Reinforcement Learning for POMDP: Partitioned Rollout and Policy Iteration With Application to Autonomous Sequential Repair Problems

Sushmita Bhattacharya, Sahil Badyal, Thomas Wheeler, Stephanie Gil, and Dimitri Bertsekas

***Abstract*—In this letter we consider infinite horizon discounted dynamic programming problems with finite state and control spaces, and partial state observations. We discuss an algorithm that uses multistep lookahead, truncated rollout with a known base policy, and a terminal cost function approximation. This algorithm is also used for policy improvement in an approximate policy iteration scheme, where successive policies are approximated by using a neural network classifier. A novel feature of our approach is that it is well suited for distributed computation through an extended belief space formulation and the use of a partitioned architecture, which is trained with multiple neural networks. We apply our methods in simulation to a class of sequential repair problems where a robot inspects and repairs a pipeline with potentially several rupture sites under partial information about the state of the pipeline.**

***Index Terms*—Optimization and optimal control, distributed robot systems, autonomous agents, search and rescue robots, deep learning in robotics and automation.**

I. Introduction

We consider the classical partial observation Markovian decision problem (POMDP) with a finite number of states and controls, and discounted additive cost over an infinite horizon. The optimal solution is typically intractable, and several suboptimal solution/reinforcement learning approaches have been proposed. Amongst these, are point-based value iteration (see e.g., [1]–[3]), approximate policy iteration methods based on the use of finite state controllers (see e.g., [4]–[6]), the use of Monte Carlo tree search and adaptive sampling methods for multistep lookahead with terminal cost function approximation (see e.g., [7], [8]), and discretization/aggregation methods based on the solution of a related perfectly observable Markovian decision problem (see e.g., [9]–[11]). There have also been proposals of policy gradient and related actor-critic methods (see [12]–[14]) that are largely unrelated to methodology proposed here.

In this letter we focus on methods of policy iteration (PI) that are based on rollout, and approximations in policy and value space. They update a policy by using truncated rollout with that policy, and a terminal cost function approximation. Several earlier works include elements of our algorithmic approach. In particular, related methods have been applied with some variations to perfect state information problems, notably in the backgammon algorithm of Tesauro and Galperin [15], and in the AlphaGo program [16], as well as in the classification-based PI algorithm of Lagoudakis and Parr [17] (see also [18]–[20]). The AlphaZero program also involves a similar approximation architecture, but in its published description [21], it does not use rollout, likely because the use of long multistep lookahead in conjunction with Monte Carlo tree search makes the use of rollout unnecessary. A further novel feature of our algorithms is the use of a partitioned architecture, involving multiple policy and value neural networks, which is well-suited for distributed implementation. Partitioning in conjunction with asynchronous PI was originally proposed by Bertsekas and Yu [22], and further developed in the papers [23] and [24] [see also the books [25] (Section 2.6) and [26] (Section 2.6) for descriptions, analysis, and extensions]. However, most of this research was focused on the case of perfect state information, and lookup table representations of policies and cost functions. Thus, while most of the principal elements of our approach have individually appeared in various forms in earlier perfect state information algorithmic frameworks, their combination has not been brought together into a single algorithm, and moreover they have not been adapted to the special challenges posed by POMDP.

Because of its simulation-based rollout character, the methodology of this letter depends critically on the finiteness of the control space, but it does not rely on the piecewise linear structure of the finite horizon optimal cost function of POMDP. It can be extended to POMDP with infinite state space but finite control space, although we have not considered this possibility in this letter. We describe error bounds to guide the implementation of our algorithms, and we provide results of computational experimentation showing that our methods are viable and well-suited to the POMDP structure.

We apply our methods to a class of problems in robotics, involving sequential repairs, and search and rescue, where the

Manuscript received September 10, 2019; accepted January 23, 2020. Date of publication March 4, 2020; date of current version April 21, 2020. This letter was recommended for publication by Associate Editor G. Neumann and Editor T. Asfour upon evaluation of the reviewers' comments. This work was supported by the National Science Foundation CAREER Award under Grant 1845225. *(Corresponding author: Stephanie Gil.)*

Sushmita Bhattacharya, Sahil Badyal, and Thomas Wheeler are with the REACT Lab, Arizona State University, Tempe, AZ 85287 USA (e-mail: sbhatt55@asu.edu; sbadyal@asu.edu; thomassw66@gmail.com).

Stephanie Gil is with the REACT Lab, Arizona State University, Tempe, AZ 85287 USA, and an Assistant Professor of Computer Science, Arizona State University, 699 S Mill Avenue, Tempe, AZ 85287 USA (e-mail: sgil@asu.edu).

Dimitri Bertsekas is with the McAfee Professor of Engineering, Massachusetts Institute of Technology, 77 Mass Ave, Cambridge, MA 02139 USA, and a Fulton Professor of Computational Decision Making, Arizona State University, 699 S Mill Avenue, Tempe, AZ 85287 USA (e-mail: dimitrib@mit.edu).

Digital Object Identifier 10.1109/LRA.2020.2978451

POMDP model is particularly well-suited to deal with partial state information. Autonomous robots in search and rescue have been viewed as one of the robotics applications where POMDP approaches need further development and where these approaches can have great impact [27]–[32]. Indeed, exploration and learning in unknown environments has been identified as one of the grand challenges of Science Robotics [33]. These problems are very complex, characterized by large state spaces and constrained communication. Many tools in machine learning and artificial intelligence have recently been developed for tackling difficult problems in robotics [34]–[37]. Of particular relevance to the current paper are those works that involve decision making under uncertainty and POMDP frameworks [38], [39].

As an application of our methodology, we consider a robot that must decide on the sequence of linearly arranged pipeline locations to explore and/or repair using prior information and observations made *in situ*. The actual damage of each location is initially unknown and can become worse if not repaired. This process is modeled by a Markov chain with known transition probabilities. The problem has a very large number of states ($\approx 10^{26}$ in our largest implementation). Our experiments demonstrate that our methodology is well suited to robotics applications that involve: i) large state spaces and long planning horizons, which exacerbate both the curse of dimensionality and the curse of history, and ii) decaying environments as in sequential repair problems, where it is important to use a policy that can identify and execute critical actions in minimum time.

