Controller Compilation and Compression for Resource Constrained Applications

Marek Grześ, Pascal Poupart and Jesse Hoey

David R. Cheriton School of Computer Science, University of Waterloo,
200 University Avenue West, Waterloo, Ontario N2L 3G1, Canada
{mgrzes, ppoupart, jhoey}@uwaterloo.ca

Abstract. Recent advances in planning techniques for partially observable Markov decision processes have focused on online search techniques and offline point-based value iteration. While these techniques allow practitioners to obtain policies for fairly large problems, they assume that a non-negligible amount of computation can be done between each decision point. In contrast, the recent proliferation of mobile and embedded devices has lead to a surge of applications that could benefit from state of the art planning techniques if they can operate under severe constraints on computational resources. To that effect, we describe two techniques to compile policies into controllers that can be executed by a mere table lookup at each decision point. The first approach compiles policies induced by a set of alpha vectors (such as those obtained by point-based techniques) into approximately equivalent controllers, while the second approach performs a simulation to compile arbitrary policies into approximately equivalent controllers. We also describe an approach to compress controllers by removing redundant and dominated nodes, often yielding smaller and yet better controllers. The compilation and compression techniques are demonstrated on benchmark problems as well as a mobile application to help Alzheimer patients to way-find.

Keywords: Energy-efficiency, Finite-state Controllers, Knowledge compilation, Markov decision processes, Mobile Applications, POMDPs

1 Introduction

Partially observable Markov decision processes (POMDPs) provide a natural framework for sequential decision making in partially observable domains. Tremendous progress has been made in recent years to develop scalable planning techniques for POMDPs. Point-based value iteration methods for factored and continuous domains can compute good value policies for a wide range of real-world problems [1, 2]. In addition, online resources can be used to perform a search at run time to directly select the next action or refine a precomputed policy [3, 4].

In this work, we are motivated by an emerging class of applications that pose new challenges for POMDP solvers. We consider monitoring and assistive applications that run on smart-phones, wearable systems or other mobile devices.

While computational resources are rapidly increasing, energy consumption remains an important bottleneck due to limited battery life. This is especially important in monitoring and assistive applications that need to be continuously running, but should be as power efficient as possible. For such applications, online planning is not an option due to the high computational costs. Computed policies that require online belief monitoring at execution time also consume too much energy. While it is sometimes possible to offload computation through cloud solutions, this requires a data connection, which may not always be available or stable, and which has a high battery consumption.

An effective solution can be found by noting that a POMDP policy can be represented very simply using a finite state controller (FSC) [5], which only requires simple table look-ups during execution. However, controller optimization is notoriously difficult. The non-convex nature of the optimization makes it difficult for many approaches (e.g., gradient ascent [6], quadratically constrained optimization [7], bounded policy iteration [8], expectation maximization [9]) to reliably find the global optimum. An exhaustive search of the space of controllers can avoid local optima, but is clearly intractable [10, 11].

In this paper, we describe two novel techniques for compiling an existing POMDP policy (as generated by a point-based method, for example) into a finite state controller (Sec. 3). The first method requires a policy specified as a set of α-vectors and witness belief points and constructs a FSC directly that approximates the given policy. The second method needs only a simulation of the policy, and builds a controller incrementally by building a policy tree and then detecting equivalent conditional plans. We also describe a novel method for compressing a FSC into an equivalent, but smaller, FSC by removing redundant nodes (Sec. 4). We demonstrate our techniques on a set of large benchmark POMDP problems (Sec. 5), and we use policies generated by two state-of-the-art point-based techniques, namely GapMin [12] and SARSOP [13]. We show how we can construct very compact controllers that are equivalent, and sometimes better, than the policies they are derived from. We also demonstrate our methods on a set of POMDPs that are used to provide mobile assistance for persons with Alzheimer's disease for way-finding.

2 Background

A partially observable Markov decision process (POMDP) is formally defined by a tuple $\langle S, A, O, T, Z, R, b_0, \gamma \rangle$ which includes a set S of states s, a set A of actions a, a set O of observations o, a transition function $T(s', s, a) = \Pr(s'|s, a)$, an observation function $Z(o, a, s') = \Pr(o|s', a)$, a reward function $R(s, a) \in \Re$, an initial belief $b_0(s) = \Pr(s)$ and a discount factor $0 \leq \gamma \leq 1$. We assume that the planning horizon is infinite, although the proposed algorithms can be modified easily for finite horizon problems. The goal is to find an optimal policy that maximizes the discounted sum of rewards. A policy $\pi : H_t \to A_t$ can be defined as a mapping from histories $H_t \equiv A_0 \times O_1 \times ... \times A_{t-1} \times O_t$ of past actions and observations to actions A_t, however this definition is problematic for an infinite horizon since histories may be arbitrarily long. Two approaches are often used to circumvent this issue: i) replace histories by finite length sufficient statistics such as beliefs or ii) represent policies as finite state controllers, which are mappings from *cyclic* histories to actions.

A belief $b(s)$ is a distribution over states reflecting the decision maker's belief that the process may be in each state s. We can update a belief b after executing a and observing o according to Bayes' theorem:

$$b^{ao}(s') \propto \sum_{s} b(s) \Pr(s'|s,a) \Pr(o|s',a) \; \forall s' \tag{1}$$

Given the initial belief b_0 and a history $h_t = \langle a_0, o_1, ..., a_{t-1}, o_t \rangle$, we can compute the belief b_t at time step t by repeatedly applying the above equation for each action-observation pair in the history. Hence, we can equivalently define policies as mappings $\pi : B \rightarrow A$ from beliefs to actions. The value $V^{\pi}(b_0)$ of policy π when starting in b_0 is the discounted sum of expected rewards $V^{\pi}(b_0) = \sum_{t=0}^{\infty} \gamma^t R(b_t, \pi(b_t))$ where $R(b,a) = \sum_s b(s) R(s,a)$.

