Knowledge Compilation for Constrained Combinatorial Action Spaces in Reinforcement Learning

Jiajing Ling*
Singapore Management University
jjling.2018@smu.edu.sg

Moritz Lukas Schuler*
Singapore Management University
mschuler@smu.edu.sg

Akshat Kumar
Singapore Management University
akshatkumar@smu.edu.sg

Pradeep Varakantham
Singapore Management University
pradeepv@smu.edu.sg

ABSTRACT

Action-constrained reinforcement learning (ACRL), where any action taken in a state must satisfy given constraints, has several practical applications such as resource allocation in supply-demand matching, and path planning among others. A key challenge is to enforce constraints when the action space is discrete and combinatorial. To address this, *first*, we assume an action is represented using propositional variables, and action constraints are represented using Boolean functions. *Second*, we compactly encode the set of *all valid actions* that satisfy action constraints using a probabilistic sentential decision diagram (PSDD), a recently proposed knowledge compilation framework. Parameters of the PSDD compactly encode the probability distribution over all *valid* actions. Consequently, the learning task becomes optimizing PSDD parameters to maximize the RL objective. *Third,* we show how to embed the PSDD parameters using deep neural networks, and optimize them using a deep Q-learning based algorithm. By design, our approach is guaranteed to never violate any constraint, and does not involve any expensive projection step over the constraint space. Finally, we show how practical resource allocation constraints can be encoded using a PSDD. Empirically, our approach works better than previous ACRL methods, which often violate constraints, and are not scalable as they involve computationally expensive projection-over-constraints step.

KEYWORDS

Action-constrained RL; Neuro-symbolic AI

ACM Reference Format:
Jiajing Ling*, Moritz Lukas Schuler*, Akshat Kumar, and Pradeep Varakantham. 2023. Knowledge Compilation for Constrained Combinatorial Action Spaces in Reinforcement Learning. In *Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), London, United Kingdom, May 29 – June 2, 2023*, IFAAMAS, 9 pages.

1 INTRODUCTION

Constrained Markov decision process and constrained reinforcement learning (RL) [1] have become an active and emerging research field, and are widely used to solve safety-critical or resource-related decision making problems. For example, to ensure robots' safety when they are conducting tasks, robots' motion must be restricted in a certain range [14, 26] . Another example is resource allocation in supply-demand matching [6, 32]. The allocation of resources must satisfy some constraints (e.g., total assigned resources should be within a limit). In constrained RL, the goal for the agent is to find a policy that maximizes the expectation of cumulative rewards under constraints that restrict the agent's actions. There are two popular types of constraints—cumulative cost constraints and action constraints. Cumulative cost constraint considers an entire episode, and requires the discounted cumulative cost or the average of costs to be within a threshold [33]. Action constraint is also called an instantaneous constraint. This constraint is imposed on the agent's action in every time step (e.g., robot's motion) [14].

Several recent approaches have been proposed to successfully solve constrained RL with cumulative cost constraints. The most popular method is Lagrangian Relaxation (LR) [13, 24, 33], which converts the constrained RL problem into an unconstrained one [3]. Policy parameters and Lagrangian multipliers are updated iteratively using a gradient-based framework [33].

In solving action-constrained RL (ACRL), LR has also been attempted [7]. However, scalability is challenging since Lagrange multipliers are associated with different state-action pairs and they all need to be optimized, which becomes intractable for large state-action spaces. More importantly, although Lagrange multipliers penalize for constraint violations, this approach cannot guarantee zero constraint violation during training and policy execution.

Since directly applying LR to solve ACRL does not work very well, other approaches have been proposed recently aiming to solve ACRL and achieve zero constraint violation. These approaches mostly focus on continuous action spaces. One natural approach is to add a differentiable projection layer at the end of the policy network [2, 26]. The projection layer projects original actions onto the feasible action space to ensure zero constraint violation. However, there are two main drawbacks of this approach. First, it does not scale up well to problems with large action space since the projection layer typically solves a Quadratic Program (QP) in each RL step, which significantly increases the runtime of RL algorithms. Second, the projection layer could potentially result in the zero-gradient issue during the end-to-end training of a policy network [22]. To tackle the zero-gradient issue, a learning algorithm that decouples policy gradients from action constraints is proposed in [22], where policy parameters are updated by leveraging the Frank-Wolf method [19]. Unfortunately, this approach does not scale up well as it still requires solving a QP during RL training.

*Equal Contribution.

Proc. of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2023), A. Ricci, W. Yeoh, N. Agmon, B. An (eds.), May 29 – June 2, 2023, London, United Kingdom. © 2023 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

When the action space is constrained, discrete, and combinatorial, very few effective algorithms exist for this setting. For a specific type of action constraints, such as resource allocation, there are some methods [6]. However, such methods are not extendable easily to a general constrained combinatorial action space setting. This is precisely the gap that our work addresses by developing a general RL algorithm for constrained combinatorial action spaces.

In constrained combinatorial action spaces, learning a policy is quite challenging since the agent needs to explore the combinatorial action space to find feasible actions through trial and error. One possible approach is to solve a Mixed Integer Quadratic Program (MIQP) in the projection step to obtain feasible actions. However, solving MIQP can be NP-hard [27]. Therefore, it is intractable to apply MIQP in RL since feasible actions need to be computed in each time step. To address this issue, solving QP together with integer rounding as action projection is proposed in [6]. Although solving QP can obtain an approximate solution of the MIQP, rounding the solution to the nearest integers could cause constraint violations. Another recent attempt that is specialized for multiagent path finding (MAPF) problem is to compile all the routes where the agent moves in the underlying graph into a decision diagram [23]. While this works well for MAPF problem, they do not handle the general setting of ACRL, which we target. To summarize, several approaches have been proposed to solve constrained RL with both cumulative constraints and action constraints. However, most of these approaches cannot be applied to solve action-constrained RL with combinatorial action spaces. These approaches either cannot guarantee zero constraint violation, or are not scalable due to the computationally expensive projection step, or are specialized for a particular domain.

Our Contributions: We make the following contributions.