We compare the performance of our proposed method with the POMCP and DESPOT methods from [8], [40], respectively, and we showcase the generality of our method by applying it to more complex versions of the pipeline problem: a two-dimensional grid pipeline, and a multi-robot variant of the problem. The use of distributed computation within our framework also suggests future applicability to multi-robot problems with asynchronous communication and/or bandwidth constraints, which are of great importance in robotics [30], [38], [41]–[43].

We also provide theoretical support for our methodology in the form of a performance bound (Prop. 1), which shows improvement of the rollout policy over the base policy (approximately). Alternative methodologies, such as POMCP and DESPOT, do not enjoy a comparable level of theoretical support.

In summary, the contributions of the paper include: 1) the development of an algorithmic framework for finding approximately optimal policies for large state space POMDP, including the development of distributed PI methods with a partitioned architecture, 2) performance bounds to support the architectural structure, and 3) implementation and validation in simulation of the proposed methods on a pipeline repair problem, whose character is shared by broad classes of problems in robotics.

II. Extended Belief Space Problem Formulation

The starting point of our paper is the classical belief space formulation of a POMDP. However, we extend this formulation to make it compatible with a distributed multiprocessor implementation of our algorithm. In particular, we propose to use as state a sufficient statistic that subsumes the belief state in the sense that its value determines the value of the belief state. In this section, we describe this extended belief space problem formulation.

We assume that there are n states denoted by $i \in \{1, \ldots, n\}$ and a finite set of controls U at each state. We denote by $p_{ij}(u)$ the transition probability from i to j under $u \in U$, and by $g(i, u, j)$ the corresponding transition cost. The cost is accumulated over an infinite horizon and is discounted by $\alpha \in (0, 1)$. At each new generated state j, an observation z from a finite set Z is obtained with known probability $p(z \mid j, u)$ that depends on j and the control u that was applied prior to the generation of j. The objective is to select each control optimally as a function of the prior history of observations and controls.

A classical approach to this problem is to convert it to a perfect state information problem whose state is the current belief $b = (b(1), \ldots, b(n))$, where $b(i)$ is the conditional distribution of the state i given the prior history. It is well-known that b is a sufficient statistic, which can serve as a substitute for the set of available observations, in the sense that optimal control can be achieved with knowledge of just b. In this letter, we consider a more general form of sufficient statistic, which we call the *feature state* and we denote by y. *We require that the feature state y subsumes the belief state b.* By this we mean that b can be computed exactly knowing y. One possibility is for y to be the union of b and some identifiable characteristics of the belief state, or some compact representation of the measurement history up to the current time (such as a number of most recent measurements, or the state of a finite-state controller). We also make the additional assumption that *y can be sequentially generated using a known feature estimator $F(y, u, z)$*. By this we mean that given that the current feature state is y, control u is applied, and observation z is obtained, the next feature can be exactly predicted as $F(y, u, z)$.

Clearly, since b is a sufficient statistic, the same is true for y. Thus the optimal costs achievable by the policies that depend on y and on b are the same. However, specific suboptimal schemes may become more effective with the use of the feature state y instead of just the belief state b. Moreover, the use of y can facilitate the use of distributed computation through partitioning of the space of features y, as we will explain later.

The optimal cost $J^*(y)$, as a function of the sufficient statistic/feature state y, is the unique solution of the corresponding Bellman equation $J^*(y) = (TJ^*)(y)$ for all y, where T is the Bellman operator

$$(TJ)(y) = \min_{\mu \in \mathcal{M}} (T_\mu J)(y),$$

with $\mathcal{M}$ being the set of all stationary policies [functions μ that map y to a control $\mu(y) \in U$], and T_μ being the Bellman operator corresponding to μ:

$$(T_\mu J)(y) = \hat{g}(y, \mu(y)) + \alpha \sum_{z \in Z} \hat{p}(z \mid b_y, \mu(y)) J(F(y, \mu(y), z)).$$

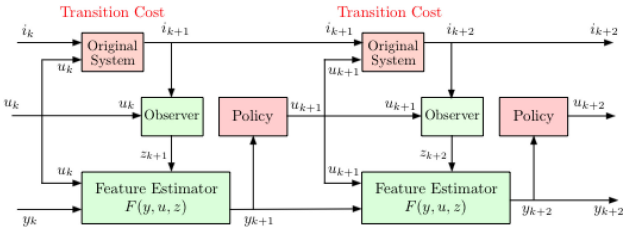

Fig. 1. Composite system simulator for POMDP for a given policy. The starting state i_k at stage k of a trajectory is generated randomly using the belief state b_k, which is in turn computed from the feature state y_k.

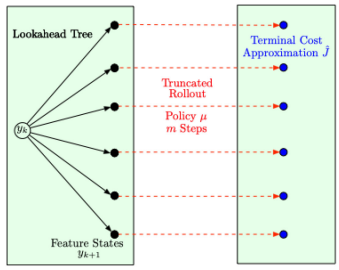

Fig. 2. A truncated rollout scheme: One-step lookahead is followed by m-step application of the base policy μ, and terminal cost approximation $\hat{J}$.

Here b_y denotes the belief state that corresponds to feature state y, $\hat{g}(y,u)$ is the expected cost per stage

$$\hat{g}(y,u) = \sum_{i=1}^{n} b_y(i) \sum_{j=1}^{n} p_{ij}(u) g(i,u,j),$$

$\hat{p}(z \mid b_y, u)$ is the conditional probability that the next observation will be z given the current belief state b_y and control u, and F is the feature state estimator.