We can also consider policies represented by a finite state controller $\pi = \langle N, \phi, \psi \rangle$, which is defined by a set N of nodes n, a mapping $\phi : N \rightarrow A$ indicating which action a to execute in each node n and a mapping $\psi : N \times O \rightarrow N$ indicating that the edge rooted at n and labeled by o should point to n'. A controller is executed by alternating between executing the action $\phi(n)$ of the current node n and moving to the next node $\psi(n,o)$ by following the edge rooted at n that is labeled with the current observation o. The value α_n of the controller when starting in n is an $|S|$-dimensional vector computed as follows:

$$\alpha_n(s) = R(s, \phi(n)) + \gamma \sum_{s',o} \Pr(s'|s,a) \Pr(o|s',a) \alpha_{\psi(n,o)}(s') \; \forall n, s \tag{2}$$

Policy optimization algorithms can be classified in two broad categories: i) *offline* techniques that pre-compute a policy before the start of the execution [14–16] and ii) *online* techniques that perform all their computation at run time by searching for the best action to execute after receiving each observation [3]. Online techniques can take advantage of the history so far to focus their computation only on the current belief. When computational resources are not constrained and there is sufficient time between decisions to search for the next action to execute, online techniques can perform very well and can scale to very large problems. In contrast, offline techniques do not scale as well, but permit the deployment of POMDP policies on mobile and/or embedded devices with severe resource constraints due to energy, memory or CPU limitations.

Among the offline techniques, we can further classify algorithms based on the type of policies (belief mapping or finite state controller) that they produce. Algorithms that produce belief mappings often exploit the fact that the value V^* of an optimal policy satisfies Bellman's equation:

$$V^*(b) = \max_a \sum_s b(s) \Big[R(s,a) + \gamma \sum_{s',o} \Pr(s'|s,a) \Pr(o|s',a) V^*(b^{ao}) \Big] \forall b \tag{3}$$

The continuous nature of the belief space prevents us from performing value iteration at all beliefs and therefore the important class of point-based techniques performs point-based Bellman backups only at a finite set of beliefs [15]. An approximation of the value function at all beliefs is obtained by computing the gradient in addition to the value at each belief. This allows the formation of a set of linear value functions that are often represented by α-vectors, similar to the value functions of controller nodes. While the details of point-based value iteration are not important for the rest of this paper (see [17] for more information), what is important to know is that they produce a set Γ of $\langle \alpha_i, b_i, a_i \rangle$-tuples that associate each α_i with an action a_i and a witness belief b_i (i.e., belief for which α_i yields the highest value: $\alpha_i(b_i) \geq \alpha_j(b_i) \; \forall j$ where $\alpha(b) = \sum_s b(s)\alpha(s)$). The policy π induced by Γ is obtained by computing

$$\pi(b) = a_{best} \text{ where } best = \arg\max_i \; \alpha_i(b) \tag{4}$$

Although point-based value iteration techniques compute the set Γ offline, they still require a certain amount of computation at each decision point. The belief must be updated after each action and observation according to Eq. 1 (complexity $O(|S|^2)$) and the best α-vector must be identified according to Eq. 4 (complexity $O(|S||\Gamma|)$). This amount of computation may still be prohibitive when S and Γ are large and there isn't enough memory, time or energy.

Alternatively, the other group of offline techniques produces policies represented as finite state controllers [10]. Since the execution of a controller merely consists of a table lookup, they are the most convenient type of policies for deployment in resource constrained applications. Unfortunately, they do not scale as well as point-based techniques and they often lack robustness due to local optima issues. Instead of directly optimizing a finite state controller, in this paper we propose two techniques to compile policies into approximately equivalent controllers. This has the benefit that we can use existing scalable algorithms such as point-based value iteration to quickly obtain a good policy. In addition, the controller compilation allows those policies to be executed on devices that are much more constrained.

3 Controller Compilation

Kaelbling et al. [5] observed that an optimal controller can be extracted from an optimal value function. Unfortunately, the best value functions found by state of the algorithms are approximate/suboptimal for most problems. Hansen wrote "it is unclear how to construct suboptimal controllers from [such value functions]" [18]. Hence, for the past 15 years, research has focused on directly optimizing controllers. We propose two approaches to compile suboptimal policies into approximately equivalent controllers. The first approach is limited to policies implicitly represented by sets of α-vectors as produced by point-based value iteration techniques. The second approach works with arbitrary policies.

3.1 Compiling Controllers from Alpha Vectors

As explained in Sec. 2, point-based value iteration techniques produce a set Γ of $\langle \alpha_i, b_i, a_i \rangle$-tuples from which a belief mapping policy is extracted. Alg. 1 shows how to compile Γ into an approximately equivalent controller $\langle N, \phi, \psi \rangle$. We create a node n_i for each vector α_i (Line 4). Each node n_i is labeled with the action $\phi(n_i) = a_i$ associated with α_i (Line 5). To determine where the edge rooted at n_i and labeled with o should point to, we update the witness b_i of belief according to Eq. 1 based on action a_i and observation o. Let the resulting belief be $b_i^{a_i,o}$. We then find which α-vector has the highest value at $b_i^{a_i,o}$ (Line 9) and assign the corresponding node to $\psi(n_i, o)$ (Line 10). The complexity of this compilation technique is $O(|\Gamma|^2|O||S|^2)$, however in practice the dependence on $|O|$ and $|S|$ can often be reduced by exploiting sparsity. The overall running time is typically a fraction of the time taken by point-based value iteration to obtain Γ. The quality of the resulting controller varies. The compilation technique ensures that the actions selected at the first two time steps are identical to that of the policy induced by Γ. This can be observed by noting that the action associated with the best α-vector is selected at the first two time steps which correspond exactly to what would be done in a policy induced by Γ. If the set of α-vectors in Γ corresponds to the optimal value function, then the resulting controller will also be optimal. However, when the set of α-vectors is suboptimal, which is the case most of the time, then actions selected after the second time step may be different than those selected by the policy induced by Γ, leading to a controller that may be better or worse. In the next section we describe an approach that ensures that the resulting controller is at least as good as the original policy in the limit.

