Integrating Knowledge Compilation with Reinforcement Learning for Routes

Jiajing Ling,[*] Kushagra Chandak,[*] Akshat Kumar
School of Computing and Information Systems
Singapore Management University
{jjling.2018, kushagrac, akshatkumar}@smu.edu.sg

Abstract

Sequential multiagent decision-making under partial observability and uncertainty poses several challenges. Although multiagent reinforcement learning (MARL) approaches have increased the scalability, addressing combinatorial domains is still challenging as random exploration by agents is unlikely to generate useful reward signals. We address cooperative multiagent pathfinding under uncertainty and partial observability where agents move from their respective sources to destinations while also satisfying constraints (e.g., visiting landmarks). Our main contributions include: (1) compiling domain knowledge such as underlying graph connectivity and domain constraints into propositional logic based decision diagrams, (2) developing modular techniques to integrate such knowledge with deep MARL algorithms, and (3) developing fast algorithms to query the compiled knowledge for accelerated episode simulation in RL. Empirically, our approach can tractably represent various types of domain constraints, and outperforms previous MARL approaches significantly both in terms of sample complexity and solution quality on a number of instances.

1 Introduction

In cooperative sequential multiagent decision making, agents acting in a partially observable and uncertain environment are required to take coordinated decisions towards a long term goal (Durfee and Zilberstein 2013). Decentralized partially observable MDPs (Dec-POMDPs) provide a rich framework for multiagent planning (Bernstein et al. 2002; Oliehoek and Amato 2016), and are applicable in domains such as vehicle fleet optimization (Nguyen, Kumar, and Lau 2017), cooperative robotics (Amato et al. 2019), and multiplayer video games (Rashid et al. 2018). However, scalability remains a key challenge with even a 2-agent Dec-POMDP, which is NEXP-Hard to solve optimally (Bernstein et al. 2002). To address the challenge of scalability, several frameworks have been introduced that model restricted classes of interactions among agents such as transition independence (Nair et al. 2005), event-driven, and population-based interactions (Becker, Zilberstein, and Lesser 2004; Varakantham et al. 2012). Recently, several multiagent reinforcement learning (MARL) approaches have been developed that push the scalability envelop (Lowe et al. 2017; Rashid et al. 2018) by using simulation-driven optimization of agent policies.

Key limitations of MARL approaches include sample inefficiency, and difficulty in learning when rewards are sparse, which is often the case in combinatorial problems. We address such a combinatorial problem of multiagent path finding (MAPF) under uncertainty and partial observability where agents also need to satisfy domain constraints (noted later) before reaching their destinations. Even the deterministic MAPF setting where multiple agents need to find collision-free paths to their destinations in a shared environment is NP-Hard (Yu and LaValle 2013). The MAPF problem has important applications in several domains such as warehouse logistics (Wurman, D'Andrea, and Mountz 2008), robotics (Sartoretti et al. 2019), and vehicle fleet optimization (Ling, Gupta, and Kumar 2020).

Deep MARL approaches have been applied to MAPF under uncertainty and partial observability (Sartoretti et al. 2019; Ling, Gupta, and Kumar 2020). A key challenge is that it takes several simulations to find even a single route to destination as model-free RL does not explicitly exploit the underlying graph connectivity. Furthermore, agents can move in cycles, specially during initial training episodes, which makes such approaches highly sample inefficient. The problem becomes harder if the agents also need to satisfy constraints such as visiting some landmark nodes before reaching the destination, or coverage constraints such as visiting some locations at least once every k time steps.

To address the challenges of delayed and sparse rewards when learning and planning with routes, we compile the underlying graph-connectivity and other domain constraints using propositional logic based *sentential decision diagrams* (sdd) (Darwiche 2011). An sdd is a succinct and tractable representation of a Boolean formula, and generalizes an ordered binary decision diagram (OBDD) (Bryant 1986). Here succinctness pertains to the size, and tractability refers to the set of polytime operators for the representation. We use sdds over OBDDs as in sdd we can choose over multiple decisions as compared to binary decisions in OBDDs. As a result, sdds tend to be more succinct in practice than OBDDs (Bova 2016). Furthermore, especially for representing routes in a graph, several tractable representations exist for

[*]Equal Contribution

sdds, which we also exploit in our work for scaling to large graphs (Choi, Shen, and Darwiche 2017; Shen et al. 2019).

Our main contributions are:

– We represent the underlying graph connectivity and domain constraints using sdd for the MAPF problem. A key benefit is that any satisfiable instantiation or a *model* of the sdd is a valid simple path (without loops) between the given source and the destination while satisfying all the domain constraints, which significantly prunes the search space, and generates high quality training samples for the learning algorithm.
– We show how to integrate the compiled knowledge in a variety of MARL algorithms based on policy gradient and Q-learning.
– Querying the sdd for sampling RL episodes is too slow using the standard inference methods. Therefore, we also develop efficient inference methods to enable fast sampling of training episodes. Our approaches are general purpose (i.e., can perform query in any sdd, and not just the sdd representing routes), and have linear worst-case complexity in the size of the sdd representation.
– We show how to compile a variety of complex constraints on routes in a simple and modular fashion by using operations such as sdd multiplication. This makes our framework flexible and general purpose for modeling real world applications.

As the number of paths increases exponentially with the graph size, we also use hierarchical decomposition of the graph to enable a tractable sdd representation (Choi, Shen, and Darwiche 2017). We integrate our framework with previous MARL approaches (Sartoretti et al. 2019; Ling, Gupta, and Kumar 2020), and show that the resulting algorithms significantly outperform the original algorithms both in terms of sample complexity and solution quality on a number of instances. We also compare environment simulation speed using our inference algorithm, and previous approaches based on model counting and specialized graph heuristics. Our inference approach is significantly faster in several settings.

2 The Dec-POMDP Model and MAPF

A Dec-POMDP is defined using the tuple $\langle S, A, T, O, Z, r, n, \gamma\rangle$. There are n agents in the environment (indexed using $i = 1 : n$). The environment can be in one of the states $s \in S$. At each time step, agent i chooses an action $a^i \in A$, resulting in the joint action $\boldsymbol{a} \in \boldsymbol{A} \equiv A^n$. As a result of the joint action, the environment transitions to a new state s' with probability $T(s, \boldsymbol{a}, s')$. The joint-reward to the agent team is given as $r(s, \boldsymbol{a})$. The reward discount factor is $\gamma < 1$.

We assume a partially observable setting in which agent i's observation $z^i \in Z$ is generated using the observation function $O(\boldsymbol{a}, s', z^i) = P(z^i|\boldsymbol{a}, s')$ where the last joint action taken was $\boldsymbol{a}$, and the resulting state was s' (for simplicity, we have assumed the observation function is the same for all agents). As a result, different agents can receive different observations from the environment.

