# Embedding Decision Diagrams into Generative Adversarial Networks*

Yexiang Xue[1] and Willem-Jan van Hoeve[2]

[1] Department of Computer Science, Purdue University, USA.
yexiang@purdue.edu
[2] Tepper School of Business, Carnegie Mellon University, USA.
vanhoeve@andrew.cmu.edu

**Abstract.** Many real-world decision-making problems do not possess a clearly defined objective function, but instead aim to find solutions that capture implicit user preferences. This makes it challenging to directly apply classical optimization technology such as integer programming or constraint programming. Machine learning provides an alternative by learning the agents' decision-making implicitly via neural networks. However, solutions generated by neural networks often fail to satisfy physical or operational constraints. We propose a hybrid approach, DDGan, that embeds a Decision Diagram (DD) into a Generative Adversarial Network (GAN). In DDGan, the solutions generated from the neural network are filtered through a decision diagram module to ensure feasibility. DDGan thus combines the benefits of machine learning and constraint reasoning. When applied to the problem of schedule generation, we demonstrate that DDGan generates schedules that reflect the agents' implicit preferences, and better satisfy operational constraints.

## 1 Introduction

Traditional management science approaches to decision-making involve defining a mathematical model of the situation, including decision variables, constraints, and an objective function to be optimized. While common objectives such as cost minimization or profit maximization are widely applied, many operational decision-making processes depend on factors that cannot be captured easily by a single mathematical expression. For example, in production planning and scheduling problems one typically takes into account priority classes of jobs and due dates, but ideally also (soft) business rules or the preferences of the workers who execute the plan. Those rules and preferences may be observed from historic data, but creating a model that results in, say, a linear objective function is far from straightforward. Instead, one may represent the objective function, or even the constraints, using machine learning models that are then embedded into the optimization models; we refer to [24] for a recent survey.

In this work, we study the integration of machine learning and constraint reasoning in the context of sequential decision making. In particular, we aim

* This work was supported by Office of Naval Research grant N00014-18-1-2129.

to extend (recursive) generative adversarial neural networks (GANs) with constraint reasoning. We assume that the context specifies certain physical or operational requirements, within which we need to find solutions that are similar to the decisions that were made historically, as in the following stylized example.

*Example 1.* Consider a routing problem in which we need to dispatch a service vehicle to perform maintenance at a set of locations. The set of locations differs per day and is almost never the same. Previous routes indicate that the driver does not follow a distance or time optimal sequence of visits, even though there are no side constraints such as time windows or precedence relations. Instead, the routes suggest that the driver has an underlying route preference, that is exposed by, e.g., visiting shopping and lunch areas at specific times of the day. Our decision making task is now: Given the historic routes and a set of locations to visit today, determine a route that 1) visits all locations, and 2) is most similar to the historic routes. In addition, we might add further restrictions such as a maximum time limit, for example 8 hours.

In these contexts, traditional optimization methods such as integer programming or constraint programming cannot be applied directly since they are unable to represent an appropriate objective function. Instead, it is natural to represent the structure and preferences from the historic solutions using a machine learning model. For example, we could design and train a generative adversarial neural network (GAN) for this purpose, which will be able to produce sequences of decisions that aim to be similar to the historic data. However, because GANs are trained with respect to an objective function (loss function) to be minimized, hard operational constraints cannot be directly enforced. For example, in case of the routing problem above, the sequences produced by the GAN usually fail to visit all locations, visit some locations multiple times, or fail to recognize constraints such as the maximum time limit.

**Contributions.** We propose a hybrid approach, which we call DDGAN, that embeds a decision diagram (DD) into a generative adversarial neural network. The decision diagram represents the constraint set and serves as a filter for the solutions generated by the GAN, to ensure feasibility. As proof of concept, we develop a DDGAN to represent routing problems as in Example 1. We show that without the DD module, the GAN indeed produces sequences that are rarely feasible, while the DD filter substantially increases the feasibility. Moreover, we show that DDGAN converges much more smoothly than the GAN.

We note that, in principle, any constraint reasoning module could have been applied; e.g., we could embed an integer program or constraint program that contains all constraints of the problem. The variable/value assignments suggested by the GAN can then be checked for feasibility by running a complete search, but this is time consuming. By compiling a decision diagram offline, we can check for feasibility instantly during execution. Moreover, for larger problems we can apply relaxed decision diagrams of polynomial size that may not guarantee feasibility in all cases, but can produce much better solutions that those generated by the GAN alone.

## 2 Related Work

Within the broader context of learning constraint models, several works have studied automated constraint acquisition from historic data or (user-)generated queries, including [11, 23, 4, 7, 1]. These approaches use partial or complete examples to identify constraints that can be added to the model. The type of constraints that can be learned depends on the formalism that is used, for example linear constraints for integer programming or global constraints for constraint programming.

A different approach is to embed a machine learning model into the optimization formalism, e.g., by extending a constraint system with appropriate global constraints. For example, [22] integrate neural networks and decision trees in constraint programming, while [25, 26] introduce a 'Neuron' global constraint that represents a pre-trained neural network. Another series of approaches based on Grammar Variational Autoencoders [21, 12, 19] 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 is also a structured domain. Compared to our work, their models are not conditional, and therefore cannot solve decision-making problems with varying contextual information.

Machine learning approaches have also been used to solve optimization problems. This includes the works [14, 30], which use neural networks to extend partial solutions to complete ones. The authors of [5] handle the traveling salesman problem by framing it as reinforcement learning. Approaches based on Neural Turing Machines [16] employ neural networks with external memory for discrete structure generation. More recently, the authors of [20] solve graph combinatorial optimization problems, and employ neural networks to learn the heuristics in backtrack-free searching. The scopes of these works are different from ours, since they do not deal with optimization problems without clear objective functions. Recently, the work of [29] combine reasoning and learning using a max-margin approach in hybrid domains based on Satisfiability Modulo Theories (SMT). They also show applications in constructive preference elicitation [13]. Compared to our work, their approach formulates the entire learning and reasoning problem as a single SMT, while we combine reasoning and learning tools, namely the neural networks and decision diagrams, into a unified framework.