- We consider ACRL with discrete, constrained combinatorial action space. We let an action be represented using a set of propositional variables, and action constraints be represented using Boolean formulas. We leverage *probabilistic sentential decision diagrams* (PSDDs) [21], a knowledge compilation framework, to compactly encode a probability distribution over the set of all the valid actions that satisfy action constraints. A key benefit is that a PSDD can represent a combinatorial, constrained action space compactly.
- We reformulate the task of optimizing a policy directly over constrained combinatorial action space to that of optimizing the parameters of the underlying PSDD, which is significantly smaller than the action space, and thus computationally tractable.
- We show how to encode PSDD parameters using deep neural networks, and optimize them using a deep Q-learning based algorithm for factored actions [16]. We develop new techniques that are needed to integrate optimization of PSDD parameters with deep RL algorithms. By design, our approach is guaranteed to output a feasible action in each RL step *without* involving any computationally expensive projection step over the constraint space. Therefore, it is able to achieve *zero constraint violation* during training and execution.
- We show how practical resource allocation constraints can be encoded using a PSDD, and prove that the compiled PSDD has a polynomial size in the number of resources and entities. Empirically, we evaluate our approach with two previous ACRL methods [6, 22]. Our approach can achieve zero constraint violation, is scalable since computationally expensive projection-over-constraints step is not required during training, and also provides better solution quality in several instances.

2 ACTION-CONSTRAINED RL

A Markov decision process (MDP) model is defined using tuple $(S, A, T, r, \gamma, b_0)$. An agent can be in one of the states $s_t \in S$ at time t. It takes an action $a_t \in A$, receives a reward $r(s_t, a_t)$, and the world transitions stochastically to a new state s_{t+1} with probability $p(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1})$. For the infinite-horizon setting, future rewards are discounted using a factor $0 < \gamma < 1$. The initial state distribution is denoted by $b_0(s)$.

In our work, we consider discrete and combinatorial action space A. As an example, in resource allocation problems where m indistinguishable resources must be assigned to n entities, action space is combinatorial—$|A| = \binom{m+n-1}{n-1}$. We also assume that for any state s, the set of valid actions is given by $C(s) \subseteq A$. We shall discuss the representation of the set $C(s)$ using Boolean constraints later. The goal is to compute the stochastic policy parameterized by θ, $\pi(s;\theta)$, that outputs a probability distribution over the valid actions $\Delta C(s)$ for $s \in S$, and maximizes the total expected rewards $J(\pi)$ as:

$$J(\pi) = \mathbb{E}_{s\sim b_0}\big[V(s;\pi)\big] \tag{1}$$

$$V(s;\pi) = \mathbb{E}\Big[\sum_{t=0}^{\infty}\gamma^t r(s_t, a_t)|s_0 = s;\pi\Big] \tag{2}$$

$$Q(s,a;\pi) = \mathbb{E}\Big[\sum_{t=0}^{\infty}\gamma^t r(s_t, a_t)|s_0 = s, a_0 = a;\pi\Big] \tag{3}$$

where V is the state value function and Q is the state-action value function as per policy π. We assume a reinforcement learning setting where transition and reward functions are not known to the agent. Instead, the agent interacts with the environment simulator and receives observations of the form (s, a, s', r) where (s, a) is the current state and action taken, s' is the next state and r is the reward received.

Action constraints: Let $\boldsymbol{X}_S = \{\beta_1, \ldots, \beta_n\}$ denote the set of n propositional variables defining a state; each $\beta_i \in \{0, 1\}$. Let $\boldsymbol{X}_A = \{\alpha_1, \ldots, \alpha_m\}$ denote the set of m propositional variables defining an action; each $\alpha_i \in \{0, 1\}$. Any state is denoted by an assignment of truth values, s_β, to variables in $\boldsymbol{X}_S$; any action is similarly defined using a_α. Such propositional logic based model representation has also been used for planning in discrete factored state and action spaces [28].

The set of valid actions in a state is defined using K Boolean functions $C_k(\boldsymbol{X}_S, \boldsymbol{X}_A)\ \forall k = 1 : K$. Let $C_k|s_\beta$ represent the reduced Boolean subfunction C_k over variables $\boldsymbol{X}_A$ where $\boldsymbol{X}_S$ variables are set to their respective instantiation in the state s_β. The set of valid actions in a state s_β, $C(s_\beta)$, is given by:

$$C(s_\beta) = \Big\{a_\alpha \;\Big|\; \bigwedge_{k=1}^{K} C_k|s_\beta(a_\alpha) = 1\Big\} \tag{4}$$

We next give some examples showing how action constraints can be represented in a variety of practical problems such as resource allocation and path planning.

Example 2.1 (*Resource allocation constraints*). Consider a resource allocation setting where we need to allocate P resources to Q entities, assuming a 0/1 allocation setting. For simplicity, we assume that there are more entities than resources and each entity can only have at most one resource (this is not a limitation; in our tested domains we can allocate any number of resources to an entity).

We encode the set of all valid allocations using *pseudo-Boolean* (PB) constraints (linear constraints over Boolean variables) [18]. A PB-constraint is an inequality $C_0p_0 + C_1p_1 + \ldots + C_{n-1}p_{n-1} \geq C_n$, where each p_i is a literal and C_i is an integer coefficient. A true literal is denoted as 1 and false as 0. Note that it is easy to incorporate $\leq$ and $=$ both as PB-constraints [18].

We create a Boolean variable X_{pq} denoting whether a resource p is assigned to entity q. The set of PB constraints encoding a valid assignment is denoted as:

$$\sum_{q=1}^{Q}\sum_{p=1}^{P} X_{pq} = P;\ \sum_{q=1}^{Q} X_{pq} = 1\ \forall p = 1:P\ \sum_{p=1}^{P} X_{pq} \leq 1\ \forall q = 1:Q \quad (5)$$

It is shown in [18] how PB-constraints can be translated into Boolean formulas which evaluate to true only if any given literal assignment satisfies the PB-Constraints. Furthermore, it is possible to represent the resulting Boolean functions using popular compact data structures such as binary decision diagrams (BDDs) [9]. We can also have a single BDD representing the conjunction of BDDs for their respective PB-constraints, which encodes the set of all the valid resource allocations.

Example 2.2 (*Multistep path planning*). Consider a multistep path planning problem in a grid introduced in [17]. In their environment, the agent receives a fixed-size square window surrounding its current position as the observation. Consider the action set to be the set of all possible simple paths in the agent's observation, or all possible simple paths of a fixed length. The set of all possible simple paths in a graph is combinatorial, which would make the action space extremely large, and render standard deep RL algorithms impractical. Fortunately, the Boolean formula representing the set of all valid paths can be represented compactly using a data structure called *sentential decision diagrams* (SDDs) [15]. Furthermore, it has been shown that even for large graphs, the size of the SDD representing the set of paths remains tractable by using an approximation scheme [12, 31].

As shown in the above two examples, our goal is to use compact and general data structures such as BDDs and SDDs to implicitly represent the set of all the valid actions without ever explicitly enumerating them. And we also show how different steps in deep RL algorithms can be implemented by using such data structures.

