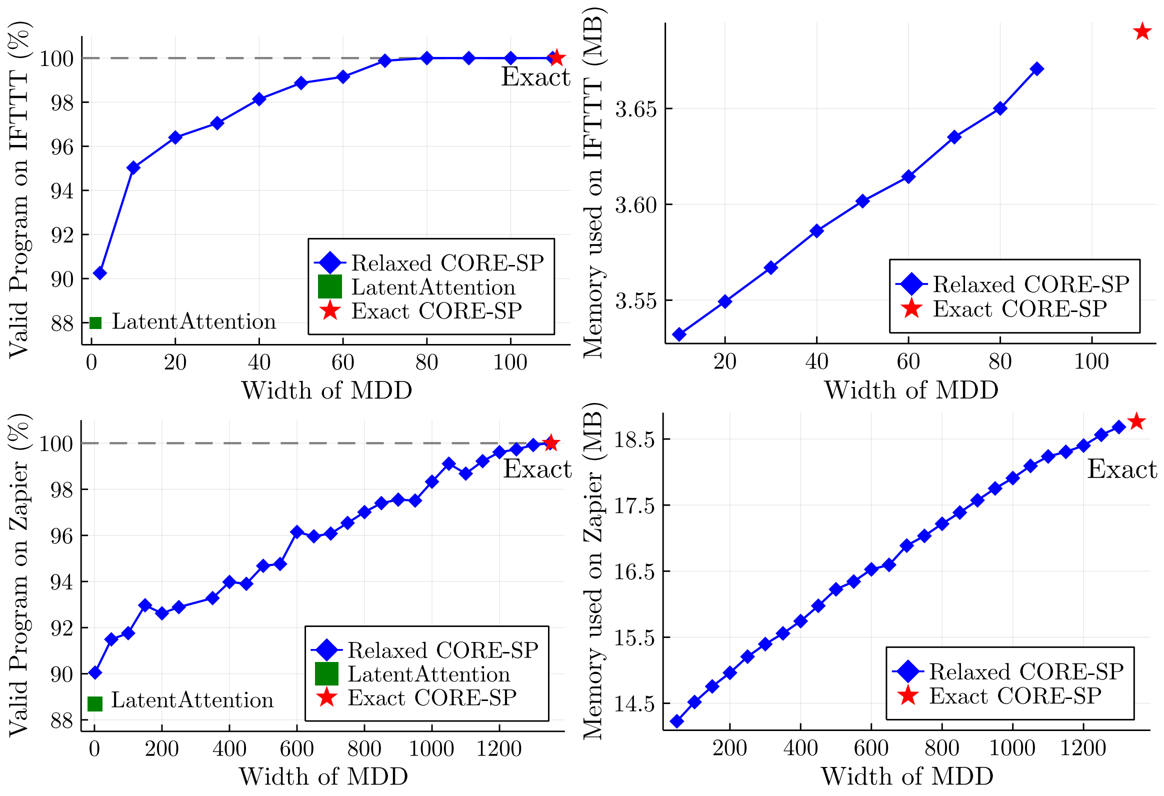


Figure 15: Percentage of valid programs (left column) and MDD memory consumption (right column) on IFTTT and Zapier data sets. Core-Sp outperforms the state-of-the-art approach LatentAttention (Liu et al., 2016) in generating valid if-then programs. The percentages of valid programs generated by Core-Sp embedding MDDs with different widths are shown for the IFTTT (top left) and Zapier (bottom left) data sets. Core-Sp model that embeds the exact MDD produces 100% valid programs on the two data sets. The relaxed and exact MDD for the IFTTT data set takes less than 4 MB and for the Zaiper data set takes less than 20 MB memory space.

it satisfies our defined `Functionality` constraints. The accuracy metric is the percentage of predicted programs that match exactly in all four fields with those in the test set. This metric shows the percentage of correctly predicted programs.