The feature space reformulation of the problem can serve as the basis for approximation in value space, whereby J^* is replaced in Bellman's equation by some function $\tilde{J}$ after one-step or multistep lookahead. For example a one-step lookahead scheme yields the suboptimal policy $\tilde{\mu}$ given by

$$\tilde{\mu}(y) \in \arg\min_{u \in U} \left[\hat{g}(y,u) + \alpha \sum_{z \in Z} \hat{p}(z \mid b_y, u) \tilde{J}\left(F(y,u,z)\right) \right]. \tag{1}$$

In ℓ-step lookahead schemes, $\tilde{J}$ is used as terminal cost function in an ℓ-step horizon version of the original infinite horizon problem (see e.g., [11]). In the standard form of a rollout algorithm, $\tilde{J}$ is the cost function of some base policy. Here we adopt a rollout scheme with ℓ-step lookahead, which involves trajectory truncation and terminal cost approximation. This scheme has been formalized and discussed in the book [11] (Section 5.1), and is described in the next section.

III. Truncated Rollout With Terminal Cost Function Approximation

In the pure form of the rollout algorithm, the cost function approximation $\tilde{J}$ is the cost function J_μ of a known policy μ, called the *base policy*, and its value $\tilde{J}(y) = J_\mu(y)$ at any y is obtained by first extracting b from y, and then running a simulator starting from b, and using the system model, the feature generator, and μ (see Fig. 1). In the truncated form of rollout, $\tilde{J}(y)$ is obtained by running the simulator of μ for a given number of steps m, and then adding a terminal cost approximation $\hat{J}(\bar{y})$ for each terminal feature state $\bar{y}$ that is obtained at the end of the m steps of the simulation with μ (see Fig. 2 for the case where $\ell = 1$).

Thus the rollout policy is defined by the base policy μ, the terminal cost function approximation $\hat{J}$, the number of steps m after which the simulated trajectory with μ is truncated, as well as the lookahead size ℓ; see Fig. 2. The choices of m and ℓ are typically made by trial and error, based on computational tractability among other considerations, while $\hat{J}$ may be chosen on the basis of problem-dependent insight or through the use of some off-line approximation method. In variants of the method, the multistep lookahead may be implemented approximately using a Monte Carlo tree search or adaptive sampling scheme. In our experiments, we used a variable and state-dependent value of m. Monte-Carlo tree search did not work well in our initial experiments, and we subsequently abandoned it.

Using m-step application of the base policy between the ℓ-step lookahead and the terminal cost approximation gives the method the character of a single PI. We have used repeated truncated rollout as the basis for constructing a PI algorithm, which we will discuss shortly (see also [11], Section 5.1.2).

There are known error bounds that can be applied to the preceding truncated rollout scheme. These bounds were given in [11], Section 5.1, Prop. 5.1.3, but they were derived for the case where the number of states is finite, so they do not apply to our feature state representation of a POMDP. However, the bounds can be extended to our infinite feature space case, since the proof arguments of [11] do not depend on the finiteness of the state space. We thus give the bounds without proof.

Proposition 1: (Error Bounds) Consider a truncated rollout scheme consisting of ℓ-step lookahead, followed by application of policy μ for m steps, and a terminal cost function approximation $\hat{J}$ at the end of m steps. Let $\tilde{\mu}$ be the policy generated by this scheme. Then: (a) We have

$$\|J_{\tilde{\mu}} - J^*\| \leq \frac{2\alpha^\ell}{1-\alpha} \|T_\mu^m \hat{J} - J^*\|,$$

where $T_\mu^m \hat{J}$ is the result of applying m times to $\hat{J}$ the Bellman operator T_μ for policy μ, and $\|\cdot\|$ is the sup norm on the space of bounded functions of the feature state y. (b) We have for all y

$$J_{\tilde{\mu}}(y) \leq J_\mu(y) + \frac{2}{1-\alpha} \|\hat{J} - J_\mu\|.$$

The first bound implies that as the size of lookahead ℓ increases, the bound on the performance of the rollout policy improves. The second bound suggests that if $\hat{J}$ is close to J_μ, the performance of the rollout policy $\tilde{\mu}$ is approximately improved (to within an error bound), relative to the performance of the base policy μ. This is typical of the practically observed cost improvement property of rollout schemes. In particular, when $\hat{J} = J_\mu$ we obtain $J_{\tilde{\mu}} \leq J_\mu$, which is the theoretical policy improvement property of rollout (see [11], Section 5.1.2).

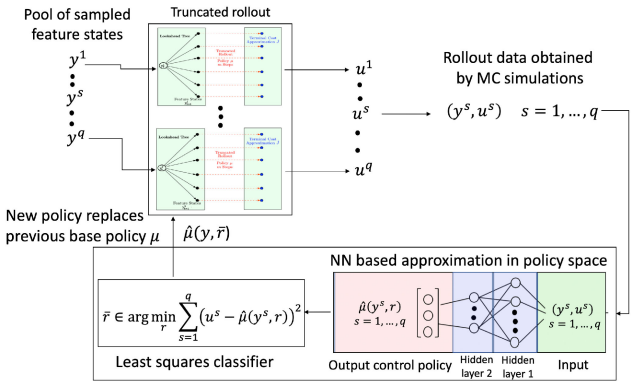

Fig. 3. Approximate PI scheme based on rollout and approximation in policy space.

IV. Supervised Learning of Rollout Policies and Cost Functions - Policy Iteration

The rollout algorithm uses multistep lookahead and on-line simulation of the base policy to generate the rollout control at any feature state of interest. To avoid the cost of on-line simulation, we can approximate the rollout policy off-line by using some approximation architecture that may involve a neural network. This is policy approximation built on top of the rollout scheme (see [11], Sections 2.1.5 and 5.7.2).

To this end, we introduce a parametric family/architecture of policies of the form $\hat{\mu}(y, r)$, where r is a parameter vector. We then construct a training set that consists of a large number of sample feature state-control pairs (y^s, u^s), $s = 1, \ldots, q$, such that for each s, u^s is the rollout control at feature state y^s. We use this data set to obtain a parameter $\bar{r}$ by solving a corresponding classification problem, which associates each feature state y with a control $\hat{\mu}(y, \bar{r})$. The parameter $\bar{r}$ defines a classifier, which given a feature state y, classifies y as requiring control $\hat{\mu}(y, \bar{r})$ (this idea was proposed in the context of PI in the paper [17], and is also described in the book [11], Section 3.5). The classification problem is often solved with the use of neural networks, and this has been our approach in our experimentation.