## 3 Preliminaries

### 3.1 Structure Generation via Generative Adversarial Networks

Finding structured sequences in presence of implicit preferences is a more complex problem compared to classical supervised learning, in which a classifier is trained to make single-bit predictions that match those in the dataset. The problem of finding such sequences is broadly classified as a *structure generation* problem in machine learning, which arises naturally in natural language processing [17, 18], computer vision [8, 28], and chemistry engineering [21].

Specifically, we are given common context $R_c$ and a historical data set $\mathcal{D} = \{(R_1, S_1), (R_2, S_2), \ldots, (R_{t-1}, S_{t-1})\}$, in which $R_i$ represents the contextual information specific to data point $i$ and $S_i$ the decision sequence for scenario $i = 1, \ldots, t-1$. For our routing example, $R_i$ represents the daily location requests while $S_i$ is the ordered sequence of visits that covers the requested locations in $R_i$. $R_c$, in this case, represents the set of constraints that is common to all data points; for example, each location can be visited at most once. Notice that $R_i$ can be also interpreted in the form of constraints, which we will leverage later in the paper. For example, in the routing problem, $R_i$ can be interpreted as the constraint that sequence $S_i$ can only contain locations from the requested set $R_i$, and all locations in $R_i$ must be visited.

Given present contextual data $R_t$, our goal is to find a good sequence of decisions $S_t$. Using machine learning terminology, we would like to learn a conditional probability distribution of "good" sequences

$$Pr(S_t \mid R_t)$$

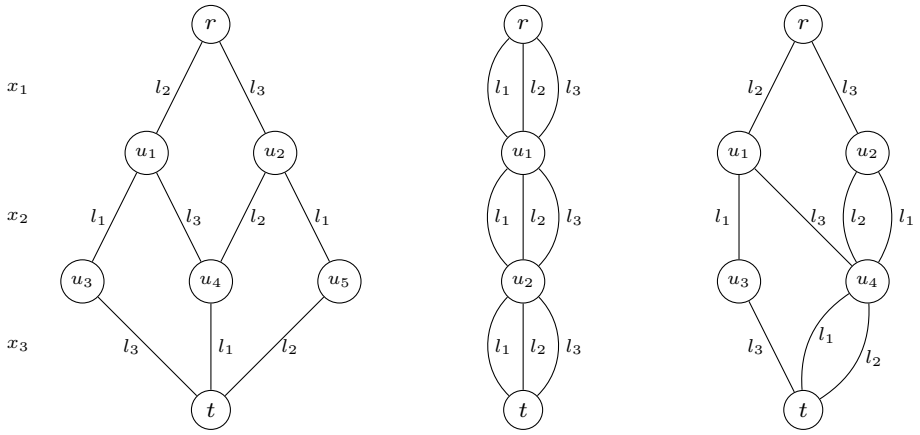
based on the historical dataset. More precisely, $Pr(S_t \mid R_t)$ should assign high probabilities to "good" sequences, i.e., sequences that satisfy operational constraints and look similar to the ones in the historical dataset. A "bad" sequence, for example, a sequence that violates key operational constraints, should ideally be assigned a probability zero. When we deploy our tool (i.e., for scenario $t$), we first observe the contextual data $R_t$, and then sample a sequence $S_t$ according to $Pr(S_t \mid R_t)$, which should satisfy all operational constraints and the implicit preferences reflected by the historical dataset.

**Generative Adversarial Networks (GAN).** GAN is a powerful tool developed in the machine learning community for complex structure generation [15]. The original GAN is developed to learn the probability distribution $Pr(x)$ of complex structural objects $x$. We will briefly review GAN in this simple context.

Given a dataset $X$ in which each entry $x \in X$ is independently and identically drawn from the underlying (unknown) data distribution $Pr_{data}(x)$, GAN fits a probability distribution $Pr(x)$ that best matches $Pr_{data}(x)$. Instead of directly fitting the density function, GAN starts with random variables $z$ with a known prior probability distribution $Pr(z)$ (such as multi-variate Gaussian), and then learns *a deterministic mapping* $G : Z \to X$ in the form of a neural network, which maps every element $z \in Z$ to an element $x \in X$. The goal is to fit the function $G(.)$ so that the distribution of $G(z)$ matches the true data distribution $Pr_{data}(x)$ when $z$ is sampled from the known prior distribution.

GAN also trains another *discriminator* neural network $D : x \to \mathcal{R}$ to determine the closeness of the generated and the true structures. $D$ is trained to separate the real examples from the dataset with the fake examples generated by function $G$. Both the generator $G$ and the discriminator $D$ are trained in a competing manner. The overall objective function for GAN is:

$$\min_G \max_D \quad \mathtt{E}_{x \sim \text{data}}[D(x)] + \mathtt{E}_z[1 - D(G(z))]. \tag{1}$$




a. Exact MDD b. Width-1 relaxed MDD c. Width-2 relaxed MDD

**Fig. 1.** Multi-valued decision diagrams (MDDs) representing possible assignments of $x_1, x_2, x_3$. (a) Exact MDD representing the subset of permutations satisfying $\texttt{alldifferent}(x_1, x_2, x_3)$ and $x_1 \neq l_1$. Each path from $r$ to $t$ represents a valid permutation satisfying the two constraints. (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).

For our application, we extend the classical GAN into a conditional structure, as will be discussed in Section 4. We acknowledge that GAN is a recent popular probabilistic model for structural generation. Nevertheless, structural generation is challenging and many research questions still remain open. Our embedding framework is general and can be applied beyond the GAN structure.

### 3.2 Decision Diagrams

Decision diagrams were originally introduced to compactly represent Boolean functions as a graphical model [2, 9], and have since been widely used, e.g., in the context of verification and configuration problems [31]. More recently decision diagrams have been used successfully as a tool for optimization, by representing the set of solutions to combinatorial optimization problems [6].