Algorithm 1: Compilation of α-vectors into an approximately equivalent controller $\langle N, \phi, \psi \rangle$

```
ALPHA2FSC($\Gamma$)
 1: Let $\Gamma$ be a set of $\langle \alpha_i, b_i, a_i \rangle$-tuples
 2: $N \leftarrow \emptyset$
 3: for $i = 1$ to $|\Gamma|$ do
 4:     $N \leftarrow N \cup \{n_i\}$
 5:     $\phi(n_i) \leftarrow a_i$
 6: for $i = 1$ to $|\Gamma|$ do
 7:     for all $o \in O$ do
 8:         if $\Pr(o|b_i, a_i) > 0$ then
 9:             $best \leftarrow \arg\max_j \ \alpha_j(b_i^{a_i,o})$
10:             $\psi(n_i, o) \leftarrow n_{best}$
11:         else
12:             $\psi(n_i, o) \leftarrow n_i$
13: return $\langle N, \phi, \psi \rangle$
```

Algorithm 2: Policy Tree Generation

```
POLICYTREE($\pi, b, depth$)
 1: $N \leftarrow \emptyset$
 2: $j \leftarrow 1$
 3: $queue \leftarrow \{\langle b, 0, j \rangle\}$
 4: while $\neg isEmpty(queue)$ do
 5:     $\langle b, d, i \rangle \leftarrow removeFirst(queue)$
 6:     $N \leftarrow N \cup \{n_i\}$
 7:     $\phi(n_i) \leftarrow \pi(b)$
 8:     if $d = depth$ then
 9:         $\psi(n_i, o) \leftarrow * \ \forall o \in O$
10:     else
11:         for all $o \in O$ do
12:             $j \leftarrow j + 1$
13:             $addLast(queue, \langle b^{\phi(n_i)o}, d+1, j \rangle)$
14:             $\psi(n_i, o) \leftarrow n_j$
15: return $\langle N, \phi, \psi \rangle$
```

3.2 Compiling Controllers from Arbitrary Policies by Simulation

We describe an approach to compile arbitrary policies into approximately equivalent controllers. The approach simulates the policy up to a certain depth and ensures that the controller will execute the same actions up to that depth. In the limit, with an infinite depth, we obtain a controller that matches the policy exactly. Although, as we show in the experiments, we can often obtain a controller that is at least as good by simulating up to a reasonable depth.

The approach works in two steps: i) first we generate a policy tree up to a certain depth, then ii) we compress the policy tree into a controller by detecting matching subtrees. Alg. 2 shows how to generate a policy tree up to a certain depth by simulating the policy. Since simulation does not require the policy to be in any format, the approach works with arbitrary policies. We just need to generate the next action given the current observation at each time step, which is always possible since this is how all policies are executed in practice. To be concrete, Alg. 2 shows how to generate a policy tree for policies that are belief mappings, but we could easily modify the algorithm to work with policies that are represented as history mappings or any other type of mapping. The algorithm generates a policy tree in breadth first order, which will become handy in the compression step. Since leaves do not have edges, we set $\psi(n, o)$ to $*$ for all edges rooted at a leaf n (Line 9).

Fig. 1 shows the policy tree generated by Alg. 2 up to a depth of 5 for the classic tiger problem [19]. In this problem there are three actions (listen, open-right and open-left), two observations (tiger-right, tiger-left). Nodes are labeled with actions and edges are labeled with observations. Nodes are also numbered according to the breadth-first order in which they were generated.

In the second step, the policy tree is compressed into a controller by identifying matching conditional plans. Each node of the policy tree is the root of a conditional plan. Conditional plans rooted at each node are compared to conditional plans rooted at previous nodes in the breadth-first order. When two conditional plans match, we replace the node with highest breadth-first index by the node with the lowest breadth-first index. Two conditional plans are said to match when they select the same actions in each path up until a leaf is encountered. Hence, conditional plans with different depths can still match since we stop the verification as soon as a leaf is encountered in a path. Alg. 3 shows how to verify whether two conditional plans match. Alg. 4 uses this verification procedure to prune nodes whose conditional plans match the conditional plan of an earlier node in the breadth-first order. This process gives rise to a controller that is often much smaller than the original policy tree and yet ensures that the same actions are executed up to the depth of the original policy tree.