Valid Programs Comparison. Core-Sp significantly boosts the percentage of valid programs generated. In this experiment, we start with evaluating the percentage of valid

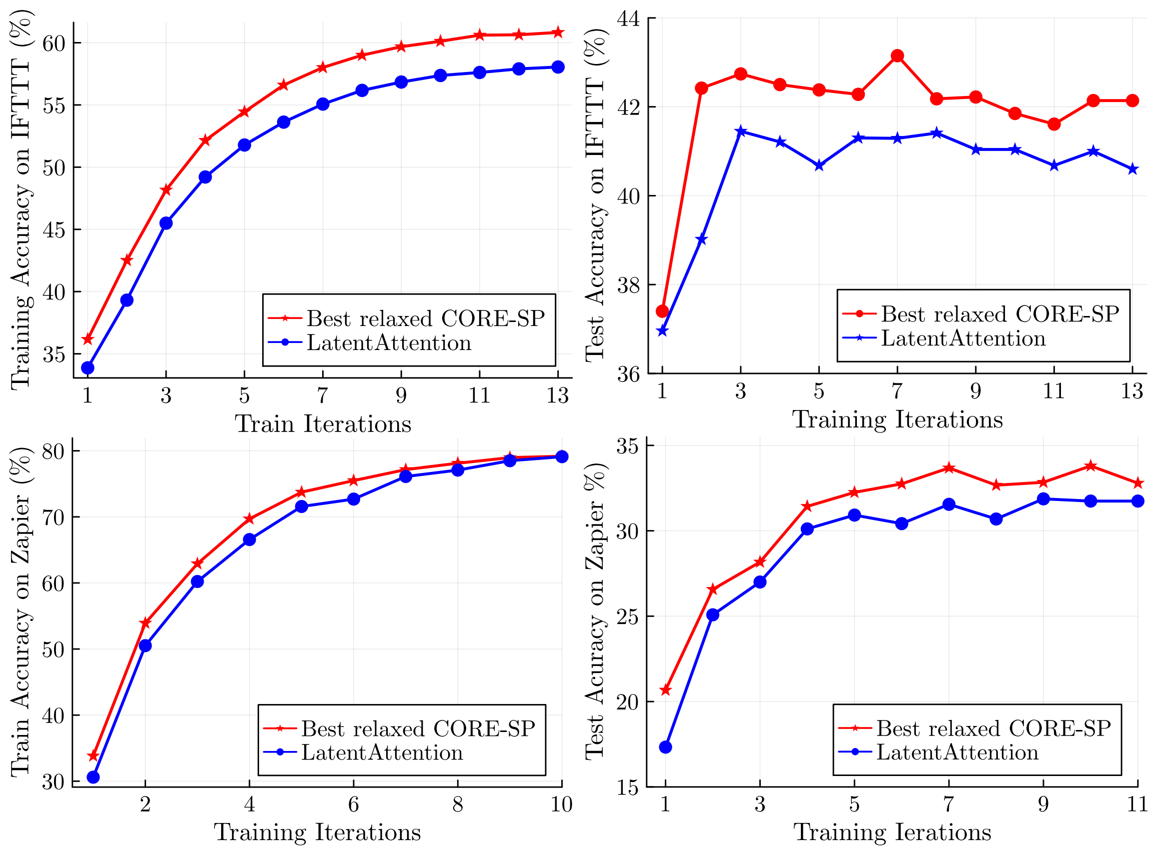


Figure 16: The Core-Sp module (red line) brings approximately 1−2% increase in accuracy for the IFTTT data set and 2% increase for the Zapier data set for the if-then program synthesis task. The LatentAttention model (blue line) is the previous state-of-the-art, which cannot guarantee the validity of the programs generated.

programs generated from the state-of-the-art LatentAttention model without the Core-Sp module. Then we apply the Core-Sp module from Algorithm 1, which iteratively increases the width of the relaxed MDD until we arrive at the exact MDD. Figure 15 shows the performance of all the relaxed and the exact Core-Sp modules when added to the LatentAttention model. Among all programs produced by the LatentAttention model without the Core-Sp layer, around 88% of them are valid on the two data sets. Once we enforce the exact Core-Sp capturing the `Functionality` constraint, all the programs (100%) produced are valid. We also study the effect of restricting the maximum layer width of the MDDs used in Algorithm 1. We evaluate Core-Sp with MDDs of width-2 up to the largest width, which is width-111 for IFTTT and width-1353 for Zapier. The percentage of valid programs on a separate testing set is shown in the blue lines. The performance of the relaxed Core-Sp increases gradually with the increase of the MDD width.

Accuracy Comparison. Figure 16 compares the training set and testing set accuracy for the state-of-the-art LatentAttention model and Core-Sp as the training progresses.