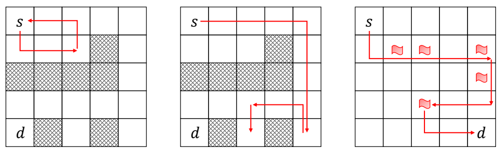
An agent's policy is a mapping from its action-observation history $\tau^i \in (Z\times A)^*$ to actions or $\pi^i(a^i|\tau^i;\theta^i)$, where θ^i parameterizes the policy. Let the discounted future return be denoted by $R_t = \sum_{k=0}^{\infty}\gamma^k r_{k+t}$. The joint-value function induced by the joint-policy of all the agents is denoted as $V^{\boldsymbol{\pi}}(s_t) = \mathbb{E}_{s_{t+1:\infty}, \boldsymbol{a}_{t:\infty}}\big[R_t|s_t, \boldsymbol{a}_t\big]$, and the joint action-value function as $Q^{\boldsymbol{\pi}}(s_t, \boldsymbol{a}_t) = \mathbb{E}_{s_{t+1:\infty}, \boldsymbol{a}_{t+1:\infty}}\big[R_t|s_t, \boldsymbol{a}_t\big]$. The goal is to find the best joint-policy $\boldsymbol{\pi}$ to maximize the value for the starting belief b_0: $V(\boldsymbol{\pi}) = \sum_s b_0(s)V^{\boldsymbol{\pi}}(s)$.

Learning from simulation: In the RL setting, we do not have access to transition and observation functions T, O. Instead, multiagent RL approaches (MARL) learn via interacting with the environment simulator. The simulator, given the joint-action input $\boldsymbol{a}_t$ at time t, provides the next environment state s_{t+1}, generates observation z^i_{t+1} for each agent, and provides the reward signal r_t. Similar to several previous MARL approaches, we assume a centralized learning and decentralized policy execution (Foerster et al. 2018; Lowe et al. 2017). During centralized training, we assume access to extra information (such as environment state, actions of different agents) that help in learning value functions $V^{\boldsymbol{\pi}}$, $Q^{\boldsymbol{\pi}}$. However, during policy execution, agents rely on their local action-observation history. An agent's policy π^i is typically implemented using recurrent neural nets to condition on action-observation history (Hausknecht and Stone 2015). However, our developed results are not affected by a particular implementation of agent policies.

MARL for MAPF: MAPF can be mapped to a Dec-POMDP instance in multiple ways to address different variants (Ma, Kumar, and Koenig 2017; Sartoretti et al. 2019; Ling, Gupta, and Kumar 2020). We present the MAPF problem under uncertainty and partial observability using minimal assumptions to ensure the generality of our knowledge compilation framework. Given an undirected graph $G = (V, E)$, the set V denotes the locations where agents can move, and edges E connect different locations. An agent i has a start vertex s_i and final goal vertex g_i. At any time step, an agent can be located at a vertex $v \in V$, or in-transit on an edge (u, v) (i.e., moving from vertex u to v).

An agent's action set is denoted by $A = A_{\text{mov}} \cup A_{\text{oa}}$. Intuitively, A_{mov} denotes actions that intend to change the location of the agent from the current vertex to a neighboring directly connected vertex in the graph (e.g., move up, right, down, left in a grid graph). The set A_{oa} denotes other actions that do not intend to change the location of the agent (e.g., noop that intends to make the agent stay at the current vertex). We do not make any assumptions regarding the actual transition after taking the action (i.e., move/stay actions may succeed or fail as per the specific MAPF instance).

Depending on the states of all the agents, an agent i receives observation z^i. We assume that an agent is able to fully observe its current location (i.e., the vertex it is currently located at). Other information can also be part of the observation (e.g., locations of agents in the local neighborhood of the agent). We make no specific assumptions about the joint-reward r, other than assuming that an agent prefers to reach its destination as fast as possible if the agent's move-



(a) (b) (c)

Figure 1: (a) Path with a loop; (b) Path to a deadend; (c) Path with landmarks; Dark nodes are blocked. Landmark nodes are flagged.

ment does not conflict with other agents' movements. Similarly, the agent may receive higher reward if it satisfies different types of domain constraints, e.g., reaching the destination after visiting a set of landmark nodes in any order via a simple path, or satisfying coverage constraints, such as visiting some landmark nodes every k time steps. Section 4 contains more details on the constraints types.

Our developed knowledge compilation methods are general and applicable in a variety of multiagent models, e.g., in collective Dec-POMDPs where agents are identical to each other (Nguyen, Kumar, and Lau 2017), or the heterogeneous setting where each agent is unique.

3 Incorporating Compiled Knowledge in RL

A key challenge for RL algorithms for MAPF is that often finding feasible paths to destinations requires a large number of samples. E.g., Figure 1(a) shows the case when an agent loops back to one of its earlier vertex. Figure 1(b) shows another scenario where an agent moves towards a deadend. Such scenarios increase the episode length during training in RL. In Figure 1(c), the agent is required to visit some landmark nodes before reaching the destination. In this case, the agent might not be able to visit all the landmarks in each episode of RL, which would decrease the quality of the samples fed to the learning algorithm. Our goal is to develop techniques that ensure that RL approaches only sample paths that (i) are simple, (ii) satisfy other constraints (like visiting landmarks), and (iii) always originate at the source vertex s_i and end at the goal vertex g_i for any agent i. Each sampled path must satisfy these hard constraints, which is different from standard constrained RL where the expected value of a cost function (over a finite/infinite horizon) is constrained to be less than a threshold (Altman 1999).

Let p_t^i denote the path taken by an agent i until time t (or the sequence of vertices visited by an agent starting from source s_i). We also assume that it does not contain any cycle. This information can be extracted from agent's history τ_t^i. Let $a \in A_{\text{mov}}$ be a movement action towards vertex a_v. We assume the existence of a function $\mathrm{feasibleActions}(\mathrm{p}_t; s_i, d_i)$ that takes as input an agent's current path p_t^i and returns the set $\mathrm{nextActions} = \{a \in A_{\text{mov}}$ s.t. $[\mathrm{p}_t^i, a_v] \rightsquigarrow d_i\}$. The condition $[\mathrm{p}_t^i, a_v] \rightsquigarrow d_i$ implies there exists at least one simple path from source s_i to destination d_i that includes the path segment $[\mathrm{p}_t^i, a_v]$, and satisfies all domain constraints. Thus, starting with $\mathrm{p}_0 = [s_i]$, the RL approach would only sample simple paths that are guaranteed to satisfy the constraints and reach the agent's destination, thereby significantly pruning the search space, and resulting in high reward trajectories. The information required for implementing $\mathrm{feasibleActions}$ can be compiled offline even before training and execution of policy starts (explained in the next section, using decision diagrams), and does not include any communication overhead during policy execution. Using this abstraction, we next present simple and easy-to-implement modifications to a variety of deep multiagent RL algorithms.