3 PROBABILITY DISTRIBUTION OVER VALID ACTIONS USING DECISION DIAGRAMS

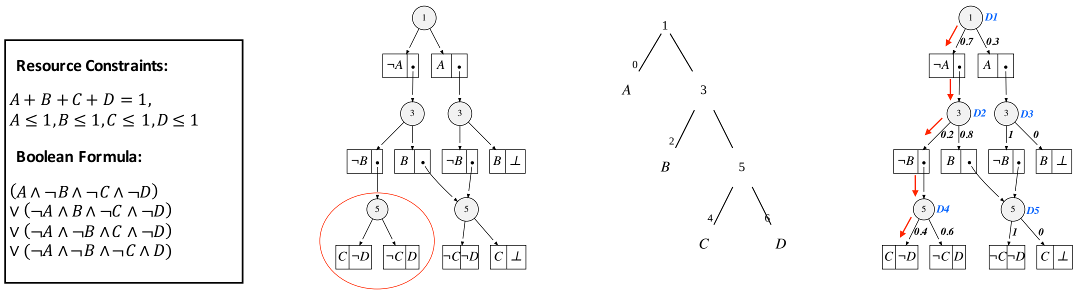
We represent an action constraint $C_k(\boldsymbol{X}_S, \boldsymbol{X}_A)$ using a general data structure called sentential decision diagrams (SDDs) [15]. An SDD is a succinct and tractable representation of a Boolean formula that generalizes the well-known ordered binary decision diagrams (OBDDs) [9]. Succinctness refers to the size of the compiled knowledge representation, and tractability implies that operations such as conjunction (or conjoin) are polytime operations in the size of SDDs [30]. It has also been shown that SDDs can be exponentially more succinct than OBDDs [8], and can also be parameterized to represent a probability distribution over its models (assignments satisfying the Boolean formula represented by the SDD). The latter property is crucial to represent the distribution $\Pr(a|s)$ over valid actions $a \in C(s)$. Our goal is to represent each Boolean function C_k using an SDD. A key benefit of this approach is that we can construct a separate SDD for each $C_k \forall k = 1:K$. Then we can conjoin all such SDDs to represent the single Boolean function (in an SDD form) that represents the set of all valid actions. Such an approach is useful as it allows splitting complex constraints into multiple simpler ones, which can be compactly represented using SDDs. We next describe the SDD structure [15]. An SDD represented as a decision diagram describes members of a combinatorial space (e.g., different valid resource allocations) using propositional logic in a tractable manner.

An SDD represents a Boolean function $f(\boldsymbol{A}, \boldsymbol{B})$ on some non-overlapping variable sets $\boldsymbol{A}$ and $\boldsymbol{B}$ [15]. The function f is represented as $f = (p_1(\mathbf{A}) \wedge s_1(\mathbf{B})) \vee \ldots \vee (p_n(\mathbf{A}) \wedge s_n(\mathbf{B}))$, with each *element* (p_i, s_i), $i = 1 \ldots n$ of the decomposition composed of a *prime* p_i and a *sub* s_i, which themselves are SDDs. A *model* for a given SDD is an instantiation for all the variables such that the corresponding Boolean function f evaluates to true.

An SDD is characterized by a *vtree* for the variable set $\boldsymbol{X} = \boldsymbol{A} \cup \boldsymbol{B}$. The vtree is a *full* binary tree whose leaves are in one-to-one correspondence with variables in $\boldsymbol{X}$. An SDD respecting a vtree T on the variable set $\boldsymbol{X} = \{X_1, X_2, \ldots, \}$ is defined inductively as next. It has two kinds of nodes:

- *terminal node*, which can be a literal (X or $\neg X$), always true ($\top$) or always false ($\bot$), and
- *decision node* having n branches. Decision node represents $(p_1 \wedge s_1) \vee \ldots \vee (p_n \wedge s_n)$ where all (p_i, s_i) are recursively SDDs. The primes for a decision node are always consistent, mutually exclusive and exhaustive [15].

A *vtree* induces a total order on the variables from a left-right traversal of the vtree. E.g., for the vtree in figure 1(c), the variable order is (A, B, C, D). In the SDD shown in figure 1(b), every circular node is a decision node. Its branches are denoted by outgoing arrows denoting corresponding prime and sub (p_i, s_i). Given a fixed vtree, the SDD is unique (or canonical). Every decision node of an SDD is *normalized* for a vtree node v. The number denoted inside each circular node in SDD in figure 1(b) shows the id of the vtree node for which it is normalized for. Intuitively, a decision node n being normalized for vtree node v implies that the Boolean formula encoded by n contains only those variables contained in the sub-tree rooted at v. E.g., for SDD in 1(b), two decision nodes labeled 3 are normalized for node 3 in vtree in figure 1(c), and hence their encoded formulas only contain variables B, C, D. The Boolean formula encoded by the whole SDD is given by the SDD root node.
SDD for resource allocation: Now we show an example of how the constraints in resource allocation are represented using an SDD. Assume there is one resource and four entities. We create four Boolean variables $X_{11}, X_{12}, X_{13}, X_{14}$ denoting whether the resource should be assigned to entity $q, q = 1, \ldots, 4$ or not. To have better viewing of the SDD graph, we rename these four Boolean variables as A, B, C, D respectively. The resource constraints and corresponding Boolean formula are shown in figure 1(a). Given the Boolean



Resource Constraints:
$A+B+C+D=1$,
$A \leq 1, B \leq 1, C \leq 1, D \leq 1$
Boolean Formula:
$(A \wedge \neg B \wedge \neg C \wedge \neg D)$
$\vee (\neg A \wedge B \wedge \neg C \wedge \neg D)$
$\vee (\neg A \wedge \neg B \wedge C \wedge \neg D)$
$\vee (\neg A \wedge \neg B \wedge \neg C \wedge D)$
D1
D2
D3
D4
D5
0.7
0.3
0.2
0.8
1
0
0.4
0.6

(a) (b) (c) (d)

Figure 1: (a) Resource allocation constraints and Boolean formula, (b) SDD **for this constraint, each circular node is a decision node with outgoing arrows denoting its different branches (c) Vtree for the** SDD**, (d) Parameterized** PSDD**. Red arrows denote a sampled action from this** PSDD**. We associate a unique id for each decision node (D1–D5).**

formula, figure 1(b) shows the constructed SDD, figure 1(c) is the vtree for the SDD. The red circle in figure 1(b) is a decision node with two branches: (a) $(C \wedge \neg D)$ with C as prime and $\neg D$ as sub, and (b) $(\neg C \wedge D)$. The formula this decision node represents is $(C \wedge \neg D) \vee (\neg C \wedge D)$. In this example, one model for the SDD is $(\neg A, \neg B, C, \neg D)$, which means the resource is assigned to the third entity. A model can be obtained by traversing the SDD from top to bottom. For each decision node, we select a branch whose sub is not false to visit. Traversal terminates after we have the instantiation for all the variables.