Decision diagrams are defined with respect to a sequence of decision variables $x_1, x_2, \ldots, x_n$. Variable $x_i$ has a domain of possible values $D(x_i)$, for $i = 1, \ldots, n$. For our purposes, a decision diagram is a layered directed acyclic graph, with $n+1$ layers of nodes; see Figure 1 for an example. Layer 1 contains a single node, called the root $r$. Layer $n+1$ also contains a single node, called the terminal $t$. 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 $l \in D(x_i)$. For an arc $(v, w)$, we use $var(v, w)$ to represent the variable being assigned through this arc, and use $val(v, w)$ to represent its assigned value. For a node $m$ in the MDD, we use $val(m)$ to represent the union of the values of each

arc starting from node $m$, i.e., $val(m) = \{val(u, v) : u = m\}$. Notice that $val(m)$ represents the possible value assignments of the decision variable corresponding to node $m$. Each path from the root $r$ to the terminal $t$ represents a solution, i.e., a complete variable-value assignment. We can extend the arc definition to allow for "long arcs" that skip layers; a long arc out of a node in layer $i$ still represents a value assignment to variable $x_i$ and assigns the skipped layers to a default value (for example 0). In our case, we consider variables with arbitrary domains, which results in so-called multi-valued decision diagrams (MDDs).

*Example 2.* Let $x_1, x_2, x_3$ represent a sequence of decision variables, each with domain $\{l_1, l_2, l_3\}$. The constraint $\texttt{alldifferent}(x_1, x_2, x_3)$ restricts the values of $x_1, x_2, x_3$ to be all different; i.e., they form a permutation. Along with another constraint $x_1 \neq l_1$, it restricts the set of feasible permutations to be $\{(l_2, l_1, l_3), (l_2, l_3, l_1), (l_3, l_2, l_1), (l_3, l_1, l_2)\}$. Figure 1(a) depicts the exact MDD that encodes all permutations satisfying these two constraints.

**Exact Decision Diagram.** Given a set of constraints $R$, MDD $\mathcal{M}$ is said to be *exact* with respect to $R$ if and only if every path that leads from the root node $r$ to the terminal node $t$ in $\mathcal{M}$ is a variable-value assignment satisfying all constraints in $R$. Conversely, every valid variable-value assignment can be found as a path from $r$ to $t$ in $\mathcal{M}$. For example, Figure 1(a) represents an exact MDD that encodes the constraints $\texttt{alldifferent}(x_1, x_2, x_3)$ and $x_1 \neq l_1$.

**Relaxed Decision Diagram.** Because exact decision diagrams can grow exponentially large, we will also apply *relaxed* decision diagrams of polynomial size [3]. The set of paths in a relaxed decision diagram forms a *superset* of that of an exact decision diagram. For example, the set of paths in Figure 1(a) is fully contained in the sets of paths in the Figures 1(b) and (c). Therefore, the MDDs in Figures 1(b) and (c) form two relaxed versions of the MDD in Figure 1(a). Relaxed MDDs are often defined with respect to a maximum *width*, i.e., the number of nodes in its largest layer. For example, Fig. 1(b) is a width-1 relaxed MDD, which trivially forms the superset of any constrained set of $x_1, x_2, x_3$, while Fig. 1(c) is a width-2 relaxed MDD.

**Decision Diagram Compilation.** Decision diagrams can be constructed to encode constraints over the variables, by a process of node refinement and arc filtering [3, 6]. In general, arc filtering removes arcs that lead to infeasible solutions, while node refinement (or splitting) improves the precision in characterizing the solution space. One can reach an exact MDD by repeatedly going through the filtering and the refinement process from a width-1 MDD. We refer to [10] for details on MDD compilation for sequencing and permutation problems.

**Exact MDD Filtering.** MDD filtering algorithms can also be applied without node refinement, to represent additional constraints in a given MDD. Generally, MDD filtering does not guarantee that each remaining $r$-$t$ path is feasible. To establish that, we next introduce the notion of an exact MDD filter.

**Definition 1.** *Let $\mathcal{M}$ be an exact MDD with respect to a constraint set $R$, and let $C$ be an additional constraint. An MDD filter for $C$ is* exact *if applying it to $\mathcal{M}$ results in an MDD $\mathcal{M}'$ that is exact with respect to $R$ and $C$.*

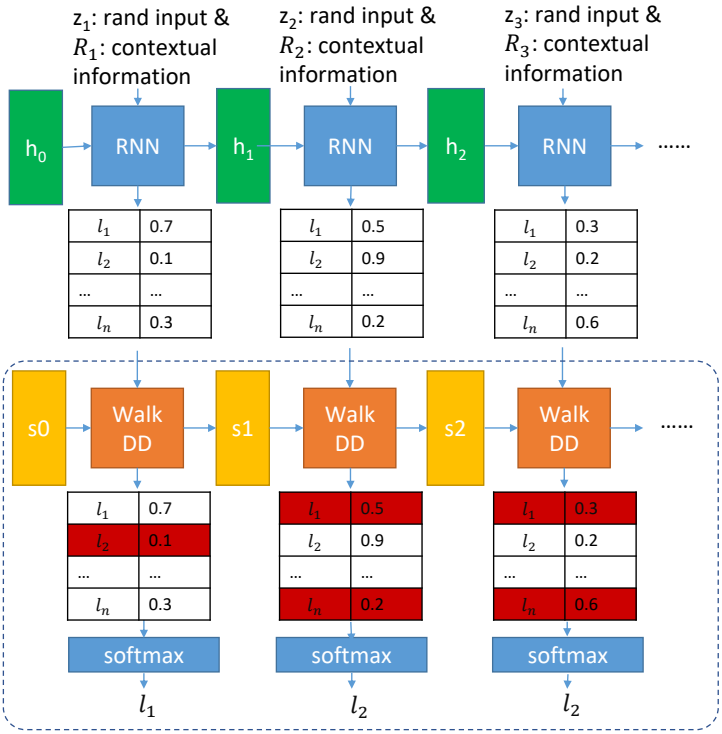



**Fig. 2.** DDGAN as the generator $G$ within the GAN framework. On the top is a recursive neural network (RNN) structure. RNN takes as input the contextual information $R_t$ and random variables $z$, and outputs scores for values for each variable. The values are filtered by the WALKDD module on the bottom that represents a constraint set. The values that lead to contradictions are filtered out (marked by the red color). Finally, a softmax layer decides the value of each variable by picking up the one with the largest score among all non-filtered values.