Fig. 2 shows again the policy tree for the tiger problem with additional dashed edges indicating that the parent node is replaced by the child node due to matching conditional plans. For instance, node 4 will be replaced by node 0 since their conditional plans match. Fig. 3 shows the resulting reduced controller once all node substitutions indicated by dashed edges in Fig. 2 are performed. Since leaf nodes have a trivial one-step conditional plan and they are last in the

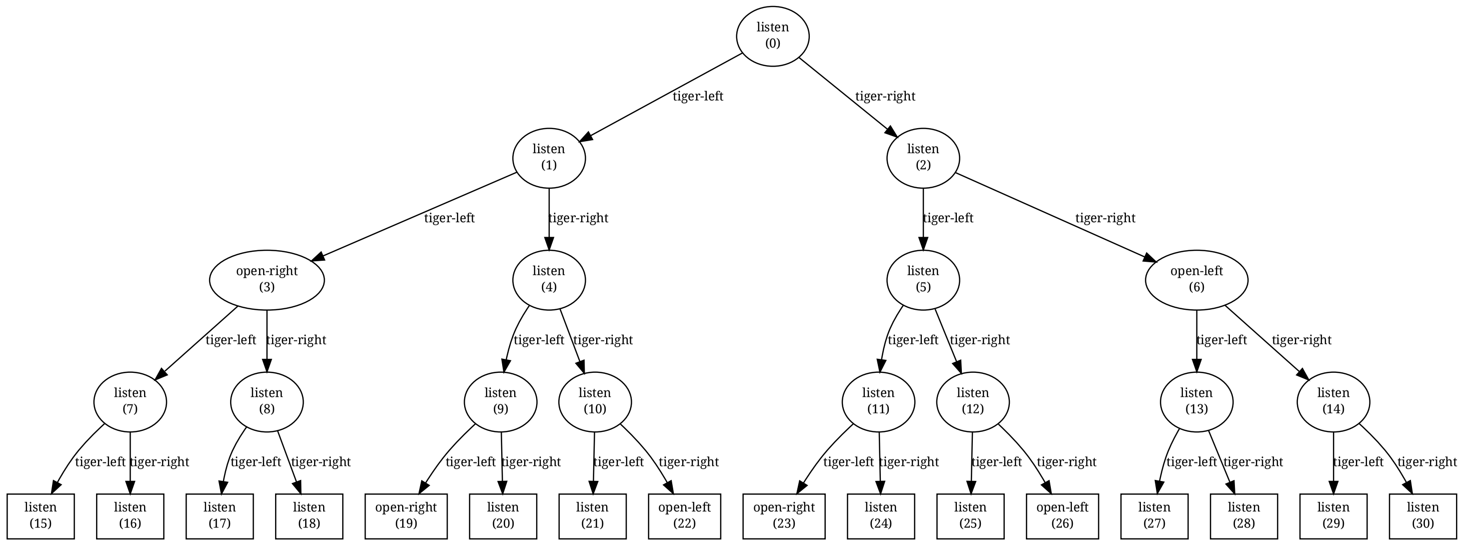

Fig. 1: Policy tree up to a depth of 5 for the classic tiger problem

Algorithm 3: Equivalent Conditional Plans

```
EQUIVALENTCP($n_1, n_2, \phi, \psi$)
1: if $\phi(n_1) \neq \phi(n_2)$ then
2:   return $false$
3: for all $o \in O$ do
4:   if $\psi(n_1, o) \neq *$ and $\neg$EQUIVALENTCP($\psi(n_1, o), \psi(n_2, o), \phi, \psi$) then
5:     return $false$
6: return $true$
```

Algorithm 4: Compilation of arbitrary π into approx. equivalent controller $\langle N, \phi, \psi \rangle$

```
POLICY2FSC($\pi, b, depth$)
1: $\langle N, \phi, \psi \rangle \leftarrow$ POLICYTREE($\pi, b, depth$)
2: for all $n_i \in N$ in increasing index $i$ do
3:   for all $n_j \in N$ such that $j < i$ do
4:     if EQUIVALENTCP($n_i, n_j, \phi, \psi$) then
5:       $N \leftarrow N \setminus \{n_i, descendents(n_i)\}$
6:       for all $n \in N, o \in O$ do
7:         if $\psi(n, o) = n_i$ then
8:           $\psi(n, o) \leftarrow n_j$
9: return $\langle N, \phi, \psi \rangle$
```

breadth-first order, they will be replaced by interior nodes as long as there is an interior node with the same action. Since actions eventually repeat in a large enough tree, the compilation procedure generally produces controllers without leaves (i.e., all nodes have a full set of edges). The breadth-first order also ensures that Alg. 3 terminates since in each pair of conditional plans that we compare, the one rooted at the node with the highest index is necessarily a tree of finite depth (i.e., no loop). In addition, when we replace the node with the highest index we can delete the entire subtree below it since there is no way to reach that subtree other than through the node that is being replaced. This pruning greatly improves the running time. Finally, the breadth first order also helps to produce a small controller since nodes are always replaced by nodes with a lower index and therefore earlier in the tree.

The complexity of Alg. 4 is quadratic in the size of the policy tree. However, due to the pruning of subtrees each time a node is replaced, we can show that the complexity is really linear in the size of the policy tree times the size of the reduced controller. The experiments show that the reduced controller is often significantly smaller than the policy tree, yielding a substantial speed up. That being said, the linear dependence on the size of the policy tree is still significant since the size of policy trees is exponential in the depth (i.e., $O(|O|^{depth})$). We can often reduce the base $|O|$ of the exponential by exploiting sparsity or considering only observations with a probability greater than some threshold.

4 Controller Compression

Once a policy is compiled to a controller, it often contains redundant or dominated nodes. Redundant nodes often occur when some observations have zero probability, leading to multiple conditional plans with the same value. Dominated nodes often occur when the original policy is suboptimal and the compilation process generates suboptimal conditional plans. We describe a technique to compress a controller while ensuring that its value does not decrease and in some cases it increases. The idea is to prune all nodes with α-vectors that are dominated in value by other α-vectors. This approach was first used by Hansen in his policy iteration algorithm [14]. Below, Algorithm 5 describes how to re-