Boolean formulas encoding different action constraints C_k can be compiled into SDDs using the SDD compilers [11, 25], and the open source SDD library is available for such compilation which we used in our experiments [10]. The resulting SDD may not be exponential in size even though it represents an exponential number of objects. We show in Section 6 how resource allocation constraints can be tractably represented using an SDD, even though the total number of valid resource allocations is combinatorial.

3.1 PSDD for distribution over valid actions

Once we conjoin the SDDs for all the action constraints $C_k \forall k=1:K$, we get the final SDD $\mathcal{S}$ for the Boolean formula over (X_S, X_A) encoding all the action constraints. To represent a probability distribution over valid actions, ΔC, so that it can be optimized using an RL algorithm, we need to parameterize the SDD. A *Probabilistic sentential decision diagram* (PSDD) is the parameterized SDD that induces a probability distribution over the *models* of the underlying SDD. That is, a PSDD assigns a strictly positive probability to an instantiation that satisfies the Boolean formula for the underlying SDD; and zero probability to instantiations that do not satisfy the underlying SDD. Thus, by parameterizing the SDD $\mathcal{S}$, we obtain a valid probabilistic distribution only over the valid action set, which is our key objective. We next describe the structure of a PSDD.

If we parameterize each decision node of the SDD, such that the local parameters form a distribution, the resulting probabilistic structure is called a PSDD or a *probabilistic* SDD [21]. It can be used to represent discrete probability distributions $Pr(\mathbf{X})$ where several instantiations $\mathbf{x}$ have zero probability $Pr(\mathbf{x}) = 0$ because of the constraints imposed on the space. More concretely, a PSDD *normalized* for an SDD is defined as follows:

- For each decision node $(p_1, s_1), \ldots, (p_n, s_n)$, there are non negative parameters θ_i such that $\sum_{i=1}^{n} \theta_i = 1$ and $\theta_i = 0$ iff $s_i = \bot$.
- For each terminal node $\top$, there is a parameter $0 < \theta < 1$.

Parameters $\theta_i, i = 1, \ldots, n$ are also called the *local distribution* associated with a decision node. PSDDs are tractable structures of probability distributions as several probabilistic queries can be performed in poly-time such as computing marginal probabilities, conditional probabilities, and sampling from the distribution $Pr(\mathbf{X})$ represented by the PSDD. Given a world state s_β, we can set variables X_S as per β as *evidence*. To obtain the distribution $\Delta C(s_\beta)$ (4), we need to compute the probability of evidence for every node in the PSDD, and renormalize the PSDD parameters based on the probability of evidence [21].

Benefits of PSDD **representation in RL:** There are several significant benefits of using a PSDD for constrained, combinatorial action spaces in RL. *First*, we can decompose complex constraints over valid actions into multiple simpler constraints. We can construct an SDD *independently* for each constraint, and later conjoin such SDDs in a tractable manner to get the final SDD representing all the action constraints [30]. *Second*, we can sample from the distribution ΔC over valid actions by following a simple top-down procedure in the PSDD $\mathcal{S}$ that has complexity linear in the depth of the PSDD [21], which is crucial for fast simulation in an RL environment. *Third*, each generated action sample is *guaranteed* to satisfy all the action constraints by design. This is a major benefit over previous ACRL methods [6, 22] where there is no guarantee that sampled actions will always satisfy constraints, and guaranteeing action constraint satisfaction requires solving expensive mixed-integer math programs for each RL step. *Finally*, the number of parameters in a PSDD is linear in the number of edges in the PSDD [21]. Thus, for compact PSDDs, we can represent the distribution over an exponentially large number of models it encodes using a tractable number of parameters. Therefore, PSDD parameters can also be optimized relatively easily using RL algorithms compared with directly optimizing over the space of combinatorial actions. We also note that the whole SDD-based knowledge compilation can be done offline before the training of the RL agent starts. Thus, during training, there is no overhead involving manipulation of PSDDs other than updating its parameters manipulation of PSDDs other than updating its parameters using the RL algorithm and sampling actions from it. We also show in Section 6 that SDD for common types of resource constraints remains tractable, which shows the usefulness of this method for practical applications.

4 INTEGRATING PSDD IN POLICY NETWORK

Consider the parameterized SDD (PSDD) that encodes all the action constraints. For example, consider the PSDD in figure 1(d). The parameters of this PSDD determine a probability distribution over the models of the underlying SDD, which is precisely the set of all valid actions. Thus, PSDD parameters compactly encode the distribution over all the valid actions. Our key insight is that, as optimizing directly over all valid actions is intractable, we optimize parameters of the PSDD for action constraints, which is tractable given that number of parameters in a PSDD is linear in the number of PSDD edges [21]. For this purpose, we embed PSDD parameters in a deep neural net (which can be later optimized using RL).

We use the PSDD shown in figure 1(d) as an example. Figure 2 shows the neural network policy which has integrated the parameters. The policy network takes the state as input. The input is followed by two fully connected layers with ReLU activation function. Here, we consider fully connected layers for simplicity, in practice, any other kind (and number) of hidden layers can be used. At the end of the policy network, we create different heads for different PSDD decision nodes. Each head has a fully connected layer with *Softmax* activation function, and it outputs the parameters of its corresponding decision node. For example, $D1$ (in figure 1(d)) has two branches, as per PSDD semantics, there are two parameters of $D1$, and the Softmax layer for $D1$ has 2 units in policy network. Each policy network head must output a valid probability distribution since the parameters of a decision node are non negative and sum to 1, and determine a distribution (as noted in Section 3.1). Here, each head is also independent of others given the fact that each decision node has its separate local probability distribution (as noted in Section 3.1). In this example, there are three heads for $D1, D2$, and $D4$ respectively in figure 2. $D3$ and $D5$ only have 2 branches, and the *sub* of their second branch is false (in figure 1(d)). Therefore, the parameter for their second branch is always zero, and we do not have to create a Softmax layer for $D3$ and $D5$.