Consider the following MDD filtering algorithm `FilterUnary` for unary constraints, i.e., constraints of the form $x \neq l$ for some variable $x$ and value $l \in D(x)$. We first let `FilterUnary` remove any arc that violates the unary constraints. Then, working from the last layer up to the first layer, `FilterUnary` recursively removes any nodes and arcs that do not have path that lead to the terminal $t$. We have the following result.

**Proposition 1.** `FilterUnary` *is an exact MDD filter.*

## 4 Embedding Decision Diagrams into GANs

We next present our hybrid approach, DDGAN, which embeds a multi-valued decision diagram into a neural network, to generate structures that (i) satisfy a set of constraints, and (ii) capture the user preferences embedded implicitly in the historical dataset. The structure of DDGAN is shown in Fig. 2.

To achieve this, DDGAN has a recursive neural network (RNN) as its first layer. The RNN module generates scores of possible assignments to variables $x_1, \ldots, x_n$ in a sequence of $n$ steps. We refer the entire recursive neural network (the upper part of Fig.2) as a RNN and refer to the one-step unrolling of the network as a RNN cell. In the $i$-th step, one RNN cell takes the hidden state of the previous step as input, outputs the hidden state of this step, and a table of dimension $|D(x_i)|$. The table corresponds to the score for each value of variable $x_i$. In general, the higher the score is, the more likely the RNN believes that $x_i$ should be assigned to the particular value. RNN is trained to capture implicit preferences which give higher scores to the structures in the historical dataset. Because of the link through hidden states among RNN cells of different steps, RNN is able to capture the correlations among variables.

However, the structures generated by the RNN module may not satisfy key (operational) constraints. Therefore, we embed a WALKDD neural network as a second layer (Fig. 2 bottom, within the dashed rectangular) to filter out actions that violate the constraint set. WALKDD simulates the process of descending along a particular path of the MDD. During this process, WALKDD marks certain assignments generated by the RNN module as infeasible ones (as shown in the entries with red background). Finally, a softmax layer takes the action with highest score among all feasible actions. In this way, WALKDD filters out structures that violate operational constraints.

**Full WalkDD Filtering.** We assume access to an MDD $\mathcal{M}$, which is compiled with respect to the common constraint set $R_c$. We first focus on the case where $\mathcal{M}$ is exact, i.e., each $r$-$t$ path in $\mathcal{M}$ represents an assignment to $x_1, \ldots, x_n$ that satisfies all constraints in $R_c$. We also assume we are given an filtering scheme `Filter` for the additional contextual constraints $R_t$ for data point $t$. Let $\mathcal{M}_t$ be the MDD resulted from $\mathcal{M}$ filtered through constraints $R_t$ using `Filter`.

WALKDD is executed as follows. It keeps the current MDD node of $\mathcal{M}_t$ as its internal state. Initially, the internal state is at the root node $r$. In each step, WALKDD moves to a new MDD node in the next layer once one variable is set to a particular value. Suppose WALKDD is at step $i$, the MDD node $m_i$, and the corresponding decision variable $x_i$. Recall that $val(m_i)$ represents the set of values that can be assigned to variable $x_i$ according to the labels of the arcs out of $m_i$. WALKDD blocks all assignments to $x_i$ outside of $val(m_i)$ (shown as the entries with the red background in Figure 2). After this step, a softmax layer picks up the variable assignment with the largest score among all non-blocked entries and set $x_i$ to be the corresponding value. WALKDD then moves its state to the corresponding new node in the MDD following the variable assignment.

**Proposition 2.** *Let $\mathcal{M}$ be an exact MDD with respect to constraint set $R_c$ and let* `Filter` *be an exact filter for constraint set $R_t$. Then* WALKDD *is guaranteed to produce sequences that satisfy the constraints in both $R_c$ and $R_t$.*

**Overall Conditional GAN Architecture.** During training, the aforementioned DDGAN structure is fed as the generator function $G$ in the conditional GAN architecture [27], which is the classical GAN network, modified to take into

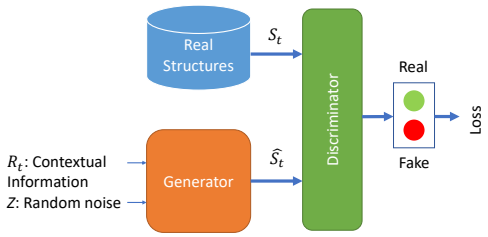



**Fig. 3.** GAN structure for solving the structure generation problem with implicit preferences. The contexts $R_t$ and random variables are fed into the generator $G$, which will produce a structure $S_t$. The Discriminator $D$ are trained to separate the generated structures $\hat{S}_t$ with the real structures $S_t$. $D$ and $G$ are trained in a competing manner.

account contextual information $R_t$. We use conditional GAN as a example to host DDGAN. Nevertheless, DDGAN can be accommodated by other structures as well. For example, infeasible actions can be filtered out by WALKDD in a similar fashion in the Viterbi algorithm for Hidden Markov Models. The overall conditional GAN architecture is shown in Figure 3. In this structure, $R_t$ and random variables $z$ are arguments of the generator $G$, which in turn produces a structure (or sequence) $\hat{S}_t$. The discriminator $D$ is trained to separate the generated structure $\hat{S}_t$ with the real one $S_t$. $D$ and $G$ are trained in a competing manner using stochastic gradient descent. The overall objective function we optimize is:

$$\min_G \max_D \quad \mathrm{E}_t \left[ D(S_t, R_t) + \mathrm{E}_z [1 - D(G(R_t, z), R_t)] \right]. \tag{2}$$

Note that, compared to the objective function of the classical GAN (Equation 1), the discriminator and the generator in Equation (2) both take the contextual information $R_t$ as an additional input.