A. Approximate Policy Iteration (API)

We can also apply the rollout policy approximation to the context of PI. The idea is to view rollout as a one-step policy improvement, and to view the PI algorithm as a *perpetual rollout process*, which performs multiple policy improvements, using at each iteration the current policy as the base policy, and the next policy as the corresponding rollout policy. In particular, we consider a PI algorithm where at the typical iteration we have a policy μ, which we use as the base policy to generate many feature state-control sample pairs (y^s, u^s), $s = 1, \ldots, q$, where u^s is the rollout control corresponding to feature state y^s. We then obtain an "improved" policy $\hat{\mu}(y, \bar{r})$ with an approximation architecture and a classification algorithm, as described above. The "improved" policy is then used as a base policy to generate samples of the corresponding rollout policy, which is approximated in policy space, etc; see Fig. 3.

To use truncated rollout in this PI scheme, we must also provide a terminal cost approximation, which may take a variety of forms. Using zero is a simple possibility, which may work well if either the size ℓ of multistep lookahead or the length m of the simulated trajectory is relatively large. Another possibility, employed in our pipeline repair problem, is to use as terminal cost in the truncated rollout an approximation of the cost function of some base policy, which is obtained with a neural network-based approximation architecture.

In particular, at any policy iteration with a given base policy, once the rollout data is collected, one or two neural networks are constructed: A *policy network* that approximates the rollout policy, and (in the case of rollout with truncation) a *value network* that constructs a cost function approximation for that rollout policy (the essentially synonymous terms "actor network" and "critic network" are also common in the literature). We consider two methods:

1) *Approximate rollout and PI with truncation*, where each generated policy as well as its cost function are approximated by a policy and a value network, respectively. The cost function approximation of the current policy is used to truncate the rollout trajectories that are used to train the next policy.
2) *Approximate rollout and PI with no truncation*, where each generated policy is approximated using a policy network, but the rollout trajectories are continued up to a large maximum number of stages (enough to make the cost of the remaining stages insignificant due to discounting) or upon reaching a termination state. Here only a policy network is used; a value network is unnecessary since there is no rollout truncation.

Note that as in all approximate PI schemes, the sampling of feature states used for training is subject to exploration concerns. In particular, for each policy approximation, it is important to include in the sample set $\{y^s \mid s = 1, \ldots, q\}$, a subset of feature states that are "favored" by the rollout trajectories; e.g., start from some initial subset of feature states y^s and selectively add to this subset feature states that are encountered along the rollout trajectories. This is a challenging issue, which must be approached with care; see [17], [18].

V. Policy Iteration With a Partitioned Architecture

We will now discuss our partitioned architecture. It is based on a partition of the set Y of feature states into disjoint sets $Y_1, \ldots, Y_N$, so that $Y = \cup_{\nu=1}^{N} Y_\nu$. We train in parallel a separate (local) policy network and a (local) value network (in the case of a scheme with truncation) using feature state data from each of the sets $Y_1, \ldots, Y_N$. Thus at each policy iteration, we use N policy networks and as many as N additional value networks (in some specially structured problems, including the pipeline repair problem, some of the value networks may not be needed, as we will discuss later). The N local policy networks, each defined over a subset Y_ν, are combined into a global policy, defined over the entire feature space Y. For schemes with truncation, each local value network is trained using the global policy, but with starting feature states from the corresponding subset Y_ν of the partition; see Fig. 4.

This partitioned architecture is well-suited for distributed computation, with the local networks sharing and updating

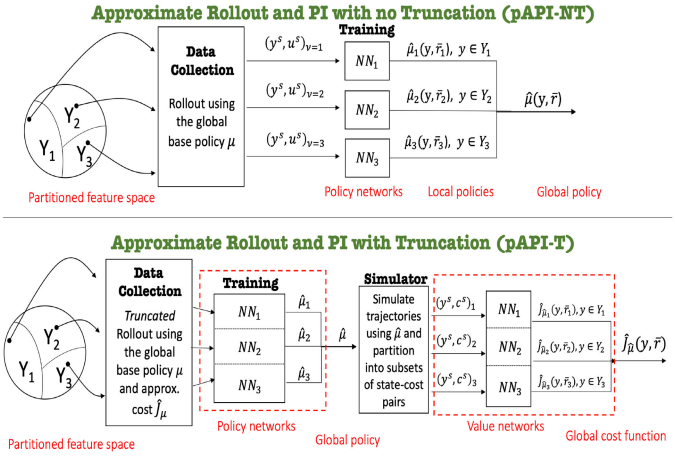

Fig. 4. Partitioned architecture for rollout and approximate PI. For truncated rollout, we may employ terminal cost approximation using a value network.

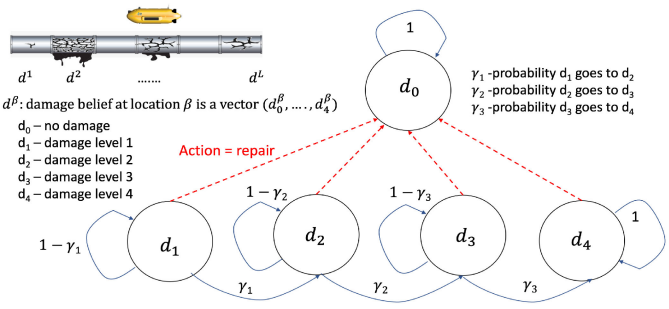

Fig. 5. Markov chain for the damage level of each location of the pipeline.

piecemeal the current global policy and (in the case of truncation) the current global terminal cost approximation. Moreover, we speculate that our methodology requires smaller training sets, which cover more evenly the feature space, thereby addressing in part the issue of adequate feature space exploration. While it is hard to quantify this potential advantage, our computational experience indicates that it is substantial. Regarding the distributed training of our partitioned architecture, we may conceptually assign one virtual processor for each set of the feature space partition (of course multiple virtual processors can coexist within the same physical processor). The empirical work described in the following sections provides, among others, guidelines for future research directions on the performance and implementations of synchronous and asynchronous distributed PI with partitioned architectures (in the spirit of [22]), particularly in the context of POMDP as well as multi-agent robotics.