Sampling valid action from PSDD distribution: Sampling a valid action from the probability distribution encoded by the PSDD can be done using a fast and top-down procedure, which is critical for fast simulation in RL. We take the PSDD for resource allocation constraints and its parameters as shown in figure 1(d) as a running example. We follow the below process for action sampling [21]. We start from the PSDD root node i.e., $D1$. Since $D1$ is a decision node, we sample one of its branches according to its local distribution $(0.7, 0.3)$. Suppose we have selected the first branch, then we get $\neg A$ (which is a terminal node) and decision node $D2$. We conduct the same sampling process for $D2$. Assume we have selected the first branch of $D2$ with probability 0.2, we get $\neg B$ and decision node $D4$. We continue the sampling process for $D4$. Assume that we sample the first branch of $D4$ with probability 0.4,

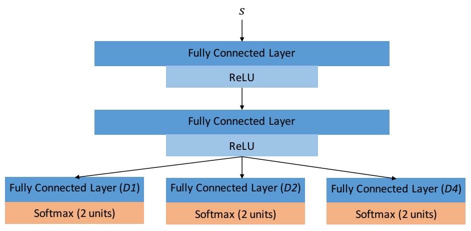


Figure 2: PSDD integrated Policy Network. Each softmax output head is a distribution over different branches of the corresponding PSDD decision node

and get C and $\neg D$. We have now completed the sampling process since we have obtained a model (assignment for all the literals). In this example, the sampled model is $(\neg A, \neg B, C, \neg D)$. To compute the probability of this model, we just need to multiply all parameters that we have encountered during sampling, and we have $Pr(\neg A, \neg B, C, \neg D) = 0.7 \times 0.2 \times 0.4 = 0.056$. The complexity of the sampling is linear in the depth of the PSDD [21], which is significantly cheaper than sampling naively from a tabular distribution over the combinatorial action space.

PROPOSITION 1. *Let the sampled decision branches for all decision nodes from the policy network be denoted using* a_{PSDD}. *There exists a unique environment action* a *corresponding to* a_{PSDD} *which satisfies all the action constraints encoded by the underlying* PSDD.

PROOF SKETCH. The sampled action a_{PSDD} tells which branch to follow for each decision node in the PSDD. Consider the sampling process in Section 4. Starting from the root node of the PSDD, we choose the branch as denoted in a_{PSDD}. This gives us a *unique* path from root to a leaf node in the PSDD. All the literals encountered on this path constitute the unique environment action a corresponding to a_{PSDD}. We also know that $Pr(a) > 0$ (i.e., the action a satisfies all the constraints), as probability of action a is the multiplication of the local probabilities of the chosen branches in a_{PSDD} (as noted earlier, details in [21]), which are non-zero by definition (as they are the output of a Softmax layer). □

5 PUTTING IT ALL TOGETHER: FACTORED ACTION Q-LEARNING

We have shown that the policy network outputs the parameters for each decision node in a PSDD and that sampling decision branch of each decision node gives us a PSDD action a_{PSDD}. Technically, this action cannot be accepted by the RL simulator. However, based on Proposition 1, PSDD action a_{PSDD} can be mapped into a unique environment action a. Environment actions are the actions that can be fed into RL simulator. At each step, given the current state s, we can obtain the parameters of the PSDD from the policy network. Then we sample the PSDD action a_{PSDD} and get the uniquely mapped environment action a. We pass a to the RL simulator and receive a reward r. The environment transitions to a new state s'. The transition sample (s, a, r, s') can be stored in an experience replay buffer, and used to train a critic i.e., the Q function. The critic will guide the update of PSDD parameters using the policy network as in figure 2.

Policy update: Next, we show how to update the PSDD parameters using a recently proposed Q-learning based approach for factored action spaces (named AQL) [16]. The high-level idea is as follows. Given a state s, we first search for the PSDD action a_{PSDD} with the highest Q-value approximately [16], where the Q-value is computed using a_{PSDD}'s corresponding environment action a. As the action space is combinatorial, the AQL algorithm uses two heuristics to sample actions:

- Sample actions a_{PSDD} using the current policy network.
- Sample actions uniformly from the action space. This is challenging as the action space is both constrained and combinatorial. However, this can be addressed easily using the underlying SDD. The open source library [1] provides functionality to sample variables (environment actions in our case) uniformly from a given SDD, which is fast and tractable in the size of the SDD.

From the action samples collected as above, we can get the best action a^*_{PSDD} that has the highest Q-value. We then update the PSDD parameters by maximizing the probability of getting the best action a^*_{PSDD}. To prevent the output distribution of each decision node from becoming deterministic, we also add an entropy regularization. Assume the policy network π which outputs the PSDD parameters is parameterized by θ. The loss function for training the policy is given as follows.

$$\mathcal{L}(\pi_\theta; s) = -\log \pi(a^*_{\text{PSDD}}|s, \theta) - \lambda H(\pi(\cdot|s, \theta)) \tag{6}$$

The first term in the above loss function is the negative log-likelihood with a^*_{PSDD} as the target (similar to the cross-entropy for classification task). By minimizing it, it is more likely to sample a PSDD action from the policy network with the highest Q-value. The second term is the regularization term which makes the policy network output more diverse distributions for decision nodes. λ is a hyperparameter for the regularization term.

Critic update: As we mentioned, the Q-value is computed using the environment action a instead of a_{PSDD} due to the deterministic mapping shown in Proposition 1. We use function g to denote the mapping i.e., $a = g(a_{\text{PSDD}})$. Assume the Q function is parameterized by ϕ, and we have the loss function for updating the Q function as:

$$\mathcal{L}(\phi^Q) = \mathbb{E}_\pi\Big[\big((r_t + \gamma Q(s_{t+1}, a^*_{t+1})) - Q(s_t, a_t; \phi^Q)\big)^2\Big] \tag{7}$$

where $a^*_{t+1} = g(a^*_{\text{PSDD},t+1})$ and $a_t = g(a_{\text{PSDD},t})$.

The original AQL algorithm is designed for large action spaces. However, it is not specialized to handle combinatorial action spaces *with* complex logical constraints. A key benefit of our combination of AQL with PSDD is that we can address rich *constrained* and combinatorial action spaces. Key insights that have helped are to re-interpret PSDD parameters as actions to be optimized by AQL, use Proposition 1 to extract the unique environment action, and modify the policy and critic loss functions appropriately. Finally, we emphasize that AQL algorithm is not the only algorithm that can be used to train the policy network in our case. Most RL algorithms designed for factored action spaces can be applied as the sampled action a_{PSDD} maps deterministically to an environment action a. It also shows the generality of our developed techniques to optimize PSDD parameters.