	IFTTT			Zapier		
Methods	Width	Accuracy	Valid (%)	Width	Accuracy	Valid (%)
LatentAttention	N/A	42.17%	87.51%	N/A	31.74%	88.00%
Best relaxed Core-Sp	80	**44.12%**	99.19%	1200	**34.28%**	99.53%
Exact Core-Sp	111	43.07%	**100%**	1353	32.83%	**100%**

Table 2: The relaxed and exact Core-Sp modules boost the percentage of valid programs generated and the accuracy for the if-then program synthesis task on both the IFTTT and the Zapier data sets. Exact Core-Sp produces 100% valid programs while Core-Sp with the best relaxed MDD produced by Algorithm 1 leads to the best accuracy in the prediction and close to 100% valid programs.

We also collect the results of the LatentAttention model without Core-Sp, the model with the best relaxed Core-Sp model (in terms of accuracy) and with the exact Core-Sp model on the two data sets in Table 2. The best relaxed Core-Sp model achieves $1-2\%$ higher accuracy than the LatentAttention model and still generates around 11% more valid programs than the LatentAttention model. Similarly, the model with the exact Core-Sp module improves approximately 1% in accuracy but generates 100% valid programs.

6.3 SQL Query Generation from Natural Language

Dataset and Metrics. We conduct experiments on the large-scale WikiSQL data set (Zhong et al., 2017), which contains 80,654 examples of questions and SQL queries distributed across 24,241 tables from Wikipedia. We observe that most of the SQL queries are not complex. Therefore, we further select queries within the data set to form a moderate and a hard test set. The *moderate test set* consists of those queries containing at least one conditional statement (*i.e.*, "`cond-col cond-op cond-val`"). The *hard test set* is composed of those queries that have at least two conditional statements.

The metrics applied for this task are: 1) Percentage of valid SQL queries, *i.e.*, generated queries that satisfy the `datatype` constraint. 2) Execution accuracy. A generated query is considered correct if the returned value of executing the generated SQL query matches the returned value from the ground truth query. 3) Logical accuracy, which evaluates the percentage of the generated queries that match exactly the ground truth queries in every field. The implementation is based on SQLNova. We use the BERT-base model (Devlin et al., 2019) as the word embedding. The entire model takes up to 3 days to train for 50 epochs. We choose the model that achieves the best execution accuracy on the validation data set for both the baseline and Core-Sp and calculate the corresponding statistics reflected in Table 3.

Valid SQL Queries Comparison. As shown in Table 3, SQLNova with the Core-Sp module embedded generates 100% valid SQL programs, demonstrating 0.7% improvement over the original SQLNova model on the full testing set. On the moderate testing set, the improvement increases to 5.7%. On the most difficult hard testing set, the improvement becomes 16.3%. Due to the fact that a majority of the SQL queries in the full test set have `empty` value at `cond-op` and `=` value at `sel-op`, SQLNova has a high probability to predict

Accuracy per component	Full test set SQLNova	Full test set Core-Sp	Moderate test set SQLNova	Moderate test set Core-Sp	Hard test set SQLNova	Hard test set Core-Sp
`sel-col`	96.3%	96.3%	96.4%	**97.0%**	96.6%	**97.7%**
`agg-op`	**89.8%**	89.7%	75.7%	**77.8%**	75.4%	**75.8%**
`#WHERE`	**98.1%**	97.9%	98.5%	**98.6%**	**98.9%**	98.5%
`cond-col`	93.6%	93.6%	**94.0%**	93.8%	93.6%	**93.7%**
`cond-op`	96.7%	**96.9%**	89.8%	**91.6%**	84.8%	**87.9%**
`where-val-idx`	94.5%	**94.8%**	89.4%	**92.3%**	86.7%	**87.5%**
`where-val`	94.7%	**94.9%**	89.3%	**92.2%**	86.4$	**87.1%**

Overall Accuracy	Full test set SQLNova	Full test set Core-Sp	Moderate test set SQLNova	Moderate test set Core-Sp	Hard test set SQLNova	Hard test set Core-Sp
Logical Accuracy	79.3%	**79.9%**	61.6%	**65.8%**	58.3%	**62.5%**
Execution Accuracy	85.5%	**86.1%**	75.4%	**79.1%**	76.1%	**78.0%**
Valid SQL	99.3%	**100.0%**	94.3%	**100%**	83.7%	**100%**

Table 3: Core-Sp outperforms the previous state-of-the-art SQLNova on three testing sets in SQL query generation. Core-Sp leads to 100% valid SQL queries generated and increases in both the execution accuracy and the logical accuracy compared with SQLNova for the Text2SQL generation task. The top table shows the accuracy of predicting each field in the SQL queries for both models.

prevalent labels in the data set and coincidentally satisfies the SQL grammar. This is the main reason that our relative improvement is not significant for the full test set.