**Implementation**. WALKDD heavily uses matrix operations, most of which can be efficiently carried out on GPUs. We have a state-transition matrix of the MDD, which are hard-coded prior to training and in which infeasible transitions are labeled with a unique symbol. During execution, we maintain the *state* tensor, which contains the current MDD node of each data point in the mini-batch. We also maintain the *mask* tensor, which indicates values that variables cannot take. **Backpropagate the Gradients.** We heavily rely on non-differentiable *gather* and *scatter* operations offered by Pytorch to maintain the state and the mask tensors. As a filter, WALKDD can have non-differentiable components because it is not updated during training. We pass gradients through the non-blocked entries of WALKDD into the fully-differentiable RNN. We leave it as a future research direction to make WALKDD fully differentiable following the work of Neural Turing Machine [16].

## 5 Case Study: Routing with Implicit Preferences

As proof of concept, we apply DDGAN to a routing problem similar to Example 1. We consider a set of $n$ locations $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$. At each day $t$, a service person (an agent) receives the request to visit a subset of locations $R_t \subset \mathcal{L}$. For this work, we assume that the agent can visit at most $M$ locations in a day; i.e., $|R_t| \leq M$ for all $t$. The agent has a starting location $s \in \mathcal{L}$ and an ending location $e \in \mathcal{L}$. When $s = e$, the agents's route is a Hamiltonian circle.

The agent's actual visit schedule for day $t$, $S_t = (s_{1,t}, s_{2,t}, \ldots, s_{|R_t|,t})$, is a permutation (or, an ordered list) of locations in $R_t$. Notice that it is sufficient to specify the schedule fully using a permutation. Other information, such as the earliest time and latest time to visit locations in the schedule, can be inferred from the permutation. For this work, we assume the agent's schedules are subject to the following constraints:

1. **All-different Constraint.** $S_t$ must be a subset of $\mathcal{L}$, in which no location is visited more than once.
2. **Full-coverage Constraint.** $S_t$ must visit all and only the locations in $R_t$.
3. **Total Travel Distance Constraint.** Let $d_{i,j}$ be the length of the shortest path between $l_i$ and $l_j$. The total travel distance for the agent is: $dist_t = d_{s,s_1} + \sum_{i=1}^{|R_t|-1} d_{s_i,s_{i+1}} + d_{s_{|R_t|},e}$. Suppose the agent has a total travel budget $B$. We must have $dist_t \leq B$.

Observe that only the Full-coverage constraint requires the contextual data $R_t$. Therefore, the common constraints $R_c$ contains the All-different and the Total Travel Distance constraint. Lastly, the agent has **implicit preferences**. As a result, the schedule $S_t$ may deviate from the shortest path connecting the locations in $R_t$. Moreover, the agent cannot represent these preferences as a clear objective function. Instead, we are given the travel history $S_1, S_2, \ldots, S_{t-1}$ and the request locations $R_t$ for day $t$. The goal is to find a valid schedule $S_t$, which satisfies all constraints, but also serves his preferences reflected implicitly in the travel history.

**Constructing MDD for the Routing Problem.** For this application, the set of nodes of the MDD $\mathcal{M}$ is partitioned into $M + 2$ layers, representing variables $x_1, \ldots, x_{M+2}$. The first layer contains only one root node, representing the agent at the starting location, $s$. The last layer also contains only one terminal node, representing the agent at the ending location, $e$. The nodes in the layer of variable $x_i$ correspond to the agent making stops at the $i$-th position of the schedule. As initial domain we use $D(x_i) = \mathcal{L}$, i.e., the set if all possible locations. There are two types of arcs. An arc $a = (u, v)$ of the first type is always directed from a source node $u$ in one layer $x_i$ to a target node $v$ in the subsequent layer $x_{i+1}$. Each arc $a$ of the first type in the $i$-th layer is associated with a label $val(a) \in \{l_1, \ldots, l_n\}$, meaning that the agent visits location $l_i$ as the $i$-th stop. The second type of arcs $b$, whose values $val(b)$ are always $e$, connect every node to the terminal node. These arcs are used to allow the agent to travel back to the end location at any time. The terminal node is connected to itself with an

arc of the second type. This allows the agent to stay at the end location for the rest of his day, once arrived.

We follow the procedure in [10] for constructing the (relaxed) MDD for the routing problem, with respect to both the alldifferent and the maximum distance constraint in $R_c$. That is, we start with a width-1 MDD, and then repeatedly apply the filter and the refine operations until the MDD is exact or a fixed width limit has reached. These operations make use of specific state information that is maintained at each node of the MDD. Since we will re-use some of these, we revisit them here. For an MDD node $v$, (i) $All^{\downarrow}(v)$ is the set of locations that *every* path from the root node $r$ to the current node $v$ passes, while (ii) $Some^{\downarrow}(v)$ is the set of locations that *at least* one path from the root node $r$ to the current node $v$ passes. (iii) $All^{\uparrow}(v)$ and (iv) $Some^{\uparrow}(v)$ are defined similarly, except that they consider locations from the current node $v$ to the terminal node $t$. An arc in the MDD of a routing problem corresponds to visiting one location. For an arc $a$, define (v) $st^{\downarrow}(a)$ as the *shortest travel distance from the root*, which is the shortest distance that the location $val(a)$ can be reached along any path from the root $r$. Similarly, define (vi) $st^{\uparrow}(a)$ as the *shortest travel distance from the target*, which is the shortest distance to travel to the target node $t$ from the location $val(a)$ along any path in the MDD. When the MDD is exact, the sum $st^{\downarrow}(a) + st^{\uparrow}(a)$ represents the shortest distance to travel from the root $r$ to target $t$ along any valid path passing arc $a$. When the MDD is relaxed, the computation of $st^{\downarrow}(a)$ and $st^{\uparrow}(a)$ can be along any path, regardless its validity. The sum $st^{\downarrow}(a)+st^{\uparrow}(a)$ therefore becomes the lower bound of the shortest distance. More details on how this information is used for filtering and refinement can be found in [10].