[1]https://github.com/art-ai/pypsdd

Table 1: Table for Boolean constraint for the setting with total 3 resources and 2 entities. Each entity can get 0-3 resources.

b_{11}	b_{12}	b_{13}	$\top$
b_{12}	b_{13}	b_{21}	$\top$
b_{13}	b_{21}	b_{22}	$\top$
b_{21}	b_{22}	b_{23}	$\top$
$\bot$	$\bot$	$\bot$	

6 COMPACT SDD ENCODING OF RESOURCE CONSTRAINTS

In this section, we describe different resource constraints on integer variables. We also introduce how to translate integer variables to Boolean variables, and constraints to Pseudo-Boolean constraints in a scalable fashion.

Types of resource constraints: Three types of resource constraints we consider in this paper are global, regional, and local constraints, as introduced by [6]. *Global constraints* are also known as equality constraints. They ensure that all available resources m are allocated to the n entities: $\sum_{d=1}^{n} a_d = m$ where $d \in D$ is one of the entities and $a_d \in \mathbb{Z}_{\geq 0}$ represents the number of resources allocated to it. *Regional constraints* require a grouping of entities G, so that constraints can be enforced on each group $G_j \in G$. E.g., the case of a regional minimum bound G_j^{min}: $\sum_{d \in G_j} a_d \geq G_j^{min}$. *Local constraints* limit the number of resources that can be allocated to a single entity. E.g., an entity has a maximum capacity d^{max}: $a_d \leq d^{max}$. For both local and regional constraints, maximum, minimum, and equality constraints are possible. Other sensible constraints do also exist, such as resource flow constraints. They are not presented here, even though it is possible to represent them using an SDD.

Resource constraints as Boolean constraints: We translate the action representation from an integer vector a to a Boolean representation X_A by creating a Boolean variable b_{di} for every integer $i \in \{1, ..., u_d\}$, where u_d is the highest value a_d can take. We do this for all entities $d \in D$. If we consider a global constraint for m resources and n entities, this will lead to $O(m \times n)$ Boolean variables b_{di} where $d \in \{1, ..., n\}$ and $i \in \{1, ..., m\}$, as $u_d = m$, $\forall d$. A corresponding less than or equal to constraint is $\sum_{d,i} b_{di} \leq m$. Local and regional constraints can be constructed in a similar fashion using analogous Boolean variables.

Translating Boolean constraints into decision diagrams: We show a scalable method to compile these Boolean constraints into a compact SDD. Translating these constraints into Boolean formulas and then compiling these into SDDs using standard packages [10, 29] did not scale; The resulting SDD was extremely large in size. Instead, we developed a way to create SDDs for these Boolean constraints which is inspired by pseudo-Boolean constraints [18]: First, we create a table that represents the Boolean constraint. In the case of a *less than or equal to* constraint, it has $m + 2$ rows and $m * n - m + 1$ columns. An example for $m = 3$ (total resources) and $n = 2$ (total entities) is shown in table 1.

In order to check whether an instantiation of all Boolean variables is valid from this table, we must start in the top left corner. If we want to set the variable in the current cell to $\top$ (true) (i.e., one unit of resource is consumed), we must move one cell downwards.

If we want to set it to $\perp$ (false) (i.e., no resource is consumed), we move to the right. This process is repeated until we reach a cell with $\top$ (true) or $\perp$ (false), which tells us whether the variable assignment we tested is valid. We encode this logic into an SDD structure.

This table representation allows us to efficiently compute the information needed to construct an SDD. For space reasons, we provide steps for the construction of the SDD in supplementary. The total number of decision nodes in this SDD is $(m+1)*(n*m-m)-1$. With this SDD, we can encode $\sum_{i=0}^{n*m-m}\binom{n*m}{m+i}$ models. Thus, in a polynomial-sized SDD (in the number of resources and entities), we can represent combinatorial number of valid resource allocations.
Multiplication of PSDDs: For all the constraints, we create SDDs in this fashion. We then conjoin (or multiply) all such SDDs (which is also a poly-time operation as noted earlier) to encode all the constraints into a single SDD. We parameterize the resulting SDD to obtain the PSDD, which is incorporated into our algorithm.

7 EXPERIMENTS

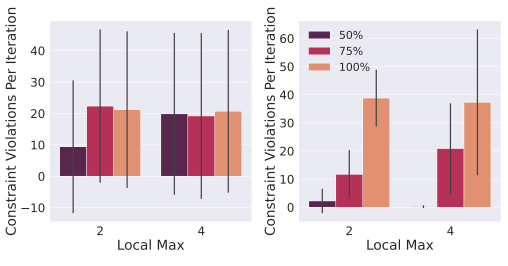
We evaluate our approach on two developed simulation environments for resource allocation: an emergency response system (ERS) [5] and a bike sharing system (BSS) [4]. These environments are used in evaluating the previous proposed resource allocation approaches in [6, 22]. Our code is publicly available in GitHub repository [2].
Emergency response system: This system was originally proposed in [34]. The task here is to provide a target allocation of ambulances to stations in a day. The allocation of ambulances depends on the observation of emergency request demand, the current allocation of ambulances to stations, and the time of the day. This environment also considers adding Gaussian noise (called surge) to the demand request, occurring at a random zone (which a station belongs to) and random time once per day.
Bike sharing system: The bike sharing system was originally introduced in [20]. The task is to provide a target allocation of bicycles to stations to minimize the lost customer demand (no bicycle or no empty parking spot at the station) in a day.
Baselines: We compare our proposed framework KCAC against two previous approaches: DDPG with projection using Quadratic Programming [6] and NFWPO [22] where MIQP is used to enforce constraints on resource allocations. The implementations for both approaches are publicly available. There are two other methods developed in [6]: DDPG with constrained Softmax and DDPG approximate OptLayer. However, these two approaches are restricted in the types of resource constraints they can handle (i.e., they cannot be easily extended to more complicated regional constraints). Therefore, we compare against the most general DDPG with QP-based projection.

7.1 Emergency Response System

In ERS, we consider total 32 ambulances and 25 stations as in [6]. We consider three types of resource constraints as follows.

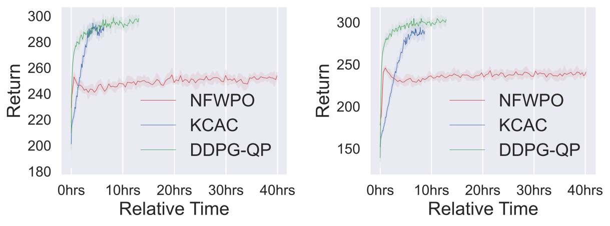
- Global sum: This constraint says the total resource assigned to all stations must be equal to total available resources.
- Local min and max: This constraint specifies the lower bound and upper bound for the resources assigned to a station.