Execution and Logical Accuracy. Figure 17 compares SQLNova and the *exact* Core-Sp model over execution and logical accuracy metrics as the training progresses. We also collect the accuracy of predicting each field in the SQL queries as shown in the table (top) of Table 3. The execution and logical accuracy are shown at the bottom of Table 3. Core-Sp gains improvement for predicting `sel-col`, `cond-op`, `where-val-idx` and `where-val` components. For the other components in the SQL queries, the difference in accuracy between Core-Sp and SQLNova is less than 0.4%. In terms of the execution accuracy, the exact Core-Sp is higher than SQLNova by 0.6%, 3.7% and 1.9% on the full, moderate, and hard test sets, respectively. In terms of the logical accuracy, the exact Core-Sp is higher than SQLNova by 0.6%, 4.2%, and 4.2% for the three testing sets. The improvement in the execution and logical accuracy is due to the fact that the Core-Sp module removes invalid operators during SQL generation and as a consequence reduces the modeling space.

7. Conclusion

In this work, we proposed Core-Sp, an end-to-end neural module that embeds constraint reasoning into machine learning for structured prediction problems. Core-Sp represents the constraints using decision diagrams and filters out invalid solutions. Core-Sp is then embedded into a neural network which can be trained in an end-to-end fashion. We demonstrate the effectiveness of Core-Sp on three structured prediction applications including

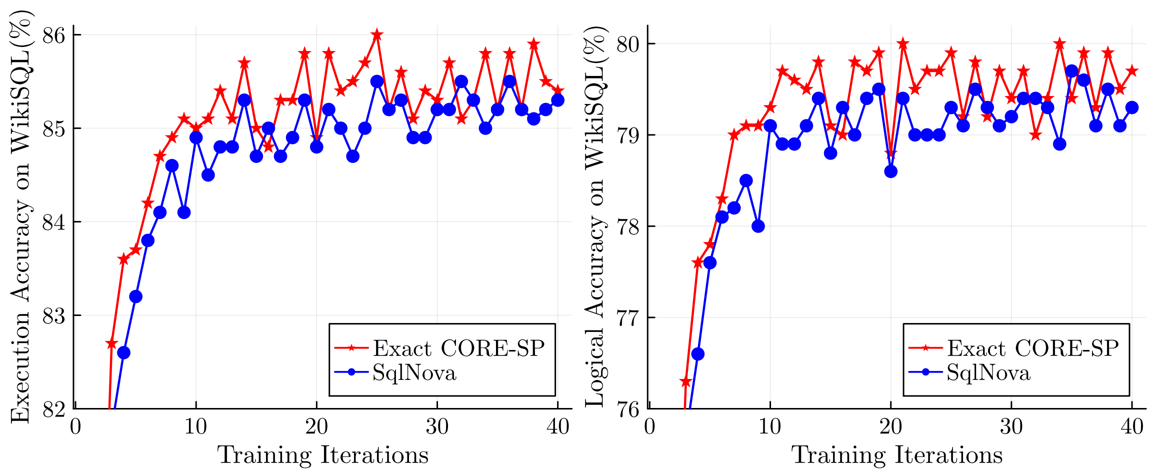


Figure 17: The execution accuracy (left) and logical accuracy (right) over training iterations for both Core-Sp and SQLNova. Core-Sp leads to higher execution and logical accuracy throughout the training iterations.

vehicle dispatching service planning, if-then program synthesis, and Text2SQL generation. We also propose an iterative search algorithm to find the optimal decision diagram structure for these applications. We show that the Core-Sp module improves constraint satisfaction in all three applications. In addition, Core-Sp reduces the modeling space. As a consequence, neural networks with Core-Sp embedded learn faster and generalize better than the pure neural network models. For future work, we plan to generalize Core-Sp in continuous domains and in reinforcement learning.

Acknowledgments

We thank all the reviewers for their constructive comments. This research was supported by NSF grants IIS-1850243, CCF-1918327, CCF-1918102, and Office of Naval Research Grant No. N00014-21-1-2240. M. Z. completed this work when he was a master's student at Purdue University.

References

Hajar Ait Addi, Christian Bessiere, Redouane Ezzahir, and Nadjib Lazaar. Time-bounded query generator for constraint acquisition. In *CPAIOR*, volume 10848 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2018.

Sheldon B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

Brandon Amos and J. Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 136–145. PMLR, 2017.

Henrik Reif Andersen, Tarik Hadzic, John N. Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2007.