VI. Sequential Repair and the Pipeline Problem

Various elements of the approximate PI methodology just described have been applied to a pipeline repair problem. We will describe the implementation, computational results, and comparisons with other methodologies in detail. The problem involves autonomous repair of a number of pipeline locations where damage may have occurred, and the objective is to find a policy to repair the pipeline with minimum cost, based on available information. Our results show that 1) approximate PI can be successfully applied in the context of POMDP, 2) a partitioned architecture results in substantial computation time savings, thus allowing applicability of our methodology to large state space problems, and 3) our pipeline repair model and solution methods can be extended to complex two-dimensional and multi-robot contexts, where a solution can be facilitated by using partitioning.

A. Pipeline Repair Problem Description

In this problem, an autonomous robot moves along a pipeline consisting of a sequence of L locations, denoted $1,\dots,L$. Each location of the pipeline can be in one of $\zeta+1$ progressively worse damage levels indicated by $d_0,d_1,\dots,d_\zeta$, where d_0 indicates a no damage condition. The damage level of each location changes stochastically over time according to a known Markov chain with the $\zeta+1$ states $0,1,\dots,\zeta$ (see Fig. 5). As the figure indicates, we assume that a location that is not damaged (state d_0), cannot become damaged. However, for a damaged location, the level of damage can stochastically become worse. We assume that the robot has a sensing radius of one location within which it can verify the damage level of the location. Thus with each visit to a location, an observation is obtained, namely, the exact damage level of the location.

The robot knows its current location, and maintains for each location β, a belief state of damages $d^\beta=(d_0^\beta,\dots,d_\zeta^\beta)$ consisting of the conditional probabilities $d_0^\beta,\dots,d_\zeta^\beta$ of the damage level, given the current action-observation history. The initial belief state could be obtained from information gathered *a priori*, for example, via noisy images of the pipeline. Upon reaching a location β, the robot determines its damage level, and decides upon one of three actions: Stay in β for one time period and repair the damage, bringing it to level d_0, or move to one of the two adjacent locations $\beta-1$ or $\beta+1$ without repairing the damage (if $\beta=1$ or $\beta=L$, only two of these actions are available). There is a known cost per unit time for each location, depending on its damage level, and the objective is to minimize the discounted sum of costs of all the locations of the pipeline over an infinite horizon. This is a POMDP with $L\cdot(\zeta+1)^L$ states and three actions per state.

It is straightforward to implement the belief state estimator, given the initial belief distributions of the damage levels of all the locations. It is also straightforward to program a simulator of the type illustrated in Fig. 1, and to generate rollout trajectories with a given base policy starting from a given feature state.

B. Partitioned Architecture for the Pipeline Repair Problem

For our partitioning scheme, we use two characteristics of the belief vector. These are (see Fig. 6 a):

Percentage of disrepair: If less than one half of the pipeline locations are believed to be in some damaged level (d_1,d_2,d_3,d_4), we say that we are in an *endgame* state, and otherwise we say that we are in a *startgame* state.

Density of possible damage: We distinguish three cases, based on two variables, LD and RD (Left and Right Damage),

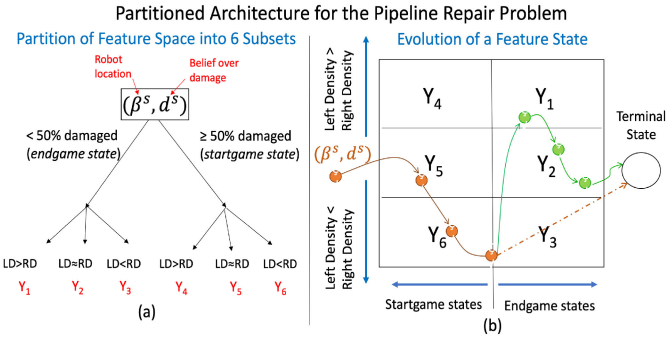

Fig. 6. Partition of the feature space into subsets for the pipeline problem.

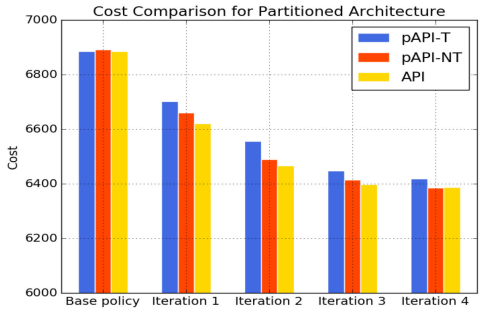

Fig. 8. Performance comparison of API, where pAPI-T and pAPI-NT use partitioning, with and without truncation respectively.

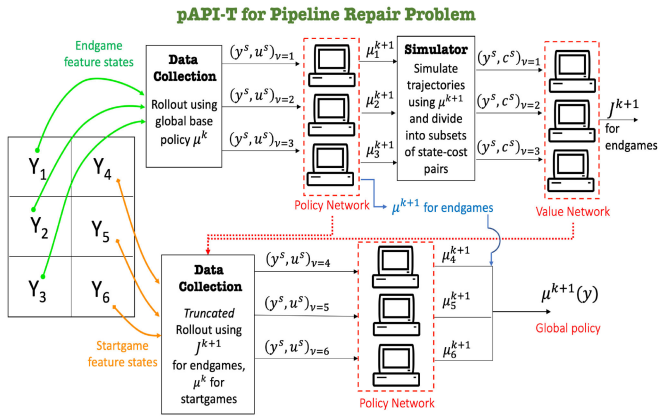

Fig. 7. Algorithm for applying pAPI-T to our pipeline repair problem.

which will be defined shortly. The three cases are a) damage to the left is greater than to the right: $\frac{LD}{LD+RD} > 0.7$, b) damage to the left is approximately equal to damage to the right: $0.3 \leq \frac{LD}{LD+RD} \leq 0.7$, and c) damage to the right is greater than to the left: $\frac{LD}{LD+RD} < 0.3$.