Policy gradient based MARL: We first provide a brief background of policy gradient approaches for single agent case (Sutton et al. 2000). An agent's policy π^θ is parameterized using θ. The policy is optimized using gradient ascent on the total expected reward $V(\theta) = \mathbb{E}_{\pi^\theta}[R_0]$. The gradient is given as:

$$\nabla_\theta V(\theta) = \mathbb{E}_{s_{0:\infty}, a_{0:\infty}}\Big[\sum_{t=0}^{\infty} R_t \nabla_\theta \log \pi^\theta(a_t|s_t)\Big] \quad (1)$$

The above gradient expression is also extendible to the multiagent case in an analogous manner (Peshkin et al. 2000; Foerster et al. 2018). The input to policy are some features of the agent's observation history or $\phi(\tau^i)$. The function ϕ can be either hard-coded (e.g., only last two observations), or can be learned using recurrent neural networks.

For using compiled knowledge using the function $\mathrm{feasibleActions}$, the only change we require is in the structure of an agent's policy π (we omit superscript i for brevity). The main challenge is addressing the variable sized output of the policy in a differentiable fashion. Assuming a deep neural net based policy π, given the discrete action space A, the last layer of the policy has $|A|$ outputs using the Softmax. However, when using $\mathrm{feasibleActions}$, the probability of actions not in $\mathrm{feasibleActions}$ needs to be zero. Also, the set $\mathrm{feasibleActions}$ changes as the observation history τ of the agent is updated. However, there is an easy fix similar to proposed in (Buffet and Aberdeen 2009). We use $\tilde{\pi}$ to denote the standard way policy π is constructed with last layer having fixed $|A|$ outputs. The policy π is re-defined as:

$$\pi(a|\tau) = \begin{cases} 0 \text{ if } a \notin \mathrm{feasibleActions}(\mathrm{p}(\tau); s, d) \\ \text{else } \dfrac{\exp\big(\tilde{\pi}(a|\phi(\tau))\big)}{\sum_{a' \in \mathrm{feasibleActions}(\mathrm{p}(\tau); s, d)} \exp\big(\tilde{\pi}(a'|\phi(\tau))\big)} \end{cases}$$

where $\mathrm{p}(\tau)$ denotes the path taken by the agent so far, and s, d are its source and destination. Sampling from π guarantees that invalid actions are not sampled. Furthermore, π is differentiable even when $\mathrm{feasibleActions}$ gives different length outputs at different time steps. The above operation can be easily implemented in autodiff libraries without requiring a major change in the policy structure π.

Q-learning based MARL: Deep Q-learning for the single agent case (Volodymyr et al. 2015) has also been extended to the multiagent setting (Rashid et al. 2018). In the QMIX approach (Rashid et al. 2018), the joint action-value function $Q_{tot}(\boldsymbol{\tau}, \boldsymbol{a}; \psi)$ is factorized as (non-linear) combination of action-value functions $Q_i(\tau^i, a^i; \theta^i)$ of each agent i. A key operation when training different parameters θ^i and ψ involves maximizing $\max_{\boldsymbol{a}} Q_{tot}(\boldsymbol{\tau}, \boldsymbol{a}; \phi)$

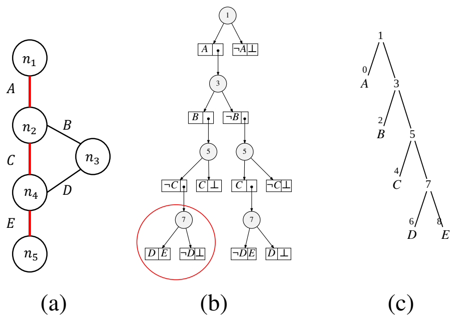


Figure 2: (a) An undirected graph; A, B, C, D, E represent the edge variables. A simple path from $s = n_1$ to $d = n_5$ is highlighted in red and can be written as a propositional sentence $A \wedge C \wedge E \wedge \neg B \wedge \neg D$; (b) An sdd for the graph in (a) where the encircled node represents a decision node $(p_1, s_1), (p_2, s_2)$ (c) a vtree for the sdd

(for details we refer to Rashid et al.). This operation is intractable in general, however, under certain conditions, it can be approximated by maximizing individual Q functions $\max_{a \in A} Q_i(\tau^i, a^i)$ in QMIX. We require two simple changes to incorporate our knowledge compilation scheme in QMIX. First, instead of maximizing over all the actions, we maximize only over feasible actions of an agent as $\max_{a \in \text{feasibleActions}(\text{p}(\tau^i); s^i, d^i)} Q_i(\tau^i, a)$. Second, in Q-learning, typically a replay buffer is also used which stores samples from the environment as $(\boldsymbol{\tau}, \boldsymbol{a}, \boldsymbol{\tau}', r)$. In our case, we also store additionally the set of feasible actions for the next observation history τ'^i for each agent i as $\text{feasibleActions}(\text{p}(\tau'^i); s^i, d^i)$ along with the tuple $(\boldsymbol{\tau}, \boldsymbol{a}, \boldsymbol{\tau}', r)$. The reason is when this tuple is *replayed*, we have to maximize $Q^i(\tau'^i, a)$ over $a \in \text{feasibleActions}(\text{p}(\tau'^i); s^i, d^i)$, and storing the set $\text{feasibleActions}(\text{p}(\tau'^i); s^i, d^i)$ would reduce computation.

We have integrated our knowledge compilation framework with two policy gradient approaches proposed in (Sartoretti et al. 2019; Ling, Gupta, and Kumar 2020) (one using feedforward neural net, another using recurrent neural network based policy), and a QMIX-variant (Fu et al. 2019) for MAPF, demonstrating the generalization power of the framework for a range of MARL solution methods.

4 Compiling and Querying Decision Diagrams for MAPF

We now present our decision diagram based approach to implement the feasibleActions function. Let upper case letters (X) denote variables, and lowercase letters (x) denote their instantiations. Bold upper case letter $(\mathbf{X})$ denotes a set of variables and its lower case counterpart $(\mathbf{x})$ denotes the instantiations.