G. Bakır, T. Hofmann, A. J. Smola, B. Schölkopf, B. Taskar, and S. V. N. Vishwanathan, editors. *Predicting Structured Data*. The MIT Press, 2007.

David Belanger and Andrew McCallum. Structured prediction energy networks. In *ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 983–992. JMLR.org, 2016.

Nicolas Beldiceanu and Helmut Simonis. A model seeker: Extracting global constraint models from positive examples. In *CP*, volume 7514 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2012.

Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. In *ICLR (Workshop)*. OpenReview.net, 2017.

Kevin Bello, Asish Ghoshal, and Jean Honorio. Minimax bounds for structured prediction based on factor graphs. In *AISTATS*, volume 108 of *Proceedings of Machine Learning Research*, pages 213–222. PMLR, 2020.

David Bergman, André A. Ciré, Willem-Jan van Hoeve, and John N. Hooker. *Decision Diagrams for Optimization*. Artificial Intelligence: Foundations, Theory, and Algorithms. Springer, 2016a.

David Bergman, André Augusto Ciré, Willem-Jan van Hoeve, and John N. Hooker. Discrete optimization with decision diagrams. *INFORMS J. Comput.*, 28(1):47–66, 2016b.

Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. Constraint acquisition. *Artif. Intell.*, 244:315–342, 2017.

Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

Kris Braekers, Katrien Ramaekers, and Inneke Van Nieuwenhuyse. The vehicle routing problem: State of the art classification and review. *Computers & Industrial Engineering*, 99:300–313, 2016.

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

Di Chen, Yexiang Xue, and Carla P. Gomes. End-to-end learning for the deep multivariate probit model. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 931–940. PMLR, 2018.

Arthur Choi, Yexiang Xue, and Adnan Darwiche. Same-decision probability: A confidence measure for threshold-based decisions. *Int. J. Approx. Reason.*, 53(9):1415–1428, 2012.

André A. Ciré and Willem Jan van Hoeve. Multivalued decision diagrams for sequencing problems. *Oper. Res.*, 61(6):1411–1428, 2013.

Remi Coletta, Christian Bessiere, Barry O'Sullivan, Eugene C. Freuder, Sarah O'Connell, and Joël Quinqueton. Constraint acquisition as semi-automatic modeling. In *SGAI Conf*, pages 111–124. Springer, 2003.

Hanjun Dai, Yingtao Tian, Bo Dai, Steven Skiena, and Le Song. Syntax-directed variational autoencoder for structured data. In *ICLR (Poster)*. OpenReview.net, 2018.

Krzysztof Dembczynski, Weiwei Cheng, and Eyke Hüllermeier. Bayes optimal multilabel classification via probabilistic classifier chains. In *ICML*, pages 279–286. Omnipress, 2010.

Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning heuristics for the TSP by policy gradient. In *CPAIOR*, volume 10848 of *Lecture Notes in Computer Science*, pages 170–181. Springer, 2018.

Daniel Deutsch, Shyam Upadhyay, and Dan Roth. A general-purpose algorithm for constrained sequential inference. In *CoNLL*, pages 482–492. Association for Computational Linguistics, 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186. Association for Computational Linguistics, 2019.

Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. In *ACL*, pages 731–742. Association for Computational Linguistics, 2018.

Aaron M. Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. Mipaal: Mixed integer program as a layer. In *AAAI*, pages 1504–1511. AAAI Press, 2020.

Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Computers*, 39(5):710–713, 1990.

Andrea Galassi, Michele Lombardi, Paola Mello, and Michela Milano. Model agnostic solution of csps via deep learning: A preliminary study. In *CPAIOR*, volume 10848 of *Lecture Notes in Computer Science*, pages 254–262. Springer, 2018.

Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741, 1984.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, pages 2672–2680, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray

Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. LSTM: A search space odyssey. *IEEE Trans. Neural Networks Learn. Syst.*, 28(10):2222–2232, 2017.

Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *ACL*, pages 1051–1062. Association for Computational Linguistics, 2017.

Eric Heim. Constrained generative adversarial networks for interactive image generation. In *CVPR*, pages 10753–10761. Computer Vision Foundation / IEEE, 2019.

Samid Hoda, Willem Jan van Hoeve, and John N. Hooker. A systematic approach to mdd-based constraint programming. In *CP*, volume 6308 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2010.

Wonseok Hwang, Jinyeung Yim, Seunghyun Park, and Minjoon Seo. A comprehensive exploration on wikisql with table-aware word contextualization. *NeurIPS Workshop on Knowledge Representation & Reasoning Meets Machine Learning*, 2019.