Based on the six possible pairs of the characteristics above, the belief space is partitioned into six corresponding subsets, and the index of the current subset is added to the belief state b to form the feature state y (hence y subsumes b and its evolution can be sequentially simulated, in conjunction with b, consistent with our assumptions). Since we assume that a repaired location cannot fall into disrepair, endgame states cannot lead to startgame states. This allows a sequential training strategy whereby endgame cost and policy approximations can be used for training startgame policy and value networks.

C. Algorithmic Implementation

We employ two algorithms that are based on the partitioning scheme just described: Partitioned Approximate Policy Iteration with No Truncation (pAPI-NT) and Partitioned Approximate Policy Iteration with Truncation (pAPI-T) (see Fig. 7). These algorithms are well-suited for distributed computation, with a separate processor being in charge of the simulation and training within its own subset of the partition, and communicating its local policy and cost function to the other processors for use in their truncated rollout computations. Also, the approximate cost functions for the endgame subsets are shared as terminal cost approximations with other processors that are learning startgame policies. We find that these schemes save computation time while performing almost as well as a nonpartitioned architecture in terms of minimizing cost (see the next section and Fig. 8).

D. Experimental Results

We implemented our proposed methodology on various problems of different sizes: 1) A linear pipeline problem of 20 locations and 10^{15} states, 2) a two-dimensional grid pipeline of size 6×6 and 10^{26} states, and 3) a two-robot linear pipeline of 10^{16} states. We describe our experimentation for the linear pipeline first.

For our experiments we have made the following choices. We used neural networks for both policy and value approximations. The policy networks used 2 hidden layers of size 256 and 64 ReLU units respectively, while the value networks used 3 hidden layers of 256, 128, and 64 ReLU units. The output layer of the policy network used a softmax layer over 3 actions (repair, go left, go right) in a linear single-robot pipeline (5 actions for a single robot 2D grid, 9 actions for two-robot linear pipeline). We used the RMSprop optimizer to minimize the L2 loss with a learning rate of 0.001.

To obtain the rollout policy, we used one-step lookahead and 10 Monte-Carlo simulated trajectories with the base policy from each leaf of the lookahead tree. We have not tried multistep lookahead in order to keep the size of the lookahead tree manageable. We used a greedy policy as the starting base policy, which repairs the current location if it is damaged, moves left if out of all damaged locations, the nearest one is on the left, or moves right otherwise. The density of damage to the left of the robot at location β is defined as $LD = \sum_{j=0}^{\beta-1} d^{j'} c$, where $c = [0,\ 0.1,\ 1,\ 10,\ 100]$ is the vector of costs for the different damage levels. The definition of RD is similar. For the 20-location linear pipeline cases, 200,000 training samples were used for each subset of the feature space partition (and 1.2×10^6 samples for training using the methods that do not use partitioning). For the 6×6 grid and two-robot cases, 500,000 training samples were used for each subset of the partition. Training samples were obtained by: 1) random generation of feature states from each partition set (Fig. 6(b)), and 2) augmenting the pool of

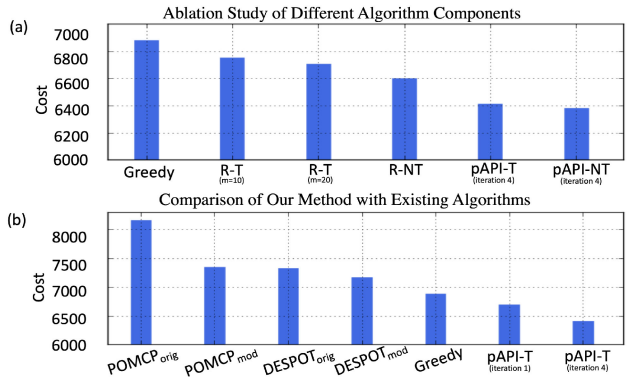

Fig. 9. (a) Ablation study demonstrating performance contributions of the different components of our methodology. (b) Comparison of our methods to DESPOT and POMCP.

samples with representative states over a memory buffer where we randomly retained around 10% of states generated while evaluating the previous policies as described in [11], Ch. 5.

We have generally observed that the partitioned architecture results in significant improvement in running time thanks to parallelization, at the expense of a modest degradation in performance (as shown in Fig. 8). We expect that this improvement will become more significant as we address larger problems that require a richer feature space partition.

We will now present an ablation study of our proposed method that demonstrates the performance of the major pieces of our algorithm. In particular, we compare: 1) pAPI-T and pAPI-NT, which are the full versions of our algorithm with and without truncation respectively, 2) R-T and R-NT, which are rollout algorithms with and without truncation, which are the basis for data collection and training for the improved policies used in pAPI-T and pAPI-NT, and finally 3) the greedy policy, which is used as the starting policy for rollout and for PI. The plot in Fig. 9(a) shows how each component of our method adds to improved performance. Notice that pAPI-T uses a terminal cost approximation and results in computational savings over pAPI-NT, but incurs a slight approximation penalty.

1) Comparison to Existing Methods: We have compared our method with two published POMDP methods: DESPOT and POMCP. Our simulations were conducted using the code for these methods, which is freely available at [40]. These methods do not use rollout with a base policy, but instead rely on long lookahead supplemented by Monte Carlo tree search and/or heuristic pruning of the lookahead tree. Results are shown in Fig. 9(b) for an evaluation set of 1000 random startgame states. We tried many DESPOT parameter combinations, including $K = 100, 200, 300$, $D = 60, 90, 100$, and $\lambda = 0.1, 0.3$. Fig. 9(b) shows the best results that we obtained with DESPOT using $D = 90$, $K = 200$, $\lambda = 0.1$, and with POMCP using the default parameters. We used a closed form of belief update equation based on the Markov chain of Fig. 5, by modifying the POMCP and DESPOT code to use a single particle to represent the belief with a weight of 1.