[2]https://github.com/lingkaching/kcac



(a) Non-surge (b) Surge

Figure 3: The number of average constraint violations in DDPG-QP



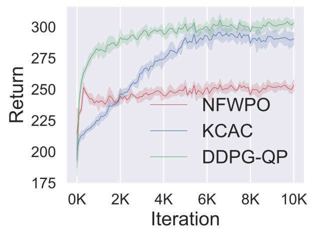
(a) Local max = 2 (b) Local max = 4

Figure 4: Learning process w.r.t run time on ERS with surge

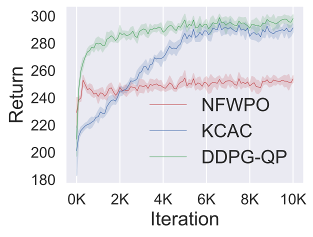
- Group min and max: In this constraint, several stations are grouped together (which is common in reality), and total resource assigned to stations belonging to the same group should be bounded.

In our experiments, the local min is always 0, and local max is 2 or 4 in different instances for all stations. We also generate different group minimal values in a systematic way as follows. We start with assuming all ambulances are uniformly allocated over all stations so that each station will be assigned 1.28 (32/25) ambulances. Then we modify this average assignment with different percentages e.g., 50%, 75%, and 100%, and we get the group min by multiplying the modified average assignment with total number of stations in the group. Intuitively, the higher the percentage is, the tighter the group min constraint is. In the experiments, we do not consider the group max since we assume a group is able to accommodate a large number of ambulances. Considering the combination of these local max and group min constraints, we have total 6 instances e.g, *2-50%*, *4-50%* etc. For each instance, we first generate different PSDDs to encode different constraints separately as described in Section 6. And we multiply all generated PSDDs to obtain a single PSDD which encodes the actions that satisfy all constraints. In the experiments, we keep the neural network architecture of our policy network and critic network the same as the architecture in DDPG-QP except for the last layer in the policy network for our approach (as noted in figure 2). For NFWPO, we use their default neural network architecture (we changed their architecture to the one used in DDPG-QP, however, it performed worst). We also tune the learning rates for all approaches using grid search over $\{10^{-2}, 10^{-3}, 10^{-4}\}$.

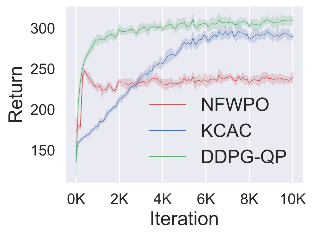
We evaluate all approaches on two ERS scenarios: non-surge and surge, and we run each approach on all instances with 5 different random seeds and report the average. In each run, there are 10k



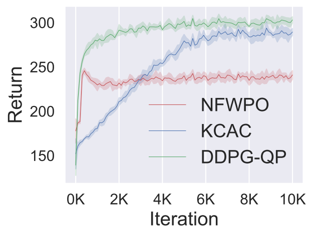
(a) 2-75%



(b) 2-100%

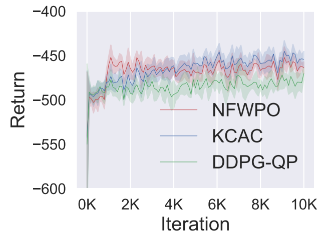


(c) 4-75%

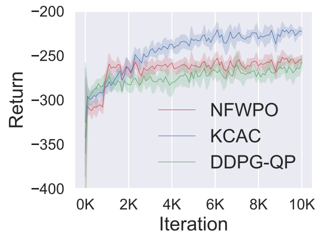


(d) 4-100%

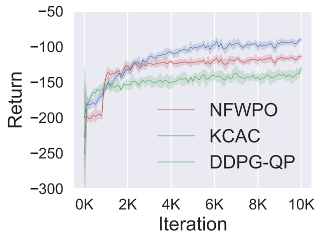
Figure 5: Learning process w.r.t iteration on ERS with surge



(a) 100 bikes



(b) 150 bikes



(c) 200 bikes

Figure 6: Learning process w.r.t iteration on BSS

iterations. Figure 3 shows the number of average constraint violations per iteration during testing using the final trained policy in DDPG-QP. In both figure 3(a) and (b), x-axis denotes the local maximum resource threshold, y-axis denotes the number of constraint violations, and different bars denote different group min threshold values (50%, 75%, 100%). We can see that there are constraint violations in all instances. The number of constraint violations is getting higher when the group min constraints are becoming tighter. When the percentage for group min is 100%, we see the most constraint violations (except for the non-surge, '2' as local max case). The constraint violations are caused by the rounding procedure after solving a QP in each RL step, and it is very hard to control the rounding given the combinatorial action space. In KCAC and NFWPO, there are zero constraint violations during testing. However, zero constraint violation is achieved via different methods. In our approach KCAC, the action is sampled from the PSDD which encodes all valid actions, and it will never violate constraints. However, in NFWPO, MIQP is used to project the action from the policy network to a valid action. Solving a MIQP is NP-hard and is computationally expensive. Therefore, it takes more time in NFWPO to obtain the same solution quality compared with our approach. In figure 4, we show the learning process of all approaches on the most difficult environments for DDPG-QP (surge, and 100% used in computing group min). In both figure 4(a) and (b), x-axis denotes the relative time elapsed during training w.r.t the starting time, and y-axis denotes the return. We can see that it takes 5 times longer for NFWPO to finish the whole 10k iterations compared with our approach KCAC (~40hrs v.s. ~8hrs). To achieve the same solution quality, NFWPO will take more time as well. In other instances for both surge and non-surge cases, NFWPO was 4-5 times slower on average than our method KCAC. Finally, we show the learning process w.r.t. the iteration for all approaches on the surge environment in figure 5. We show results on the non-surge environment in the supplementary as they are very similar. As we can see in figure 5, our approach KCAC and DDPG-QP can achieve very close average returns when convergence occurs. However, there are many constraint violations during testing using the final policy in NFWPO. When compared with NFWPO, our approach achieves better solution quality and runs much faster.

7.2 Bike Sharing System

In BSS experiments, we use the same number of bike stations as in [22] i.e, 5 stations, but we vary the number of bikes e.g, 100, 150 and 200. We consider the global sum constraint and local max constraint in BSS environment. The local max is 23, 35 and 47 for the instance with 100 bikes, 150 bikes and 200 bikes respectively. We evaluate the final trained policy of all three approaches for 1K episodes. In testing, all three approaches can achieve zero constraint violation since the constraints are not complex . However, our approach KCAC performs better in terms of average return. Figure 6 shows the learning process by different methods. Our method provides the best return over both DDPG-QP and NFWPO.