Wengong Jin, Regina Barzilay, and Tommi S. Jaakkola. Junction tree variational autoencoder for molecular graph generation. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 2328–2337. PMLR, 2018.

Umberto Junior Mele, Luca Maria Gambardella, and Roberto Montemanni. Machine learning approaches for the traveling salesman problem: A survey. In *2021 The 8th International Conference on Industrial Engineering and Applications (Europe)*, pages 182–186, 2021.

Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *NIPS*, pages 6348–6358, 2017.

Matt J. Kusner, Brooks Paige, and José Miguel Hernández-Lobato. Grammar variational autoencoder. In *ICML*, volume 70 of *Proceedings of Machine Learning Research*, pages 1945–1954. PMLR, 2017.

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, pages 282–289. Morgan Kaufmann, 2001.

Arnaud Lallouet and Andrei Legtchenko. Building consistencies for partially defined constraints with decision trees and neural networks. *Int. J. Artif. Intell. Tools*, 16(4):683–706, 2007.

Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. In *ICTAI*, pages 45–52. IEEE Computer Society, 2010.

Chang Liu, Xinyun Chen, Eui Chul Richard Shin, Mingcheng Chen, and Dawn Xiaodong Song. Latent attention for if-then program synthesis. In *NIPS*, pages 4574–4582, 2016.

Michele Lombardi and Stefano Gualandi. A lagrangian propagator for artificial neural networks in constraint programming. *Constraints*, 21(4):435–462, 2016.

Michele Lombardi, Michela Milano, and Andrea Bartolini. Empirical decision model learning. *Artif. Intell.*, 244:343–367, 2017.

Ryszard S. Michalski and John R. Anderson. *Machine learning - an artificial intelligence approach*. Symbolic computation. Springer, 1984.

Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

Vlad Niculae and André F. T. Martins. Lp-sparsemap: Differentiable relaxed optimization for sparse structured prediction. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pages 7348–7359. PMLR, 2020.

Vlad Niculae, André F. T. Martins, Mathieu Blondel, and Claire Cardie. Sparsemap: Differentiable sparse structured inference. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 3796–3805. PMLR, 2018.

Xingyuan Pan and Vivek Srikumar. Learning to speed up structured output prediction. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 3993–4002. PMLR, 2018.

Ben Peters, Vlad Niculae, and André F. T. Martins. Sparse sequence-to-sequence models. In *ACL*, pages 1504–1519. Association for Computational Linguistics, 2019.

Vasin Punyakanok, Dan Roth, Wen-tau Yih, and Dav Zimak. Semantic role labeling via integer linear programming inference. In *COLING*, pages 1346–1353. Association for Computational Linguistics, 2004.

Jesse Read, Luca Martino, Pablo M. Olmos, and David Luengo. Scalable multi-output label prediction: From classifier chains to classifier trellises. *Pattern Recognit.*, 48:2096–2109, 2015.

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 1530–1538. JMLR.org, 2015.

Dan Roth and Wen-tau Yih. Integer linear programming inference for conditional random fields. In *ICML*, volume 119 of *ACM International Conference Proceeding Series*, pages 736–743. ACM, 2005.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *ICLR (Poster)*. OpenReview.net, 2019.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *NAACL-HLT*, pages 464–468. Association for Computational Linguistics, 2018.

Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL):73:1–73:29, 2019.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, pages 1631–1642. ACL, 2013.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, pages 3104–3112, 2014.

Luming Tang, Yexiang Xue, Di Chen, and Carla P. Gomes. Multi-entity dependence learning with rich context via conditional variational auto-encoder. In *AAAI*, pages 824–832. AAAI Press, 2018.

Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. Large margin methods for structured and interdependent output variables. *J. Mach. Learn. Res.*, 6:1453–1484, 2005.

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

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *NIPS*, pages 2692–2700, 2015.

Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. Society for Industrial and Applied Mathematics, 2000.

Hao Wu, Ziyang Chen, Weiwei Sun, Baihua Zheng, and Wei Wang. Modeling trajectories with recurrent neural networks. In *IJCAI*, pages 3083–3090. ijcai.org, 2017.

Rongjing Xiang and Jennifer Neville. Collective inference for network data with copula latent markov networks. In *WSDM*, pages 647–656. ACM, 2013.

Yang Zeng, Jin-Long Wu, and Heng Xiao. Enforcing imprecise constraints on generative adversarial networks for emulating physical systems. *Communications in Computational Physics*, 30(3):635–665, 2021.

Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.