Our results show the cost of two implementations of POMCP and DESPOT; the first implementation denoted DESPOT$_{\mathbf{orig}}$ is "straight out of the box," using the code provided in [40]. We obtained the other implementation, DESPOT$_{\mathbf{mod}}$, by modifying the first to disallow "repair" actions at already repaired locations. We note that without this modification the DESPOT and POMCP algorithms will sometimes choose a repair action on an already repaired location, leading to higher costs. Our results indicate that pAPI-NT and pAPI-T outperform DESPOT and POMCP (original and modified versions) for our pipeline repair problem. The greedy policy, somewhat unexpectedly, also performs better than DESPOT and POMCP. The reason may be that the pipeline repair problem has a long effective planning horizon. Intuitively for such problems, DESPOT and POMCP are handicapped by their reliance on a tradeoff between lookahead length and the sparsity of the lookahead tree. Since both DESPOT and POMCP estimate Q-values approximately, with a larger and sparser lookahead, they can perform worse than the greedy base policy. By contrast, our rollout policy has the cost improvement property described in Prop. 1, and improves over the base policy (approximately, within a bound).

TABLE I
EXTENSIONS TO PIPELINE REPAIR PROBLEM

Extension	Greedy	Iter 1	Iter 2
pAPI-NT on 5x4 grid	5476.79	5255.64	4952.26
pAPI-NT on 6x6 grid	18799.5	18037.1	17341.9
pAPI-NT 2-robot, linear ($L = 20$)	4142.84	3321.95	3133.08

Indeed, by examination of sample generated trajectories, we have observed that for states requiring a relatively long horizon to repair the pipeline, POMCP and DESPOT often make poor decisions. We have concluded that the advantage that our method holds is principally due to the long horizon exploration that is characteristic of the use of rollout with terminal cost approximation.

2) Extensions of the Pipeline Repair Problem: We present in Table I results for two extensions of our problem to demonstrate the generality of our methodology. Specifically we show results for: 1) a two-dimensional grid pipeline and 2) a two-robot implementation of our methodology. Generally, our results show that both pAPI-T and pAPI-NT can be applied to problems with larger state space and multiple robots thanks to the use of partitioning and the attendant computational savings.

VII. CONCLUDING REMARKS

This letter develops rollout algorithms and PI methods that are well-suited to deal with the challenges of POMDP, including large state spaces, incomplete information, and long planning horizons. Thus these methods are of high relevance to robotic tasks in uncertain environments such as search and rescue. While several of the components of our methodology have been suggested for perfect state information problems, they have not been combined and adapted to POMDP. Our methods are based on partitioning the feature space and training local policies that are specialized, easier to train, can be combined to provide a global policy over the entire space, and are amenable to a highly distributed implementation.

We have applied our methods in simulation to a class of sequential repair problems where a robot inspects and repairs a linear pipeline with potentially several rupture sites under partial information about the state of the pipeline (acquired from *a priori* obtained knowledge and in situ observations). Our method's partitioning of the state space has important implications for robotics problems – namely, it allows for massive parallelization over several potentially independent processors (or robots). Importantly, this framework lends itself to distributed learning about the environment where different partitions can correspond to states in spatially different areas of the world, thus suggesting a new basis from which to solve future multi-robot POMDP problems.

We finally note that there are several possible extensions to our sequential repair problem, for example, allowing for stochastic repair times, two-dimensional problems with obstacles, and multiple robots.

ACKNOWLEDGMENT

The authors would like to thank C. Norman and S. Kailas for their help with numerical studies.