**Full WalkDD Filtering for the Routing Problem.** The daily requests $R_t$ for the routing problem can be translated into the following two constraints in addition to the all-different and maximum distance requirements in $R_c$: (i) only locations in the requested set $R_t$ are allowed to be visited other than the starting and ending locations. (ii) The trip length is exactly $|R_t| + 2$. Notice that these two constraints can be realized by imposing unary constraints to the MDD. To enforce the first constraint, we can remove all first-type arcs in the MDD whose corresponding location is outside of $R_t$. To enforce the second constraint, we remove all second-type arcs which imply the wrong trip length. Because we have access to an exact filter for unary constraints (Proposition 1), the schedules produced by the full WalkDD scheme presented in Section 4 satisfy all constraints, if the MDD is exact with respect to $R_c$ (Proposition 2).

**Local WalkDD Filtering for the Routing Problem.** The full WalkDD filtering scheme requires a pass over the entire MDD for each data point (filtering unary constraints). While this guarantees to produce structures that satisfy all constraints, when the MDD is exact, it is also relatively computationally expensive. We next discuss a more efficient ***local* WalkDD *filtering scheme*** that removes infeasible transitions only from the information that is *local* to the current MDD node. This local scheme is not comprehensive, i.e., local WalkDD filter may generate structures that do not fully satisfy all constraints, even though the MDD is exact. In practice, however, exact MDDs often become too large, in

which case we apply relaxed MDDs for which the guarantees in Proposition 2 no longer hold. In other words, the computational benefit of local filtering often outweighs theoretical guarantees: it only requires to visit *a single path* from the root to the terminal for local filtering, which is substantially cheaper than visiting the entire MDD as in the full case.

The local WalkDD filter rules out actions of the RNN in each step by only examining information that is local to the current MDD node, as follows. As before, we assume that MDD $\mathcal{M}$ represents the common constraints $R_c$. For day $t$, local WalkDD keeps its own internal state: $W_{t,i} = (u_{t,i}, l_{t,i}, V_{t,i}, R_{t,i}, Time_{t,i})$ after deciding the first $i$ locations, where $u_{t,i}$ is the MDD node in $\mathcal{M}$ representing the current state, $l_{t,i}$ is the current location of the agent, $V_{t,i}$ is the set of locations that the agent already visited, $R_{t,i}$ is the set of locations that remains to be visited, and $Time_{t,i}$ is the time elapsed after visiting the first $i$ locations. Local WalkDD applies the following local filters based on its internal state:

- **Next Location Filter**: The location to be visited should follow one of the arcs that starts from the current node $u_{t,i}$. Otherwise, the location is filtered.
- **Locations Visited Filter**: If the location to be visited is in the set of visited locations $V_{t,i}$, then the location is filtered out, except for the end location $e$.
- **Locations to be Visited Filter**: If the location to be visited is not in the set $R_{t,i}$, then the location is filtered out. We guarantee that the end location $e$ is always in $R_{t,i}$.
- **Future Location Set Filter**: Suppose the next location to be visited is $l_{t,i+1}$ and the MDD node following the arc of visiting $l_{t,i+1}$ is $u_{t,i+1}$. If $R_{t,i} \setminus \{l_{t,i+1}\}$ is not a subset of $Some^{\uparrow}(u_{t,i+1})$, then we cannot cover all the locations remaining to be visited following any paths starting from $u_{t,i+1}$. Therefore, $l_{t,i+1}$ should be filtered out.
- **Total Travel Time Filter**: Let $l_{t,i+1}$ be the next location to visit and $st^{\uparrow}(l_{t,i+1})$ be the shortest time to reach the end location $e$ from $l_{t,i+1}$. If $Time_{t,i} + d_{l_{t,i},l_{t,i+1}} + st^{\uparrow}(l_{t,i+1}) > B$, this suggests that no path from $l_{t,i+1}$ to the end location satisfies the total distance constraint. Therefore, $l_{t,i+1}$ should be filtered out.

## 6 Experiments

The purpose of our experiments is to evaluate the performance of the GAN with and without the DD module. We first describe the implementation details of the GAN. We use a LSTM as the RNN module. The dimension of the hidden state of the LSTM in DDGan is set to 100. The dimension of the random input $z$ is 20. During training, DDGan is used as the generator of the conditional GAN infrastructure. The discriminator of the conditional GAN is also a RNN-based classifier, whose hidden dimension is 100. The batch size is set to be 100. We compare our DDGan with the same neural network structure except without the WalkDD module as the baseline. The entire conditional GAN is trained using stochastic gradient descent. The learning rate of both the generator and the discriminator are both set to be 0.01. Every 10 epochs, the performance of

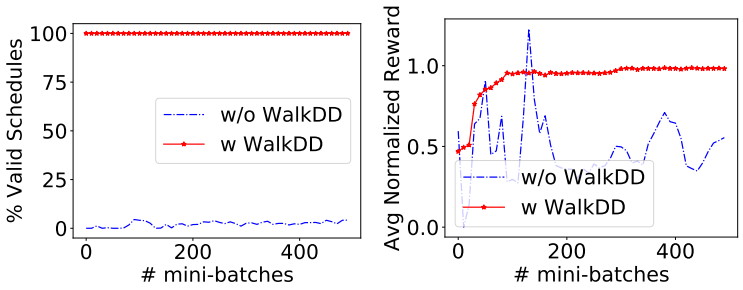



**Fig. 4.** DDGAN on a small scheduling problem with 6 locations. DDGAN has access to an exact MDD and the full filtering scheme. (Left) The percentage of valid schedules generated along training progress. DDGAN always generates valid schedules, while the same neural network without the WALKDD component cannot. (Right) the normalized reward for the schedules generated by DDGAN and the competing approach. The normalized rewards converge to 1 for DDGAN, suggesting that DDGAN is able to fully recover the implicit preference of the agent.

DDGAN and the baselines are tested by feeding in 1,000 scheduling requests into the generator part of the neural network (the structure shown in Figure 2) and examining the schedules it generates.

We assume the agent's implicit preferences are reflected by a hidden reward function $r_{i,j}$, which is the reward that the agent visits location $j$ in the $i$-th position of his schedule. For our experiments, this reward function is generated uniformly randomly between 0 and 1. The agent's optimal schedule is the one that maximizes the total reward while observing all the operational constraints. The reward function is hidden from the neural network. In our application, the goal of the neural networks is to generate schedules subject to operational constraints, which also score high in terms of this hidden reward function.