Paths as a Boolean formula: A path p in the underlying undirected graph $G = (V, E)$ can be represented as a Boolean formula as follows. Consider Boolean random variables $X_{i,j}$ for each edge $(i, j) \in E$. If an edge (i, j) occurs in p, then $X_{i,j}$ is set to true, otherwise set to false. Hence, conjunction of these *literals* denotes path p, and the Boolean formula representing *all paths* is obtained by disjoining formulae for all such paths (Choi, Tavabi, and Darwiche 2016). An example path in a graph is given in Figure 2(a).

Sentential decision diagrams: Since the number of paths between two nodes can be exponential, we need a compact representation of the Boolean formula representing paths. To this end, we use *sentential decision diagram* or sdd (Darwiche 2011). It is a Boolean function $f(\mathbf{X}, \mathbf{Y})$ on some non-overlapping variable sets $\mathbf{X}, \mathbf{Y}$ and is written as a *decomposition* in terms of functions on $\mathbf{X}$ and $\mathbf{Y}$. In particular, $f = (p_1(\mathbf{X}) \wedge s_1(\mathbf{Y})) \vee \ldots \vee (p_k(\mathbf{X}) \wedge s_k(\mathbf{Y}))$, with each *element* (p_i, s_i), $i = 1 \ldots k$ of the decomposition composed of a *prime* p_i and a *sub* s_i. An sdd represented as a decision diagram describes members of a combinatorial space (e.g., paths in a graph) using propositional logic in a tractable manner. 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*, which is represented as $(p_1 \wedge s_1) \vee \ldots \vee (p_k \wedge s_k)$ where all (p_i, s_i) pairs are recursively sdds. The primes are always consistent, mutually exclusive and exhaustive.

Figure 2(b) represents an sdd for the graph in Figure 2(a) encoding all paths from n_1 to n_5. The encircled node is a decision node with two elements (D, E) and $(\neg D, \bot)$. The primes are D and $\neg D$ and subs are E and $\bot$. The Boolean formula representing this sdd node is $(D \wedge E) \vee (\neg D \wedge \bot)$. The Boolean formula encoded by the whole sdd is given by the root node r of the sdd.

An sdd is characterized by a *full* binary tree, called a *vtree*, which induces a total order on the variables from a left-right traversal of the vtree. E.g., for the vtree in Figure 2(c), the variable order is (A, B, C, D, E). Given a fixed vtree, the sdd is unique. An sdd node n is *normalized* (or associated with) for a vtree node v as follows:

- If n is a terminal node, then v is a leaf vtree node which contains the variable of n (if any).
- If n is a decision node, then n's primes (subs) are normalized for the left (right) child of v.
- If n is the root node, then v is the root vtree node.

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. The Boolean formula encoding the domain knowledge can be compiled into a decision diagram using the sdd compiler (Oztok and Darwiche 2015). The resulting sdd may not be exponential in size even though it represents an exponential number of objects.

Satisfiability for feasibleActions: Given an sdd r and evidence $\mathbf{e}$ (an instantiation of some variables), we use f_n to denote a mapping from an sdd node n to the Boolean function it represents. We also assume n is normalized for a vtree node v. We use $\mathbf{e}_v$ to denote the subset of evidence $\mathbf{e}$ that pertains to the variables of the subtree rooted at v. We will use $f_n|\mathbf{e}_v$ to denote the conditioning of f_n on $\mathbf{e}_v$, i.e.,

$f_n|\mathbf{e}_v$ is a *subfunction* resulting from setting variables of f_n to their values in $\mathbf{e}_v$. We define a function $\mathrm{SAT}(f_n|\mathbf{e}_v)$ as:

$$\mathrm{SAT}(f_n|\mathbf{e}_v) = \begin{cases} true & \text{if } f_n|\mathbf{e}_v \text{ is satisfiable,} \\ false & \text{otherwise} \end{cases} \tag{2}$$

We note that $\mathrm{SAT}(f_n|\mathbf{e}_v)$ only answers whether the formula is satisfiable or not; it does not provide a satisfiable instantiation.

We assume that the current sampled path by the agent is p. In the context of sdd, we assume that p is a set of edges in graph G traversed from source s by the agent, and $\boldsymbol{X}_\mathrm{p}$ denotes variables for edges in p. Let v_p denote the current vertex of the agent in G (and assume v_p is not the destination). Let $\mathrm{Nb}(v_\mathrm{p})$ denote all direct neighbors of v_p in G. The feasibleActions set is given as:

$$\mathrm{feasibleActions}(\mathrm{p}) = \\ \{v' \mid v' \in \mathrm{Nb}(v_\mathrm{p}), (v_\mathrm{p}, v') \notin \mathrm{p}, \mathrm{SAT}(f_r|\mathbf{e}_{p'}) = true\} \tag{3}$$

where $\mathbf{e}_{p'}$ is the set of variables for the current partial path and the next action in consideration, i.e., $\mathbf{e}_{p'} = \boldsymbol{X}_\mathrm{p} \cup \{X_{v_\mathrm{p},v'}\}$ ($X_{v_\mathrm{p},v'}$ is set to true when added to the evidence) and f_r is the formula for the sdd root. If $\mathrm{SAT}(f_r|\mathbf{e}_{p'})$ is false, then v' can be pruned from the action set as it implies that there is no simple path to destination d that takes the edge (v_p, v') after taking the partial path p while also satisfying domain constraints.

The above query, as such, can be easily performed for an sdd; we do not require a general purpose Boolean satisfiability solver. The sdd package [1] is equipped with a model counting routine that can also take into account the evidence set. This routine has polytime complexity in the number of sdd nodes. If model count is zero, then we can prune v', otherwise it is feasible. However, this method was still slow with RL, as a query is made to the sdd at each step of episode sampling. To address this, we present a fast algorithm that performs inference in the sdd in a top-down search fashion, and empirically, is much faster than the standard model counting approach. We also highlight that the inference method we present is general purpose and is not limited to sdds that represent paths or any specific constraint.

Algorithms to check sdd satisfiability: Algorithm 1 describes a way to compute $\mathrm{SAT}(f_r|\mathbf{e})$, and is motivated by the sdd model counting approach[1]. Table 1 shows $\mathrm{SAT}(f_n|\mathbf{e}_v)$ for a *terminal* sdd *node* n that is normalized for vtree node v (assume v has the variable X). In this case, $\mathbf{e}_v$ is either a literal (X or $\neg X$) or an empty instantiation $\emptyset$ (which means the variable X is not in the evidence $\mathbf{e}$). Entries in this table can be justified using standard Boolean logic. If X is not in $\mathbf{e}$, then we can set X to either true or false.