8 CONCLUSION

We have presented a new approach that combined propositional logic based decision diagrams with action-constrained RL for large combinatorial action spaces. Key benefits of our approach are that it can encode complex logical action constraints compactly using sentential decision diagrams and in the context of RL, always guaranteed to provide valid actions without using any expensive projection operation over the constraint space. Empirically, our approach worked much better than previous methods for different resource allocation problems, often providing higher solution quality and is much faster on challenging instances.

ACKNOWLEDGEMENT

This research/project is supported by the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-017). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and DSO National Laboratories.

REFERENCES

[1] E. Altman. 1999. *Constrained Markov Decision Processes.* Chapman and Hall.

[2] Brandon Amos and J Zico Kolter. 2017. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning.* 136–145.

[3] D.P. Bertsekas. 1999. *Nonlinear Programming.* Athena Scientific.

[4] Abhinav Bhatia. 2018. Bike Sharing System (BSS) simualtor gym environment. https://github.com/bhatiaabhinav/gym-BSS

[5] Abhinav Bhatia. 2018. Emergency Response System (ERS) simualtor gym environment. https://github.com/bhatiaabhinav/gym-ERSLE

[6] Abhinav Bhatia, Pradeep Varakantham, and Akshat Kumar. 2019. Resource Constrained Deep Reinforcement Learning. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling.* 610–620.

[7] Steven Bohez, Abbas Abdolmaleki, Michael Neunert, Jonas Buchli, Nicolas Heess, and Raia Hadsell. 2019. Value constrained model-free continuous control. *arXiv preprint arXiv:1902.04623* (2019).

[8] Simone Bova. 2016. SDDs are exponentially more succinct than OBDDs. In *Proceedings of the AAAI Conference on Artificial Intelligence.*

[9] Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.* C-35, 8 (1986), 677–691.

[10] Arthur Choi. 2018. The PyPSDD Package. https://github.com/art-ai/pypsdd

[11] Arthur Choi and Adnan Darwiche. 2013. Dynamic minimization of sentential decision diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 27. 187–194.

[12] Arthur Choi, Yujia Shen, and Adnan Darwiche. 2017. Tractability in structured probability spaces. *Advances in Neural Information Processing Systems* (2017), 3477–3485.

[13] Yinlam Chow, Mohammad Ghavamzadeh, Lucas Janson, and Marco Pavone. 2017. Risk-constrained reinforcement learning with percentile risk criteria. *The Journal of Machine Learning Research* (2017), 6070–6120.

[14] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. 2018. Safe exploration in continuous action spaces. *arXiv preprint arXiv:1801.08757* (2018).

[15] Adnan Darwiche. 2011. SDD: A new canonical representation of propositional knowledge bases. In *International Joint Conference on Artificial Intelligence.* 819–826.

[16] Tom Van de Wiele, David Warde-Farley, Andriy Mnih, and Volodymyr Mnih. 2020. Q-Learning in enormous action spaces via amortized approximate maximization. *CoRR* (2020). arXiv:2001.08116

[17] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. 2015. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679* (2015).

[18] Niklas Een and Niklas Sörensson. 2006. Translating Pseudo-Boolean Constraints into SAT. *JSAT* 2 (03 2006), 1–26.

[19] Marguerite Frank and Philip Wolfe. 1956. An algorithm for quadratic programming. *Naval Research Logistics Quarterly* 3 (1956), 95–110.

[20] Supriyo Ghosh and Pradeep Varakantham. 2017. Incentivizing the Use of Bike Trailers for Dynamic Repositioning in Bike Sharing Systems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 27. 373–381.

[21] Doga Kisa, Guy Van Den Broeck, Arthur Choi, and Adnan Darwiche. 2014. Probabilistic sentential decision diagrams. In *Principles of Knowledge Representation and Reasoning.* 558–567.

[22] Jyun-Li Lin, Wei Hung, Shang-Hsuan Yang, Ping-Chun Hsieh, and Xi Liu. 2021. Escaping from zero gradient: Revisiting action-constrained reinforcement learning via Frank-Wolfe policy optimization. In *Uncertainty in Artificial Intelligence.* 397–407.

[23] Jiajing Ling, Kushagra Chandak, and Akshat Kumar. 2021. Integrating knowledge compilation with reinforcement learning for routes. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 31. 542–550.

[24] Jiajing Ling, Arambam James Singh, Nguyen Duc Thien, and Akshat Kumar. 2022. Constrained multiagent reinforcement learning for large agent population. In *the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases.*

[25] Umut Oztok and Adnan Darwiche. 2015. A Top-down Compiler for Sentential Decision Diagrams. In *International Conference on Artificial Intelligence.* 3141–3148.

[26] Tu-Hoa Pham, Giovanni De Magistris, and Ryuki Tachibana. 2018. Optlayer-practical constrained optimization for deep reinforcement learning in the real world. In *International Conference on Robotics and Automation.* 6236–6243.

[27] Alberto Del Pia, Santanu S. Dey, and Marco Molinaro. 2017. Mixed-Integer Quadratic Programming is in NP. *Math. Program.* 162, 1–2 (2017), 225–240.

[28] Buser Say and Scott Sanner. 2020. Compact and efficient encodings for planning in factored state and action spaces with learned Binarized Neural Network transition models. *AIJ* 285 (2020).

[29] Yujia Shen. 2020. PSDD. https://github.com/hahaXD/psdd

[30] Yujia Shen, Arthur Choi, and Adnan Darwiche. 2016. Tractable Operations for Arithmetic Circuits of Probabilistic Models. In *Advances in Neural Information Processing Systems.*

[31] Yujia Shen, Anchal Goyanka, Adnan Darwiche, and Arthur Choi. 2019. Structured bayesian networks: From inference to learning with routes. In *Proceedings of the AAAI Conference on Artificial Intelligence.* 7957–7965.

[32] Arambam James Singh, Akshat Kumar, and Hoong Chuin Lau. 2020. Hierarchical multiagent reinforcement learning for maritime traffic management. In *the International Conference on Autonomous Agents and Multiagent Systems.*

[33] Chen Tessler, Daniel J Mankowitz, and Shie Mannor. 2018. Reward constrained policy optimization. *arXiv preprint arXiv:1805.11074* (2018).

[34] Yisong Yue, Lavanya Marla, and Ramayya Krishnan. 2012. An Efficient Simulation-Based Approach to Ambulance Fleet Allocation and Dynamic Redeployment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 26. 398–405.