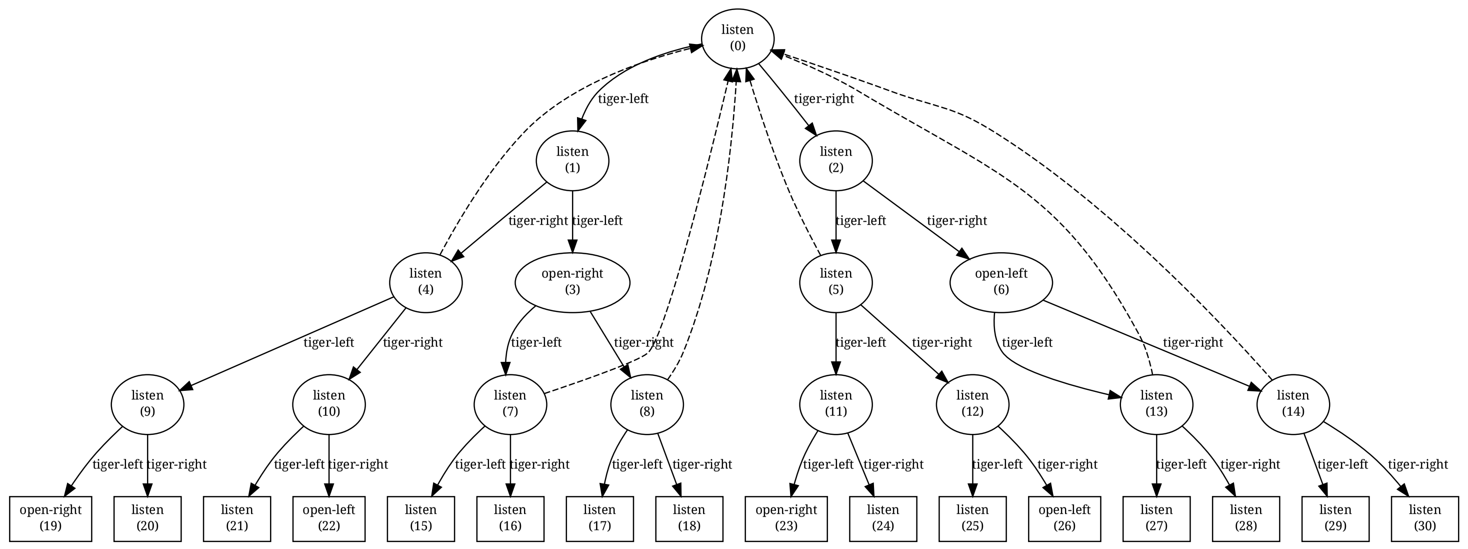

Fig. 2: Policy tree up to a depth 5 for the classic tiger problem with dashed edges indicating nodes whose conditional plans match according to Alg. 3

Algorithm 5: FSC Compression

```
fscCompression($N, \phi, \psi$)
 1: repeat
 2:   Eval controller by solving (2)
 3:   for each $n_1 \in N$ do
 4:     for each $n_2 \in N \setminus \{n_1\}$ do
 5:       if $\alpha_{n_1}(s) \leq \alpha_{n_2}(s)\ \forall s$ then
 6:         $N \leftarrow N \setminus \{n_1\}$
 7:         for all $n \in N, o \in O$ do
 8:           if $\psi(n, o) = n_1$ then
 9:             $\psi(n, o) \leftarrow n_2$
10:         break
11:       if $\alpha_{n_1}(s) \geq \alpha_{n_2}(s)\ \forall s$ then
12:         $N \leftarrow N \setminus \{n_2\}$
13:         for all $n \in N, o \in O$ do
14:           if $\psi(n, o) = n_2$ then
15:             $\psi(n, o) \leftarrow n_1$
16: until $N$ doesn't change
17: return $\langle N, \phi, \psi \rangle$
```

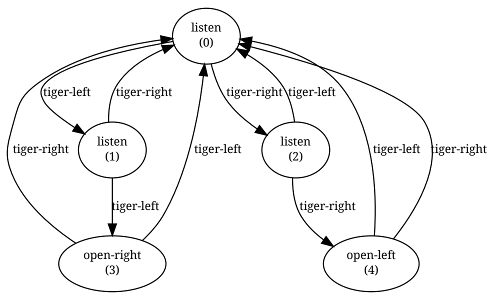

Fig. 3: Controller obtained by reducing a 5-step policy tree according to Alg. 4 for the classic tiger problem.

peatedly compress a controller until there are no dominated nodes. The approach alternates between policy evaluation and node substitution. The evaluation step computes the α-vector of each node by solving a system of linear equations. Then the α-vector of each node is compared to the α-vectors of the other nodes. When $\alpha_1(s) \leq \alpha_2(s)\ \forall s$ then n_1 can be replaced by n_2. Since the value of n_2 is at least as good as that of n_1 in all states, then pruning n_1 and replacing it by n_2 does not lower the value of the controller. The value will go up if there is an s such that $\alpha_2(s) > \alpha_1(s)$. The complexity of the policy evaluation step in Alg. 5 is $O(|N|^3|S|^3|O|)$, however sparsity often allows to reduce the dependence on $|S|$ and $|O|$. The complexity of the pruning step is $O(|N|^2|S|)$. Overall, compression time is a small fraction of compilation time.