When n is a decision node with k elements, $\mathrm{SAT}(f_n|\mathbf{e}_v)$ is computed as $\bigvee_{i=1}^{k} \big(\mathrm{SAT}(f_{p_i}|\mathbf{e}_l) \wedge \mathrm{SAT}(f_{s_i}|\mathbf{e}_r)\big)$, where $\mathbf{e}_l$ and $\mathbf{e}_r$ denote the subsets of evidence $\mathbf{e}$ that pertain to the variables of the left and right subtree of v respectively. We prove the correctness of Algorithm 1 by induction.

Proof. Base Case: *n is a terminal node.* If n is a terminal node, then $\mathrm{SAT}(.|.)$ is given by Table 1.

[1]https://github.com/wannesm/PySDD

n	$\mathrm{SAT}(f_n\|X)$	$\mathrm{SAT}(f_n\|\neg X)$	$\mathrm{SAT}(f_n\|\emptyset)$
$\top$	$true$	$true$	$true$
$\bot$	$false$	$false$	$false$
X	$true$	$false$	$true$
$\neg X$	$false$	$true$	$true$

Table 1: $\mathrm{SAT}(f_n|\mathbf{e}_v)$ for a terminal node given evidence

Inductive Step: *n is a decision node.* Assume it has k elements, and is denoted as $(p_1 \wedge s_1) \vee \ldots \vee (p_k \wedge s_k)$. Let $\mathbf{X}$ and $\mathbf{Y}$ denote variables in the left and the right subtree of v respectively. Notation $\vee_{\mathbf{x}\models\mathbf{e}}$ implies disjunction over all instantiations $\mathbf{x}$ that are consistent with evidence $\mathbf{e}$.

$$\begin{aligned}
\mathrm{SAT}(f_n|\mathbf{e}_v) &= \bigvee_{\mathbf{x}\models\mathbf{e}_l} \bigvee_{\mathbf{y}\models\mathbf{e}_r} \mathrm{SAT}(f_n|\mathbf{xy}) \\
&= \bigvee_{i=1}^{k} \Big(\bigvee_{\mathbf{x}\models\mathbf{e}_l} \bigvee_{\mathbf{y}\models\mathbf{e}_r} \Big(\mathrm{SAT}(f_{p_i}|\mathbf{x}) \bigwedge \mathrm{SAT}(f_{s_i}|\mathbf{y}) \Big) \Big) && (4) \\
&= \bigvee_{i=1}^{k} \Big(\bigvee_{\mathbf{x}\models\mathbf{e}_l} \mathrm{SAT}(f_{p_i}|\mathbf{x}) \Big) \bigwedge \Big(\bigvee_{\mathbf{y}\models\mathbf{e}_r} \mathrm{SAT}(f_{s_i}|\mathbf{y}) \Big) && (5) \\
&= \bigvee_{i=1}^{k} \Big(\mathrm{SAT}(f_{p_i}|\mathbf{e}_l) \wedge \mathrm{SAT}(f_{s_i}|\mathbf{e}_r) \Big) && (6)
\end{aligned}$$

Eq. (4) holds because of the decomposability property of sdd, i.e., variables in prime p_i and sub s_i are disjoint. □

The time complexity of Algorithm 1 is linear in the size of sdd as every node n is visited only once. However, in RL, the inference for feasibleActions needs to be done at each time step for each training episode. We observed empirically that this method was slow, and difficult to scale. We next develop our inference method that implements the same logic as in Algorithm 1, but in a top-down fashion. Our approach does not always need to visit each sdd node, which makes it faster.

In Algorithm 2, for a decision node n (which has elements (p_i, s_i)), we recursively check if $\mathrm{SAT}(f_{p_i}|\mathbf{e}_l) = \mathrm{true}$ and if $\mathrm{SAT}(f_{s_i}|\mathbf{e}_r) = \mathrm{true}$ for all i. As we only need to answer whether the formula $f_n|\mathbf{e}_v$ is satisfiable or not, we can terminate the procedure once we have found that both the conditions are true. We note that we check whether $f_{s_i}|\mathbf{y}$ is satisfiable only if $f_{p_i}|\mathbf{x}$ is satisfiable. We also store values of terminal and decision nodes that we have already visited and reuse them during the recursion. Each edge in sdd will

Algorithm 1: BU-SAT: bottom-up method

```
1 Input: SDD $r$ and evidence $\mathbf{e}$
2 // visit children before parents
3 for node n in the SDD do
4     if n is a terminal node then
5         $n.value \leftarrow \mathrm{SAT}(f_n|\mathbf{e}_v)$
6     else
7         $n.value \leftarrow \bigvee_{i=1}^{k} p_i.value \wedge s_i.value$
8         // $(p_i, s_i)$ is the $i^{th}$ element of node $n$
9 return $r.value$
```

Algorithm 2: TD-SAT: top-down search

```
1  Input: sdd node $n$ and evidence $\mathbf{e}_v$
2  if $n.visited$ is false then
3      if $n$ is a terminal node then
4          $n.value \leftarrow \mathrm{SAT}(f_n|\mathbf{e}_v)$
5      else
6          $n.value \leftarrow false$
7          for element($p_i, s_i$) of node $n$ do
8              if TD-SAT($p_i, \boldsymbol{e}_l$) is true then
9                  if TD-SAT($s_i, \boldsymbol{e}_r$) is true then
10                     $n.value \leftarrow true$
11                     Break
12     $n.visited \leftarrow true$
13 return $n.value$
```

be visited at most once. Hence, Algorithm 2 has linear complexity in the worst case. This simple approach gives us significant improvement in inference speed as compared to Algorithm 1. We also optimize this approach over an episode. That is, assume that the edge (v_{p}, v') is selected at time step t, we can reuse the values of visited nodes during computing $\mathrm{SAT}(f_r|\mathbf{e}_{p'})$ for checking feasibleActions at time step $t+1$ (details omitted). These optimizations are not easily integrable in the bottom-up or model counting approach.

Hierarchical clustering for large graphs: For increasing the scalability of our framework, we take motivation from (Choi, Shen, and Darwiche 2017; Shen et al. 2019). These previous results show that by suitably partitioning the graph G among clusters in a hierarchical way, we can keep the size of the sdd tractable even for very large graphs. Such partitioning does result in the loss of expressiveness as the sdd for the partitioned graph may omit some simple paths, but empirically, we found that this approach worked well. This method is described in the supplementary [2] material.

5 Domain Constraint Modeling

We next show some examples of modeling different kinds of constraints using sdd and operations over sdd.

Simple paths from a source to destination: We have already shown this modeling in Section 4. We also note that a simple graph heuristic based on the shortest path algorithm is also able to find out whether selecting the next node can result in a simple path to destination. However, using sdds, we can model much richer constraints in a tractable manner, and for these constraints, there may not be a straightforward graph heuristic such as shortest path.