REFERENCES

[1] J. Pineau, G. Gordon, and S. Thrun, "Anytime point-based approximations for large POMDPs," *J. Artif. Intell. Res.*, vol. 27, pp. 335–380, 2006.
[2] M. T. J. Spaan and N. Vlassis, "Perseus: Randomized point-based value iteration for POMDPs," *J. Artif. Intell. Res.*, vol. 24, pp. 195–220, 2005.
[3] G. Shani, J. Pineau, and R. Kaplow, "A Survey of point-based POMDP solvers," *Auton. Agents Multi-Agent Syst.*, vol. 27, pp. 1–51, 2013.
[4] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artif. intell.*, vol. 101, pp. 99–134, 1998.
[5] E. A. Hansen, "Solving POMDPs by searching in policy space," in *Proc. 14th Conf. Uncertainty Artif. Intell.*, San Francisco, CA, USA, 1998, pp. 211–219.
[6] N. Meuleau, K.-E. Kim, L. P. Kaelbling, and A. R. Cassandra, "Solving POMDPs by searching the space of finite policies," 2013, *arXiv:1301.6720 [cs.AI]*.
[7] S. Ross, J. Pineau, S. Paquet, and B. Chaib-draa, "Online planning algorithms for POMDPs," *J. Artif. Intell. Res.*, vol. 32, pp. 663–704, 2008.
[8] D. Silver and J. Veness, "Monte-carlo planning in large POMDPs," in *Proc. 23rd Int. Conf. NeurIPS*, Red Hook, NY, USA, 2010, pp. 2164–2172.
[9] R. Zhou and E. A. Hansen, "An improved grid-based approximation algorithm for POMDPs," in *Proc. 17th Int. Joint Conf. Artif. Intell.*, San Francisco, CA, USA, 2001, pp. 707–714.
[10] H. Yu and D. Bertsekas, "Discretized approximations for POMDP with average cost," 2012, *arXiv:1207.4154 [cs.AI]*.
[11] D. P. Bertsekas, *Reinforcement Learning and Optimal Control*. Belmont, MA, USA: Athena Scientific, 2019.
[12] J. Baxter and P. L. Bartlett, "Infinite-horizon policy-gradient estimation," *J. Artif. Intell. Res.*, vol. 15, pp. 319–350, 2001.
[13] H. Yu, "A function approximation approach to estimation of policy gradient for POMDP with structured policies," 2012, *arXiv:1207.1421. [cs.LG]*.
[14] R. M. Estanjini, K. Li, and I. C. Paschalidis, "A least squares temporal difference actor–critic algorithm with applications to warehouse management," *Naval Res. Logistics*, vol. 59, pp. 197–211, 2012.
[15] G. Tesauro and G. R. Galperin, "On-line policy improvement using monte-carlo search," in *Proc. Adv. NeurIPS*, Cambridge, MA, 1996, pp. 1068–1074.
[16] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.
[17] M. G. Lagoudakis and R. Parr, "Reinforcement learning as classification: Leveraging modern classifiers," in *Proc. 20th Int. Conf. Mach. Learn.*, 2003, pp. 424–431.
[18] C. Dimitrakakis and M. G. Lagoudakis, "Rollout sampling approximate policy iteration," *Mach. Learn.*, vol. 72, pp. 157–171, 2008.
[19] A. Lazaric, M. Ghavamzadeh, and R. Munos, "Analysis of a classification-based policy iteration algorithm," in *Proc. 27th Int. Conf. Mach. Learn.*, Madison, WI, 2010, pp. 607–614.
[20] D. Liu and Q. Wei, "Policy iteration adaptive dynamic programming algorithm for discrete-time nonlinear systems," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 25, pp. 621–634, 2014.
[21] D. Silver *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017, *arXiv:1712.01815 [cs.AI]*.
[22] D. P. Bertsekas and H. Yu, "Distributed asynchronous policy iteration in dynamic programming," in *Proc. 48th Annu. Allerton Conf. Commun., Control, Comput.*, 2010, pp. 1368–1375.
[23] D. P. Bertsekas and H. Yu, "Q-learning and enhanced policy iteration in discounted dynamic programming," *Math. Oper. Res.*, vol. 37, pp. 66–94, 2012.
[24] H. Yu and D. P. Bertsekas, "Q-learning and policy iteration algorithms for stochastic shortest path problems," *Annals Oper. Res.*, vol. 208, pp. 95–132, 2013.
[25] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, vol. 2, 4th ed., Belmont, MA, USA: Athena Scientific, 2012.
[26] D. P. Bertsekas, *Abstract Dynamic Programming*, 2nd ed. Belmont, MA, USA.: Athena Scientific, 2018.
[27] S. Waharte and N. Trigoni, "Supporting search and rescue operations with UAVs," in *Proc. Int. Conf. Emerg. Secur. Technologies*, 2010, pp. 142–147.
[28] A. Cassandra, "A survey of POMDP applications," in *Proc. Working Notes AAAI Fall Symp. Planning POMDP*, 1998, pp. 17–24.
[29] S. M. Veres, L. Molnar, N. K. Lincoln, and C. P. Morice, "Autonomous vehicle control systems - A review of decision making," in *Proc. Institution Mech. Engineers, Part I: J. Syst. Control Eng.*, 2011, vol. 225, pp. 155–195.
[30] M. Dunbabin, P. Corke, I. Vasilescu, and D. Rus, "Experiments with cooperative control of underwater robots," *Int. J. Robot. Res.*, vol. 28, pp. 815–833, 2009.
[31] J. Das *et al.*, "Data-driven robotic sampling for marine ecosystem monitoring," *Int. J. Robot. Res.*, vol. 34, pp. 1435–1452, 2015.
[32] W. Schwarting, J. Alonso-Mora, and D. Rus, "Planning and decision-making for autonomous vehicles," *Annu. Rev. Control, Robot., Auton. Syst.*, vol. 1, pp. 187–210, 2018.
[33] G.-Z. Yang *et al.*, "The grand challenges of Science Robotics," *Sci. Robot.*, vol. 3, pp. 1–14, 2018.
[34] M. Everett, Y. F. Chen, and J. P. How, "Motion planning among dynamic, decision-making agents with deep reinforcement learning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2018, pp. 3052–3059.
[35] H. Kretzschmar, M. Spies, C. Sprunk, and W. Burgard, "Socially compliant mobile robot navigation via inverse reinforcement learning," *Int. J. Robot. Res.*, vol. 35, pp. 1289–1307, 2016.
[36] N. Sünderhauf *et al.*, "The limits and potentials of deep learning for robotics," *Int. J. Robot. Res.*, vol. 37, pp. 405–420, 2018.
[37] F. Ingrand and M. Ghallab, "Deliberation for autonomous robots: A survey," *Artif. Intell.*, vol. 247, pp. 10–44, 2017.
[38] S. Omidshafiei, A.-A. Agha-Mohammadi, C. Amato, and J. P. How, "Decentralized control of partially observable markov decision processes using belief space macro-actions," in *IEEE Int. Conf. Robot. Automat.*, 2015, pp. 5962–5969.
[39] S. Yi, C. Nam, and K. Sycara, "Indoor pursuit-evasion with hybrid hierarchical partially observable markov decision processes for multi-robot systems," in *Proc. Distrib. Auton. Robot. Syst.*, 2019, vol. 9, pp. 251–264.
[40] A. Somani, N. Ye, D. Hsu, and W. S. Lee, "DESPOT: Online POMDP planning with regularization," in *Proc. Advances NeurIPS 26*, 2013, pp. 1772–1780.
[41] S. Gil, S. Kumar, D. Katabi, and D. Rus, "Adaptive communication in multi-robot systems using directionality of signal strength," *Int. J. Robot. Res.*, vol. 34, pp. 946–968, 2015.
[42] W. Wang, N. Jadhav, P. Vohs, N. Hughes, M. Mazumder, and S. Gil, "Active rendezvous for multi-robot pose graph optimization using sensing over Wi-Fi," 2019, *arXiv:1907.05538 [cs.RO]*.
[43] S. Gil, D. Feldman, and D. Rus, "Communication coverage for independently moving robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2012, pp. 4865–4872.