5 Experiments

We evaluate our methods using policies computed by two state-of-the-art point-based POMDP algorithms: GapMin [12] and SARSOP [13]. GapMin returns $\langle \alpha_i, b_i, a_i \rangle$-tuples and therefore we can compile its policies into finite-state controllers using both of our methods. SARSOP was used to compute policies for the largest POMDP benchmarks, however it returns only α-vectors, which is sufficient to apply policy2fsc, but not alpha2fsc (witness beliefs are also needed, but SARSOP's interface does not expose them). The experiments are conducted with some benchmark problems and a real-world POMDP for smart phones. The running time of compilation algorithms—reported in the column 'time'—corresponds to the time of actual compilation. The time to compute initial policies before compilation can be found in the columns 'GapMin' or 'SARSOP' depending on which solver was used. High time limits were selected in order to compute policies of high quality. Thus, this time could be considerably shorter if

POMDP	GapMin	method	depth	tree size	nodes	value	time	c
4x5x2.95	GM-lb=2.08	alpha2fsc			47(58)	2.08(2.08)	0.29	2
$\lvert S\rvert$=39, $\lvert A\rvert$=4	GM-ub=2.08	**GM-LB**	8	287	10(17)	**2.08**(1.85)	0.30	1
$\lvert O\rvert$=4, $\gamma = 0.95$	time=4.96s	**GM-UB**	8	287	10(17)	**2.08**(1.85)	0.29	1
	$\lvert lb\rvert$=58	B&B			5	2.02	639.9	
	$\lvert ub\rvert$=243	EM			10	2.01 ± 0.02	66.8	
		QCLP			10	1.74 ± 0.11	7.7	
		BPI			8	0.71 ± 0.09	0.72	
aloha.10	GM-lb=533.4	alpha2fsc			137(158)	533.2(533.2)	11.5	1
$\lvert S\rvert$=30, $\lvert A\rvert$=9	GM-ub=544.1	**GM-LB**	11	29525	390(1116)	**537.6**(537.5)	83.6	2
$\lvert O\rvert$=3, $\gamma = 0.999$	time=5223s	GM-UB	11	29525	402(1148)	537.6(537.6)	94.5	1
	$\lvert lb\rvert$=158	B&B			10	529.0*	24h	
	$\lvert ub\rvert$=406	EM			40	534.8 ± 0.25	2739	
		QCLP			25	534.37 ± 0.52	99.2	
		BPI			5	112.4 ± 1.59	0.69	
chainOfChains3	GM-lb=157	**alpha2fsc**	10	10	10(10)	**157**(157)	0.26	0
$\lvert S\rvert$=10, $\lvert A\rvert$=4	GM-ub=157	**GM-LB**	11	11	10(10)	**157**(157)	0.42	0
$\lvert O\rvert$=1, $\gamma = 0.95$	time=0.86s	**GM-UB**	11	11	10(10)	**157**(157)	0.26	0
	$\lvert lb\rvert$=10	**B&B**			10	**157**	1.69	
	$\lvert ub\rvert$=1	EM			10	0.17 ± 0.06	6.9	
		QCLP			10	0 ± 0	0.16	
		BPI			10	25.7 ± 0.77	4.25	
cheese-taxi	GM-lb=2.481	**alpha2fsc**			17(22)	**2.476**(2.476)	0.29	1
$\lvert S\rvert$=34, $\lvert A\rvert$=7	GM-ub=2.481	**GM-LB**	15	167	17(24)	**2.476**(2.476)	0.56	1
$\lvert O\rvert$=10, $\gamma = 0.95$	time=1.88s	**GM-UB**	15	167	17(24)	**2.476**(2.476)	0.55	1
	$\lvert lb\rvert$=22	B&B			10	-19.9*	24h	
	$\lvert ub\rvert$=13	EM			17	-12.16 ± 2.08	337.9	
		QCLP			17	-18.22 ± 1.77	227.4	
		BPI			16	-18.1 ± 0.39	7.18	
lacasa2a	GM-lb=6714.6	alpha2fsc			5(5)	6714.0(6714.0)	5.15	0
$\lvert S\rvert$=320, $\lvert A\rvert$=4	GM-ub=6717.6	**GM-LB**	5	22621	106(421)	**6715.0**(6715.0)	933.9	1
$\lvert O\rvert$=12, $\gamma = 0.95$	time=54s	GM-UB	5	22621	100(517)	6714.1(6714.1)	256.4	1
	$\lvert lb\rvert$=5	B&B			3	6710.0	493.8	
	$\lvert ub\rvert$=14	EM			11	6710 ± 0.11	6485	
		QCLP			2	6699.9 ± 5.5	181	
		BPI			26	6709.3 ± 0.2	121.5	
lacasa3.batt	GM-lb=293.4	alpha2fsc			25(26)	292.4(292.4)	399.7	1
$\lvert S\rvert$=1920, $\lvert A\rvert$=6	GM-ub=294.7	GM-LB	4	12601	47(60)	293.1(292.7)	1451	2
$\lvert O\rvert$=36, $\gamma = 0.95$	time=5386s	**GM-UB**	4	12697	41(48)	**293.2**(293.1)	1030	2
	$\lvert lb\rvert$=26	B&B			5	287.0*	24h	
	$\lvert ub\rvert$=48	**EM**			5	**293.2** ± 0.03	13331	
		BPI			9	**293.2** ± 0.12	2102	
lacasa4.batt	GM-lb=291.1	alpha2fsc			10(10)	285.5(285.5)	302	0
$\lvert S\rvert$=2880, $\lvert A\rvert$=6	GM-ub=292.6	GM-LB	3	745	19(22)	287.3(287.1)	3652	1
$\lvert O\rvert$=72, $\gamma = 0.95$	time=8454s	**GM-UB**	4	23209	87(94)	**290.8**(290.8)	3681	1
	$\lvert lb\rvert$=10	B&B			10	285.0*	24h	
	$\lvert ub\rvert$=23	EM			3	290.2 ± 0.0	19920	
		BPI			6	290.6 ± 0.2	4124	
hhepis6obs_woNoise	GM-lb=8.64	**alpha2fsc**			14(18)	**8.64**(8.64)	0.49	1
$\lvert S\rvert$=20, $\lvert A\rvert$=4	GM-ub=8.64	**GM-LB**	12	21	14(18)	**8.64**(8.64)	0.89	1
$\lvert O\rvert$=6, $\gamma = 0.99$	time=2.6s	**GM-UB**	12	21	14(18)	**8.64**(8.64)	0.74	1
	$\lvert lb\rvert$=18	**B&B**			8	**8.64**	4.48	
	$\lvert ub\rvert$=7	EM			14	0.0 ± 0.0	49.2	
		QCLP			14	0.16 ± 0.10	26	
		BPI			13	0.0 ± 0.0	1.68	
machine	GM-lb=62.38	alpha2fsc			5(39)	54.61(54.09)	5.53	1
$\lvert S\rvert$=256, $\lvert A\rvert$=4	GM-ub=66.32	GM-LB	9	376	26(41)	62.92(62.84)	18.5	1
$\lvert O\rvert$=16, $\gamma = 0.99$	time=3784s	**GM-UB**	12	2864	11(159)	**63.02**(60.29)	86.8	2
	$\lvert lb\rvert$=39	B&B			6	62.6	52100	
	$\lvert ub\rvert$=243	EM			11	62.93 ± 0.03	1757	
		QCLP			11	62.45 ± 0.22	4636	
		BPI			10	35.7 ± 0.52	2.14	