We first test DDGAN on a synthetic small instance of 6 locations. In this case, we embed an exact MDD into the DDGAN structure and we apply the full filtering scheme as discussed in Section 4. The result is shown in Figure 4. As we can see, the schedules generated by DDGAN always satisfy operational constraints (red curve, left panel), while the schedules generated by the same neural network without the WALKDD module are often not valid (blue curve, left panel). On the right we plot the total reward of the schedules generated by the two approaches. Because the number of locations are small, we compute offline the optimal schedule for each request, i.e., the one that yields the highest total reward. Then we normalize the reward of the generated schedules against that of the optimal schedule, so the optimal schedule should get a reward of 1. As we can see from Figure 4 (right), the normalized reward of the schedules generated by DDGAN converges to 1 as the training proceeds, which suggests that the schedules generated by DDGAN are close to optimal. It is interesting to note that the baseline approach also learns the implicit reward function, since its generated schedules also have high reward. In fact, the normalized reward can go beyond 1 because the schedules do not fully satisfy the constraints.

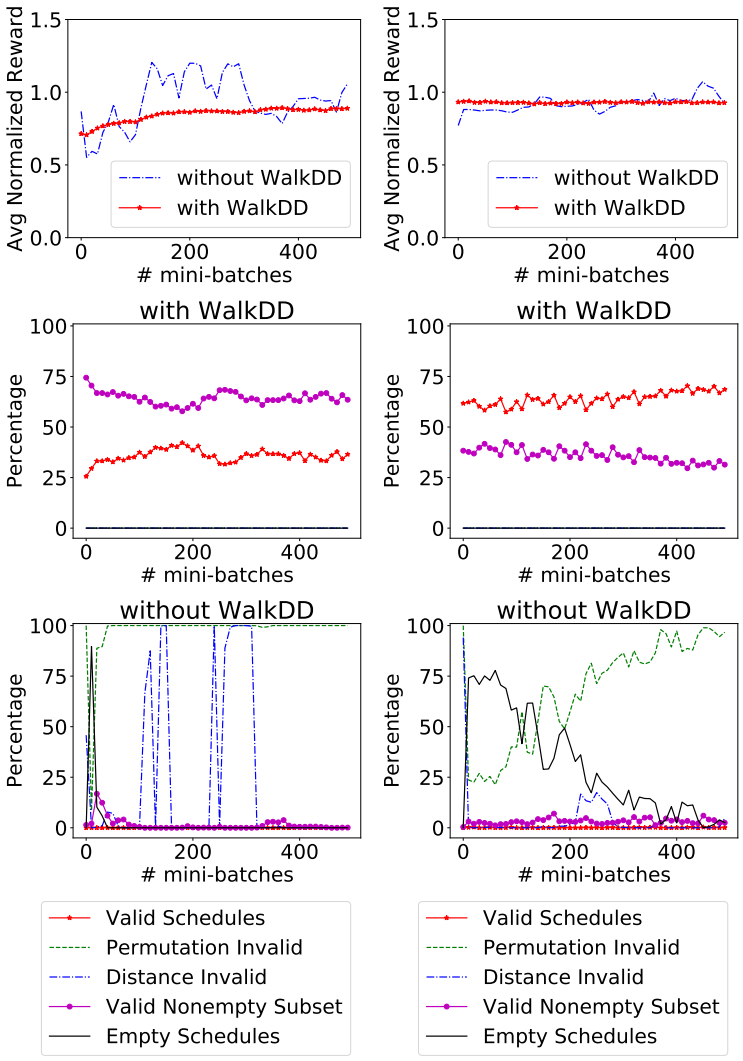



**Fig. 5.** DDGan on a medium-sized scheduling problem in TSPLib consisting of 29 locations, where DDGan can only access a relaxed MDD of width at most 1,000. The agent is allowed to visit 6 locations (left) or 12 locations (right) maximally. (Top) The normalized reward for the schedules generated with DDGan and the same neural network without the WalkDD module. DDGan learns to generate schedules that are close to the optimal ones (with normalized reward close to 1). (Middle, lower) The percentage of different types of schedules generated by DDGan (middle) and the baseline (lower). Legends on the bottom. The schedules generated by DDGan (middle) always satisfy the alldifferent and distance constraints. The percentage of fully valid schedules increase in (middle). However, the same neural network without WalkDD cannot satisfy major constraints (lower).

We then conduct an experiment using instance bayg29.tsp from the TSPLib benchmark, representing 29 cities in Bavaria with geographic distances. We first run a medium-sized experiment, in which the agents can visit at most 6 locations. Even though we do not represent the MDD exactly, we can still compute (offline) the optimal route once we know the requests of at most 6 locations. For problems of this size, we only apply local WalkDD filters, as discussed in Section 5, as they are more efficient. The results are shown in Figure 5 (left). For this experiment, we classify the generated schedules more precisely: (i) **Valid schedules** satisfy all constraints. They cover all the locations in the requested set, meet the travel distance budget, and visit each location exactly once. (ii) **Permutation invalid** schedules visit locations that are outside of the requested set and/or visit locations repeatedly. (iii) **Distance invalid** schedules break the total travel distance constraint. (iv) **Valid non-empty subset** schedules satisfy both the permutation and the distance constraints. However, they visit only a subset of the requested locations. (v) **Empty schedules** do not visit any location other than the starting location. As shown in Figure 5 (middle, left), the schedules generated by DDGan are either completely valid, or violate at most the full coverage constraint. Moreover, the percentage of valid schedules increases as the training proceeds. The schedules generated by DDGan are no longer all valid because the MDD is not exact in dealing with the problem with this scale. On the bottom are the schedules generated with the same neural net without the WalkDD module. As we can see, the schedules break all operational constraints.