Visiting Landmarks Constraint: As in Figure 1(c), consider a problem where an agent moves along a *simple path* from a source to a destination, while visiting some landmarks. Assume there are N landmark nodes numbered $1, \ldots, N$. Let $\mathrm{inc}(k)$ be the set of variables corresponding to the edges incident on landmark k or $\mathrm{inc}(k) = \{X_{i,k} \mid (i,k) \in E\}$. For each landmark k, we can make sure that the agent visits k by making sure *at least* one variable is

[2]https://lingkaching.github.io/pubs/

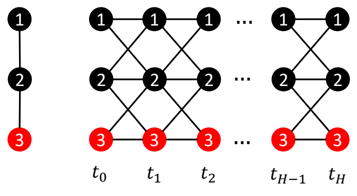


Figure 3: (a) A graph; (b) Time-indexed graph

true in $\mathrm{inc}(k)$. The constraint specifying to specify it is $\mathrm{visit}(k) = \vee_{X \in \mathrm{inc}(k)} X$. We can take conjunction of all such constraints over all landmarks: $\mathrm{visit} = \wedge_{k=1}^{N} \mathrm{visit}(k)$.

Next, we construct an sdd_1 for visit formula. However, this sdd has no information about routes. Therefore, we create another sdd_2 that encodes only simple path constraints from source to destination. We can combine (or multiply) these two sdds so that the resulting sdd has models (satisfiable instantiations) from both the sdds. This can be achieved by the conjoin operation on sdds, and it also has polytime complexity in the size of component sdds (Shen, Choi, and Darwiche 2016). This problem has applications in settings such as taxi pick-up and drop-off (Dietterich 2000) and logistics (Li et al. 2020).

For this problem, it is non-trivial to design an efficient graph heuristic. We implemented one such heuristic from the literature (Vardhan et al. 2009). Empirically, it was much slower than our inference procedure in the compiled sdd. This highlights that for rich constraints, our method provides a generic and fast technique which can be integrated in RL.

Coverage constraint: Consider a simple graph as shown in Figure 3(a) Here the agent's task is to visit node 3 at least once every K time steps over horizon H. To model such a problem, we construct a time-indexed graph as shown in Figure 3(b). In this graph, the agent selects one edge at every time step t. E.g., let the agent take an edge (i, j) between time t and $t+1$. If $i \neq j$, then it moves from node i to j. If $i = j$, then it stays at node i. In this graph, since time flows from left to right, the agent can only move in this direction. To represent this constraint, we focus on the edges between time t and $t+1$. The variables representing these edges are denoted by the set $E(t, t+1)$. Each of these edges can be taken *exactly once* so that the agent does not go backward in time. For each variable $X_i \in E(t, t+1)$, we make sure that only one of them is true using $\mathrm{only}(X_i) = X_i \wedge_{j \neq i} \neg X_j$.

The time constraint between time steps t and $t+1$ is given by: $\mathrm{tc}(t, t+1) = \vee_{X \in E(t,t+1)} \mathrm{only}(X)$. The time constraint over all the time steps is given as $\wedge_{t=0}^{H-1} \mathrm{tc}(t, t+1)$.

To model constraints for visiting node 3 every K time steps, we can follow a strategy similar to landmarks constraint visit between time steps t to $t+K$. Conjunction of all such constraints over the time horizon H gives the final Boolean formula. This problem can be easily generalized to visit N nodes at least once every K time steps. In literature, this is called the coverage problem (Yehoshua and Agmon 2016) and can be leveraged for applications like patrolling (Gupta, Kumar, and Paruchuri 2018, 2019). We can compile separate sdds for all such constraints including an sdd for the simple path constraints, and conjoin them (as in the landmark constraint) to get one sdd that represents all the constraints. In this fashion, we can represent complex constraints in a simple and modular fashion.

6 Empirical Evaluation

We present results to show the succinctness, efficiency and modularity of our knowledge compilation framework.

Number of paths encoded: Table 2 shows that an sdd can encode an exponential number of paths demonstrating its succinctness. For large maps (e.g., 10x10, 20x20), we use hierarchical clustering as noted in Section 4. The table shows the number of paths encoded in maps of sizes 10x10 and 20x20 and with landmark constraints. The largest sdd contained 127K nodes (for 20x20, open grid). We were able to compile sdd with very large number of landmarks (80 landmarks in 20x20 grid), the resulting sdd has $\sim$ 22K nodes, and is thus tractable to represent and reason with.

Simulation Speed: We compare the sampling speeds of TD-SAT, BU-SAT and graph-heuristics based inference approaches on open grid maps (5x5, 10x10, and 20x20) and maps with five landmarks. For grid maps, the graph heuristic GH1 uses Dijkstra's algorithm to determine whether the agent can take the edge next (v_p, v') by checking if there exists a path between v' and the destination d. For maps with landmark constraints, we use an approximation algorithm (GH2) (Vardhan et al. 2009) to check if there exists a simple path from the next possible node v' to destination d while going through unvisited landmark(s).

We randomly generated 10K paths given the source (top right node) and destination (bottom left node) using each approach for both open grid maps and maps with 5 landmarks. We run the simulation for each approach on all instances 5 times. We report the average simulation speed (for a total of 10K paths). Table 3 shows that our inference approach is faster by an order of magnitude than the model counting based approach BU-SAT. The graph heuristic on open grid maps performs well as checking whether the edge (v_p, v') can be taken by Dijkstra's algorithm runs in time $\Theta((|V| + |E|) \log |V|)$. However, we can use hierarchical clustering to divide 10x10 and 20x20 into smaller graphs so that the simulation speed of our approach is comparable to the graph heuristic. For landmarks, Table 4 shows that our approach is the fastest among all approaches. GH2 runs extremely slow on large maps (even with 5 landmarks), and cannot be scaled up to the high density setting.

In Table 5, we show the simulation speeds of BU-SAT and TD-SAT for the coverage problem on a straight line graph with 7 nodes with number of landmarks to visit as $N = 3$. The agent needs to visit all the landmarks every $K = 15$ time steps. The simulations are done for time horizons $H = 30, 60, 90$. We run each simulation 10,000 times and report the average time. Clearly, TD-SAT performs much faster than BU-SAT in all the instances.