Table 1: Compilation of GapMin policies using: (1) alpha2fsc applied to GapMin lower bound alpha vectors. (2) policy2fsc applied to GapMin lower bound policy (GM-LB) and (3) policy2fsc applied to GapMin upper bound policy (GM-UB).

one stops the planning algorithms as soon as a policy of sufficient quality is obtained. This could lead to a substantial reduction of the planning/initialization time since longer planning times (e.g., 10^4 seconds instead of 10^3 seconds in the case of SARSOP [13]) do not usually lead to dramatically improved policies.

5.1 LaCasa Domain

We tested our approaches on three instantiations of the LaCasa domain [20, 11], which is a real-world planning task where a smart phone estimates the risk of wandering by a dementia patient and when necessary assists the patient with wayfinding or calls a caregiver. In this domain, it is particularly important to minimize energy consumption since the smart phone won't be able to assist the patient once the battery runs out. Offloading computation to a cloud service is not desired either since it requires a data connection (which may not always be available or reliable) and wireless communication uses a non-negligible amount of energy. A controller offers the best solution since computation consists of negligible table lookups and no data connection is required.

5.2 Results

Tables 1 and 2 compare the results obtained by compiling policies produced by GapMin and SARSOP respectively to four techniques that directly optimize controllers: bounded policy iteration (BPI) with escape [8], quadratically constrained linear programming (QCLP) [7], expectation maximization (EM) with forward search [21] and branch&bound (B&B) with isomorph pruning [11]. Policy2fsc was used in an iterative deepening fashion, starting from depth 2, up to a depth where the resulting controller was at least as good as the original policy or a time limit was exceeded. Hence, the time reported for policy2fsc is the cumulative time (seconds) to process all compilations from depth 2 up to the depth reported in column depth. Tree size is the size of the policy tree for that depth (note that edges with zero probability reduce the size of the policy tree considerably). Column 'nodes' displays the number of nodes in the final controller after compression (before compression in the parentheses). Column 'value' shows the value of controllers after compression (analogously, before compression in the parentheses). Column 'c' indicates the number of iterations of the compression until there is no compression possible. The absence of any result for QCLP, EM and BPI for some problems indicates that 3Gb of memory was not sufficient. A * besides the value of B&B indicates that B&B did not complete its search in 24h and that the value reported is for the best controller found in 24h.

Table 1 compares our two compilation methods for policies computed by GapMin. Method GM-LB stands for policy2fsc applied to the GapMin lower bound policy whereas GM-UB to the upper bound policy. Results confirm that our methods are successful in compiling POMDP policies into finite-state controllers of approximately equivalent quality. The highest value found for each problem is bolded. Alpha2fsc compiles $|lb|$ α-vectors into controllers with similar value, though sometimes the value is significantly worse (e.g., lacasa4.batt and

POMDP	SARSOP	method	depth	tree size	nodes	value	time	c
baseball	time 122.7s	**policy2fsc**	7	175985	10(47)	**0.641**(0.641)	78.22	1
$\|S\|$=7681, $\|A\|$=6	$\|\alpha\|$ =1415	B&B			5	0.636*	24h	
$\|O\|$=9, $\gamma = 0.999$	UB=0.642	EM			2	0.636 ± 0.0	48656	
	LB=0.641	BPI			9	0.636 ± 0.0	445	
elevators_inst_pomdp_1	time 11,228s	**policy2fsc**	11	419	20(24)	**-44.41**(-44.41)	1357	1
$\|S\|$=8192, $\|A\|$=5	$\|\alpha\|$ =78035	B&B			10	-149.0*	24h	
$\|O\|$=32, $\gamma = 0.99$	UB=-44.31							
	LB=-44.32							
tagAvoid	time 10,073s	**policy2fsc**	28	7678	91(712)	**-6.04**(-6.04)	582.2	1
$\|S\|$=870, $\|A\|$=5	$\|\alpha\|$ =20326	B&B			10	-19.9*	24h	
$\|O\|$=30, $\gamma = 0.95$	UB=-3.42	EM			9	-6.81 ± 0.12	19295	
	LB=-6.09	QCLP			2	-19.99 ± 0.0	12.9	
		BPI			88	-12.42 ± 0.13	1808	
underwaterNav	time 10,222s	policy2fsc	51	1242	52(146)	745.3(745.3)	5308	1
$\|S\|$=2653, $\|A\|$=6	$\|\alpha\|$ =26331	B&B			10	747.0*	24h	
$\|O\|$=103, $\gamma = 0.95$	UB=753.8	**EM**			5	**749.9** ± 0.02	31611	
	LB=742.7	BPI			49	748.6 ± 0.24	14758	
rockSample-7_8	time 10,629s	**policy2fsc**	31	2237	204(224)	**21.58**(21.58)	1291	1
$\|S\|$=12545, $\|A\|$=13	$\|\alpha\|$ =12561	B&B			10	11.9*	24h	
$\|O\|$=2, $\gamma = 0.95$	UB=24.22	BPI			5	7.35 ± 0.0	78.8	
	LB=21.50							