In Figure 5 (right), we further run experiments with a maximum of 12 visits. In this case, we cannot compute the optimal schedule exactly. Instead, we use a greedy approach, which selects the best 1,000 candidate solutions for each stop in the schedule. The reward of the schedules generated by the neural networks are normalized with respect to that of the greedy approach. We can see that for larger problems we can apply relaxed decision diagrams of polynomial size that may not guarantee feasibility in all cases, but can produce much better solutions that those generated by the GAN.

## 7 Conclusion

In this work, we study the integration of machine learning and constraint reasoning in the context of sequential decision-making without clear objectives. We propose a hybrid approach, DDGan, which embeds a decision diagram (DD) into a generative adversarial network (GAN). The decision diagram represents the constraint set and serves as a filter for the solutions generated by the GAN, to ensure feasibility. We demonstrate the effectiveness of DDGan to solve routing problems with implicit preferences. We show that without the decision diagram module, the GAN indeed produces sequences that are rarely feasible, while the the decision diagram filter substantially increases the feasibility. Moreover, we show that DDGan converges much more smoothly.

## References


1. Addi, H.A., Bessiere, C., Ezzahir, R., Lazaar, N.: Time-Bounded Query Generator for Constraint Acquisition. In: Proceedings of CPAIOR. LNCS, vol. 10848, pp. 1–17. Springer (2018)
2. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers **C-27**, 509–516 (1978)
3. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A Constraint Store Based on Multivalued Decision Diagrams. In: Proceedings of CP. LNCS, vol. 4741, pp. 118–132. Springer (2007)
4. Beldiceanu, N., Simonis, H.: A Model Seeker: Extracting Global Constraint Models from Positive Examples. In: Proceedings of CP. LNCS, vol. 7514, pp. 141–157. Springer (2012)
5. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. CoRR **abs/1611.09940** (2016), http://arxiv.org/abs/1611.09940
6. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Decision Diagrams for Optimization. Springer (2016)
7. Bessiere, C., Koriche, F., Lazaar, N., OSullivan, B.: Constraint acquisition. Artificial Intelligence **244**, 315–342 (2017)
8. Brock, A., Donahue, J., Simonyan, K.: Large scale gan training for high fidelity natural image synthesis. CoRR **abs/1809.11096** (2018)
9. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35**, 677–691 (1986)
10. Cire, A.A., van Hoeve, W.J.: Multivalued decision diagrams for sequencing problems. Operations Research **61**(6), 1411–1428 (2013)
11. Coletta, R., Bessiere, C., OSullivan, B., Freuder, E.C., OConnell, S., Quinqueton, J.: Constraint Acquisition as Semi-Automatic Modeling. In: Coenen, F., Preece, A., Macintosh, A. (eds.) Research and Development in Intelligent Systems XX. SGAI 2003. pp. 111–124. Springer (2004)
12. Dai, H., Tian, Y., Dai, B., Skiena, S., Song, L.: Syntax-directed variational autoencoder for structured data. CoRR **abs/1802.08786** (2018)
13. Dragone, P., Teso, S., Passerini, A.: Constructive preference elicitation. Front. Robotics and AI **2018** (2018)
14. Galassi, A., Lombardi, M., Mello, P., Milano, M.: Model Agnostic Solution of CSPs via Deep Learning: A Preliminary Study. In: Proceedings of CPAIOR. LNCS, vol. 10848, pp. 254–262. Springer (2018)
15. Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2. pp. 2672–2680. NIPS'14 (2014)
16. Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwinska, A., Colmenarejo, S.G., Grefenstette, E., Ramalho, T., Agapiou, J., Badia, A.P., Hermann, K.M., Zwols, Y., Ostrovski, G., Cain, A., King, H., Summerfield, C., Blunsom, P., Kavukcuoglu, K., Hassabis, D.: Hybrid computing using a neural network with dynamic external memory. Nature **538**(7626), 471–476 (2016)
17. Hu, Z., Yang, Z., Liang, X., Salakhutdinov, R., Xing, E.P.: Toward controlled generation of text. In: Proceedings of the 34th International Conference on Machine Learning. vol. 70, pp. 1587–1596 (2017)

18. Hu, Z., Yang, Z., Salakhutdinov, R., Liang, X., Qin, L., Dong, H., Xing, E.: Deep generative models with learnable knowledge constraints. CoRR **abs/1806.09764** (2018)
19. Jin, W., Barzilay, R., Jaakkola, T.S.: Junction tree variational autoencoder for molecular graph generation. In: ICML (2018)
20. Khalil, E., Dai, H., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. In: Advances in Neural Information Processing Systems, pp. 6348–6358 (2017)
21. Kusner, M.J., Paige, B., Hernández-Lobato, J.M.: Grammar variational autoencoder. In: Proceedings of the 34th International Conference on Machine Learning. vol. 70, pp. 1945–1954 (2017)
22. Lallouet, A., Legtchenko, A.: Building Consistencies for Partially Defined Constraints with Decision Trees and Neural Networks. International Journal on Artificial Intelligence Tools **16**(4), 683–706 (2007)
23. Lallouet, A., Lopez, M., Marti, L., Vrain, C.: On learning constraint problems. In: Proceedings of IJCAI. pp. 45–52 (2010)
24. Lombardi, M., Milano, M.: Boosting Combinatorial Problem Modeling with Machine Learning. In: Proceedings of IJCAI. pp. 5472–5478 (2018)
25. Lombardi, M., Milano, M., Bartolini, A.: Empirical decision model learning. Artif. Intell. **244**, 343–367 (2017)
26. Lombardi, M., Gualandi, S.: A lagrangian propagator for artificial neural networks in constraint programming. Constraints **21**(4), 435–462 (2016)
27. Mirza, M., Osindero, S.: Conditional generative adversarial nets. CoRR **abs/1411.1784** (2014)
28. Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks. CoRR **abs/1511.06434** (2015)
29. Teso, S., Sebastiani, R., Passerini, A.: Structured learning modulo theories. Artif. Intell. **244**, 166–187 (2017)
30. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (eds.) Advances in Neural Information Processing Systems, pp. 2692–2700 (2015)
31. Wegener, I.: Branching programs and binary decision diagrams: theory and applications. SIAM monographs on discrete mathematics and applications, Society for Industrial and Applied Mathematics (2000)