Simple path constraint: We next evaluate the integration of our knowledge-compiled framework (KCO) with policy gradient based approaches DCRL (Ling, Gupta, and Kumar 2020) and PRIMAL (Sartoretti et al. 2019), and with Q-learning based approach MAPQN (Fu et al. 2019) on several open grid maps with varying number of agents. For these experiments, we use the simple path constraints using an sdd. We follow the same MAPF model and experimental settings as (Ling, Gupta, and Kumar 2020). More details on the experiments, the neural network structure and the hyperparameters are noted in the supplementary material. We generated 10 instances for each setting. For each instance, we run three times and choose the run with the best performance. The total objective is to minimize sum of costs (SOC) of all agents combined with penalties for congestion. We report the average total objective over all agents vs the average cumulative sample count over all instances on each map.

- Open grid: We evaluate KCO, GH1 with DCRL and MAPQN on open grid maps. Figure 4 clearly shows that both KCO and GH1 make the training process faster and more sample efficient. We note that the difference of the solution quality between KCO and GH1 is not distinct. Moreover, the solution quality is not affected by hierarchical clustering (10x10 case). Based on this observation, we do not show results on the integration of GH1 with RL approaches on maps with obstacles. As noted in (Ling, Gupta, and Kumar 2020), MAPQN cannot work well on large maps with a large number of agents due to the huge state space. Therefore, we do not run MAPQN on grid sizes more than 5x5.

- Obstacles: We evaluate KCO with DCRL, MAPQN and PRIMAL on a 10x10 map with randomly generated obstacles (density 0.35). Figure 6 clearly shows that DCRL

Approach	5x5	10x10	20x20
BU-SAT	979.7	38873.6	730031.1
TD-SAT	153.1	1313.0	26915.3
GH1	**6.9**	**42.9**	**473.4**

Table 3: Simulation speed on open grid maps (in seconds)

Approach	5x5	10x10	20x20
BU-SAT	4513.6	49135.6	1282995.0
TD-SAT	**201.9**	**2061.9**	**43619.2**
GH2	1461.0	85142.9	494469.9

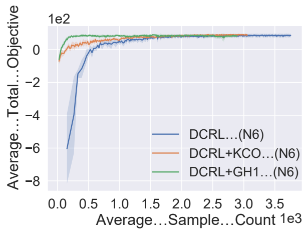
Table 4: Simulation speed on maps with 5 landmarks (in sec)

Size	Open grid	5 Landmarks
10x10	1.08E+13	2.25E+12
20x20	4.59E+51	1.21E+50

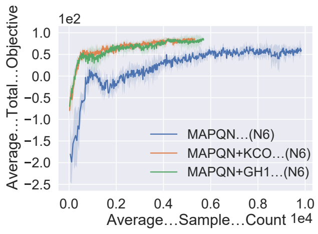
Table 2: Number of paths encoded

Approach	$H = 30$	$H = 60$	$H = 90$
BU-SAT	25764.9	54841.6	80584.0
TD-SAT	**2636.6**	**5483.8**	**8272.3**

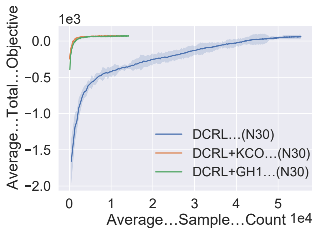
Table 5: Simulation speed for coverage problem (in sec)



(a) 4x4 open grid (DCRL)

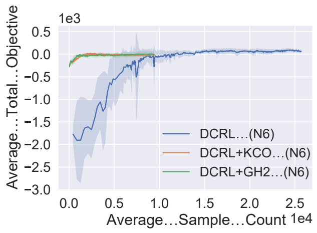


(b) 4x4 open grid (MAPQN)

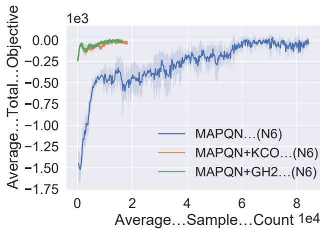


(c) 10x10 open grid (DCRL)

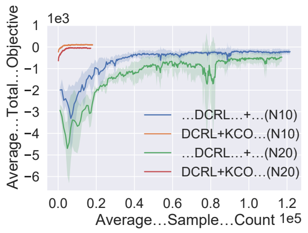
Figure 4: Sample efficiency results on open grids (N# denotes number of agents)



(a) 5x5 L5 (DCRL)

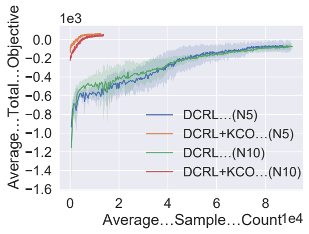


(b) 5x5 L5 (MAPQN)

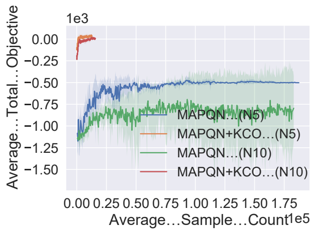


(c) 10x10 L5 (DCRL)

Figure 5: Sample efficiency results on maps with landmarks (L# denotes number of landmarks)

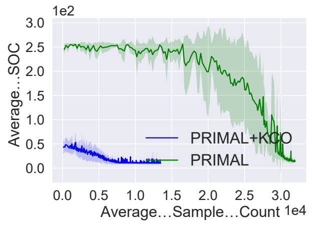


(a) DCRL+KCO

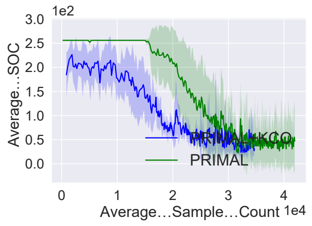


(b) MAPQN+KCO

Figure 6: Efficiency–10x10 grid with obstacles



(a) 2 agents



(b) 4 agents

Figure 7: Efficiency–10x10 grid with obstacles (PRIMAL)

and MAPQN can converge much faster with the integration of KCO. Specifically for MAPQN, several agents did not reach their destinations (8.8 agents on average, for N10 case), whereas in MAPQN+KCO, all agents reached destination, which explains much better solution quality by MAPQN+KCO. As noted in (Sartoretti et al. 2019), high obstacle density is particularly problematic for PRIMAL. Our results in Figure 7 show that PRIMAL+KCO clearly outperforms PRIMAL in terms of sample efficiency. We note that the average SOC is quite high in both PRIMAL and PRIMAL+KCO during the initial episodes (N4 case) as agents take noop actions to avoid congestion.

Simple path + landmark constraints: We evaluate KCO and GH2 based inference approaches with DCRL and MAPQN algorithms on maps with randomly generated landmarks. We do not integrate them with PRIMAL as changing the state space in PRIMAL is challenging. GH2 with RL on large maps runs extremely slow, so we omit those results as well. Figure 5 shows that DCRL and MAPQN converge using much less samples when integrated with KCO and GH2. The results also confirm that an agent needs a lot of exploration to find a path that goes through all the landmarks when there is no integration of compiled knowledge. We also show the results of the setting where there is a large number of agents. This setting is very challenging as all the agents need to visit the same set of landmarks while also avoiding congestion.