Table 2: Compilation and compression of SARSOP policies.

machine). In contrast, policy2fsc finds better controllers by simulating the input policy to a larger depth, but this takes more time. It was stopped as soon as the value of the controller matches GapMin's lower bound or 1h was reached. In many cases, the number of nodes is still less than or equal to the size of the input policy (e.g., 4x5x2.95, cheese-taxi, lacasa2, machine). The direct optimization techniques (B&B, QCLP, EM, BPI) generally take much longer and/or do not consistently produce good controllers.

Table 2 summarizes the results for some problems that are among the largest available benchmarks for point-based value iteration techniques that do not exploit factored representations. In this case, SARSOP was used to obtain a lower bound policy that is then compiled by policy2fsc. Even though SARSOP returned value functions with thousands of α-vectors, we compiled those policies into considerably smaller controllers (up to 3 orders of magnitude reduction) of the same or better quality (e.g., underwaterNav) demonstrating that our method scales to large problems. Policy2fsc produced the best value for all problems except underwaterNav where the direct optimization techniques produced better controllers. This simply indicates that the policy compiled from SARSOP was not the best as opposed to any weakness in policy2fsc.

6 Conclusion

We have presented two novel methods for compiling policies for partially observable Markov decision processes (POMDPs) into approximately equivalent finite state controllers (FSCs). Our motivation is that these FSC representations are very useful in resource-constrained applications such as on mobile or wearable devices. Methods that can create FSC policies open up new possibilities for using POMDP controllers on these devices, where battery, computation and memory resources are at a premium. We showed how we can get very compact, yet equivalent representations for POMDP policies as those generated by two state-of-the-art offline planners.

Acknowledgments

This work was supported by the Ontario Ministry of Research and Innovation, NSERC, Toronto Rehabilitation Institute and the Alzheimer's Association grant ETAC-10-173237.

References

1. Williams, J.D., Young, S.: Scaling POMDPs for spoken dialog management. IEEE Trans. on Audio, Speech, and Language Processing **15**(7) (2007) 2116–2129
2. Hoey, J., Boutilier, C., Poupart, P., Olivier, P., Monk, A., Mihailidis, A.: People, sensors, decisions: Customizable and adaptive technologies for assistance in healthcare. ACM Transactions on Interactive Intelligent Systems **2**(4) (2012)
3. Ross, S., Pineau, J., Paquet, S., Chaib-draa, B.: Online planning algorithms for POMDPs. Journal of Artificial Intelligence Research **32** (2008) 663–704
4. Silver, D., Veness, J.: Monte-Carlo planning in large POMDPs. In: NIPS. (2010)
5. Kaelbling, L., Littman, M., Cassandra, A.: Planning and acting in partially observable stochastic domains. Artificial Intelligence **101**(1-2) (1998) 99–134
6. Braziunas, D., Boutilier, C.: Stochastic local search for POMDP controllers. In: AAAI. (2004) 690–696
7. Amato, C., Bernstein, D., Zilberstein, S.: Optimizing fixed-size stochastic controllers for POMDPs and decentralized POMDPs. JAAMAS (2009)
8. Poupart, P., Boutilier, C.: Bounded finite state controllers. In Thrun, S., Saul, L.K., Schölkopf, B., eds.: NIPS, MIT Press (2003)
9. Toussaint, M., Harmeling, S., Storkey, A.: Probabilistic inference for solving (PO)MDPs. Technical Report EDI-INF-RR-0934, School of Informatics, University of Edinburgh (2006)
10. Meuleau, N., Kim, K.E., Kaelbling, L.P., Cassandra, A.R.: Solving POMDPs by searching the space of finite policies. In: UAI. (1999) 417–426
11. Grześ, M., Poupart, P., Hoey, J.: Isomorph-free branch and bound search for finite state controllers. In: Proc. of IJCAI. (2013)
12. Poupart, P., Kim, K.E., Kim, D.: Closing the gap: Improved bounds on optimal POMDP solutions. In: ICAPS. (2011)
13. Kurniawati, H., Hsu, D., Lee, W.: SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In: RSS. (2008)
14. Hansen, E.: An improved policy iteration algorithm for partially observable MDPs. In: NIPS. (1998)
15. Pineau, J., Gordon, G., Thrun, S.: Point-based value iteration: An anytime algorithm for POMDPs. In: IJCAI. (2003) 1025–1032
16. Spaan, M.T.J., Vlassis, N.: Perseus: Randomized point-based value iteration for POMDPs. Journal of Artificial Intelligence Research **24** (2005) 195–220
17. Shani, G., Pineau, J., Kaplow, R.: A survey of point-based POMDP solvers. Journal of Autonomous Agents and Multi-Agent Systems **27**(1) (2013) 1–51
18. Hansen, E.A.: Finite-memory control of partially observable systems. PhD thesis, University of Massachusetts Amherst (1998)

19. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial Intelligence **101** (1998) 99–134
20. Hoey, J., Yang, X., Quintana, E., Favela, J.: LaCasa: Location and context-aware safety assistant. In: Pervasive Comp. Techn. for Healthcare. (2012) 171–174
21. Poupart, P., Lang, T., Toussaint, M.: Analyzing and escaping local optima in planning as inference for partially observable domains. In: ECML/PKDD (2). (2011) 613–628