7 Conclusion

We addressed the problem of cooperative multiagent pathfinding under uncertainty and partial observability. Our work compiled static domain information such as underlying graph connectivity and constraints over paths using propositional logic based decision diagrams. We developed techniques to integrate such diagrams with deep RL algorithms. For faster RL simulation, we developed a highly efficient inference procedure. We also demonstrated the modeling of different kinds of constraints using propositional logic. Empirically, our approach was highly sample efficient and provided better solution quality than previous best methods.

Acknowledgments

This research is supported by the Ministry of Education, Singapore under its MOE Academic Research Funding Tier 2 (Award MOE2018-T2-1-179). Any opinions, findings, and conclusions are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

References

Altman, E. 1999. *Constrained Markov Decision Processes*. Chapman and Hall.

Amato, C.; Konidaris, G.; Kaelbling, L. P.; and How, J. P. 2019. Modeling and planning with macro-actions in decentralized POMDPs. *JAIR* 64: 817–859.

Becker, R.; Zilberstein, S.; and Lesser, V. 2004. Decentralized Markov decision processes with event-driven interactions. In *AAMAS*, 302–309.

Bernstein, D. S.; Givan, R.; Immerman, N.; and Zilberstein, S. 2002. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research* 27(4): 819–840.

Bova, S. 2016. SDDs are exponentially more succinct than OBDDs. *arXiv preprint arXiv:1601.00501* .

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE TC* 100(8): 677–691.

Buffet, O.; and Aberdeen, D. 2009. The factored policy-gradient planner. *AI* 173(5-6): 722–747.

Choi, A.; Shen, Y.; and Darwiche, A. 2017. Tractability in structured probability spaces. In *NeurIPS*, 3477–3485.

Choi, A.; Tavabi, N.; and Darwiche, A. 2016. Structured Features in Naive Bayes Classification. In *AAAI*, 3233–3240.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, 819–826.

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR* 13: 227–303.

Durfee, E.; and Zilberstein, S. 2013. Multiagent Planning, control, and execution. In Weiss, G., ed., *Multiagent Systems*, chapter 11, 485–546. Cambridge, MA, USA: MIT Press.

Foerster, J. N.; Farquhar, G.; Afouras, T.; Nardelli, N.; and Whiteson, S. 2018. Counterfactual multi-agent policy gradients. In *AAAI*, 2974–2982.

Fu, H.; Tang, H.; Hao, J.; Lei, Z.; Chen, Y.; and Fan, C. 2019. Deep multi-agent reinforcement learning with discrete-continuous hybrid action spaces. In *IJCAI*, 2329–2335.

Gupta, T.; Kumar, A.; and Paruchuri, P. 2018. Planning and Learning for Decentralized MDPs With Event Driven Rewards. In *AAAI*, 6186–6194.

Gupta, T.; Kumar, A.; and Paruchuri, P. 2019. Successor Features Based Multi-Agent RL for Event-Based Decentralized MDPs. In *AAAI*, 6054–6061.

Hausknecht, M.; and Stone, P. 2015. Deep recurrent Q-learning for partially observable MDPs. In *AAAI Fall Symposium - Technical Report*, 29–37.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. K. S.; and Koenig, S. 2020. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. In *AAMAS*, 1898–1900.

Ling, J.; Gupta, T.; and Kumar, A. 2020. Reinforcement Learning for Zone Based Multiagent Pathfinding under Uncertainty. In *ICAPS*, 551–559.

Lowe, R.; Wu, Y.; Tamar, A.; Harb, J.; Abbeel, P.; and Mordatch, I. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *NeurIPS*, 6380–6391.

Ma, H.; Kumar, T. K.; and Koenig, S. 2017. Multi-agent path finding with delay probabilities. In *AAAI*, 3605–3612.

Nair, R.; Varakantham, P.; Tambe, M.; and Yokoo, M. 2005. Networked distributed POMDPs: A synthesis of distributed constraint optimization and POMDPs. In *AAAI*, 133–139.

Nguyen, D. T.; Kumar, A.; and Lau, H. C. 2017. Collective multiagent sequential decision making under uncertainty. In *AAAI*, 3036–3043.

Oliehoek, F. A.; and Amato, C. 2016. *A Concise Introduction to Decentralized POMDPs*. Springer.

Oztok, U.; and Darwiche, A. 2015. A top-down compiler for sentential decision diagrams. In *IJCAI*, 3141–3148.

Peshkin, L.; Kim, K.-E.; Meuleau, N.; and Kaelbling, L. P. 2000. Learning to Cooperate via Policy Search. In *UAI*, 489–496.

Rashid, T.; Samvelyan, M.; de Witt, C. S.; Farquhar, G.; Foerster, J. N.; and Whiteson, S. 2018. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. In *ICML*, 4292–4301.

Sartoretti, G.; Kerr, J.; Shi, Y.; Wagner, G.; Satish Kumar, T. K.; Koenig, S.; and Choset, H. 2019. PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning. *IEEE RA-L* 4(3): 2378–2385.

Shen, Y.; Choi, A.; and Darwiche, A. 2016. Tractable operations for arithmetic circuits of probabilistic models. In *NeurIPS*, 3943–3951.

Shen, Y.; Goyanka, A.; Darwiche, A.; and Choi, A. 2019. Structured bayesian networks: From inference to learning with routes. In *AAAI*, 7957–7965.

Sutton, R. S.; McAllester, D.; Singh, S.; and Mansour, Y. 2000. Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS*, 1057–1063.

Varakantham, P.; Cheng, S. F.; Gordon, G.; and Ahmed, A. 2012. Decision support for agent populations in uncertain and congested environments. In *AAAI*, 1471–1477.

Vardhan, H.; Billenahalli, S.; Huang, W.; Razo, M.; Sivasankaran, A.; Tang, L.; Monti, P.; Tacca, M.; and Fumagalli, A. 2009. Finding a simple path with multiple must-include nodes. In *IEEE MASCOTS*, 1–3.

Volodymyr, M.; Koray, K.; David, S.; Rusu Andrei A; Joel, V.; Bellemare Marc G; Alex, G.; Martin, R.; Fidjeland Andreas K; and Georg, O. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540): 529.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* 29(1): 9–19.

Yehoshua, R.; and Agmon, N. 2016. Multi-robot adversarial coverage. In *ECAI*, 1493–1501.

Yu, J.; and LaValle, S. M. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI*, 1443–1449.