Deriving Explicit Control Policies for Markov Decision Processes Using Symbolic Regression

A. Hristov
Center for Mathematics and Computer Science,
Stochastics Group
asparuhvh@gmail.com

J.W. Bosman
Center for Mathematics and Computer Science,
Stochastics group
joost.bosman@gmail.com

S. Bhulai
Vrije Universiteit Amsterdam,
Department of Mathematics
s.bhulai@vu.nl

R.D. van der Mei
Center for Mathematics and Computer Science,
Stochastics group
mei@cwi.nl

ABSTRACT

In this paper, we introduce a novel approach to optimizing the control of systems that can be modeled as Markov decision processes (MDPs) with a threshold-based optimal policy. Our method is based on a specific type of genetic program known as symbolic regression (SR). We present how the performance of this program can be greatly improved by taking into account the corresponding MDP framework in which we apply it.

The proposed method has two main advantages: (1) it results in near-optimal decision policies, and (2) in contrast to other algorithms, it generates *closed-form approximations*. Obtaining an explicit expression for the decision policy gives the opportunity to conduct sensitivity analysis, and allows instant calculation of a new threshold function for any change in the parameters. We emphasize that the introduced technique is highly general and applicable to MDPs that have a threshold-based policy. Extensive experimentation demonstrates the usefulness of the method.

CCS CONCEPTS

• **Mathematics of computing → Stochastic processes**.

KEYWORDS

Markov Decision Processes, Genetic program, Symbolic regression, Threshold-type policy, Optimal control, Closed-form approximation

ACM Reference Format:

A. Hristov, J.W. Bosman, S. Bhulai, and R.D. van der Mei. 2020. Deriving Explicit Control Policies for Markov Decision Processes Using Symbolic Regression. In *13th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '20), May 18–20, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3388831.3388840

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

VALUETOOLS '20, May 18–20, 2020, Tsukuba, Japan

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7646-4/20/05…$15.00
https://doi.org/10.1145/3388831.3388840

1 INTRODUCTION

In practice, many control problems exhibit an optimal policy that is of a threshold type. In most cases, this structure can be shown theoretically by monotonicity and submodularity arguments combined with mathematical induction within the framework of Markov decision processes (MDPs). In real applications, however, one does not only need to know the structure of the optimal policy, but also the threshold value for implementation purposes. Unfortunately, deriving this value *explicitly* remains a hard problem and has to be solved by numerical computation.

There are many advantages of having the threshold value in an explicit form in a practical setting. It allows one to easily implement a threshold for different system parameters without having to resolve the MDP. This becomes even more relevant in an environment with time-varying parameters. Moreover, the robustness of the threshold function can be assessed through sensitivity analysis. This is important when system parameters are estimated from data.

In this paper, we develop a new approach to obtain an analytic expression for threshold-based policies in MDPs. The main idea of our method is to combine the field of Markov decision theory with a specific genetic program, namely, the symbolic regression (SR) algorithm, to learn the threshold function. We outline guidelines and useful practices when tailoring the algorithm. Although the obtained solution might not be the optimal one, the threshold-based decision policy is, nevertheless, given in a closed-form expression and near-optimal.

An introduction to MDPs and the well-known numerical techniques to solve such problems (e.g., value iteration and function iteration) are described in [10, 13, 18]. As discussed, next to the numerical approach, one might tackle the challenge of optimal control also by using algebraic techniques. However, due to the complexity of most of the MDP problems, obtaining an algebraic solution is not feasible. Therefore, the vast body of literature (e.g., [3, 6, 14]) deals with proving structural properties of various Markov decision problems rather than finding the explicit structure of the decision policy.

On the other hand, mostly due to practical reasons, there is a need for an efficient procedure that yields an MDP solution, which can be implemented afterward. There are several papers that show how one can make use of machine learning techniques to obtain such a solution. Most of the research in this domain is focusing

on one of the following two types of algorithms: reinforcement learning [16, 17], or genetic programs [1, 4, 9, 19].

We believe that our paper can serve as a link between the above described two major, and fundamentally different, approaches. Our technique exploits certain structural properties of the given MDP to produce a closed-form solution. Namely, it requires the optimal policy to be of a threshold type. To obtain the function characterizing this decision policy, we use SR. An introduction to this genetic program can be found in [5, 7]. A related research that applies SR within an MDP framework is the one conducted in [11]. However, in contrast to [11], our research aims at finding the control policy rather than the value function. Therefore, with our approach one can directly incorporate insights into the optimal policy in the SR implementation.

As an additional remark, we note that the results in this paper were obtained using the `gplearn` [15] Python package, which provides an SR implementation. Its efficiency together with the `scikit-learn` [12] inspired and compatible API, made this package our choice for representative SR solution.

The remainder of this paper is organized as follows. We first outline our implementation of the SR method for solving a given MDP with a threshold-based policy in Section 2. To present our guidelines more comprehensibly, we introduce in Section 3 an MDP that will serve as a running example. In Sections 4 and 5, we discuss our findings and our approach in the following two crucial procedures: preparing the data and adjusting the settings of the algorithm. We evaluate our threshold function discovery technique in two ways: 'How does the approximated policy perform compared to the optimal one?' and 'How much does the generated expression resemble the real closed-form solution?' We conclude with a summary and discussion in Section 8.

2 OUTLINE OF THE TECHNIQUE

In this section, we introduce our method of finding a closed-form solution for the control policy of a given MDP. As mentioned, our technique uses SR. Therefore, we briefly present the main concepts of this regression algorithm. The reader is referred to [5, 7, 15] for more details on SR.

The goal of SR is to find an algebraic expression that best fits a given dataset. Like any other genetic program, SR forms an initial population of individuals, which in this case represent mathematical formulas. Next, it iteratively generates a new offspring of individuals (e.g., a new generation) by combining and/or mutating already existing individuals. The underlying idea is that over time the population's accuracy increases due to evolving the good performing individuals. Within the SR framework, an individual represents a specific formula, which is expressed as a tree (for an example we refer to Fig.1). Note that each leaf contains a parameter, whereas each node gives the mathematical operator. In Fig. 2, we illustrate one of the few possible combination/mutation schemes.

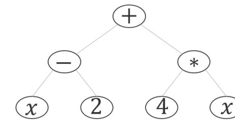

Figure 1: The formula $(x - 2 + 4x)$ expressed as a tree.

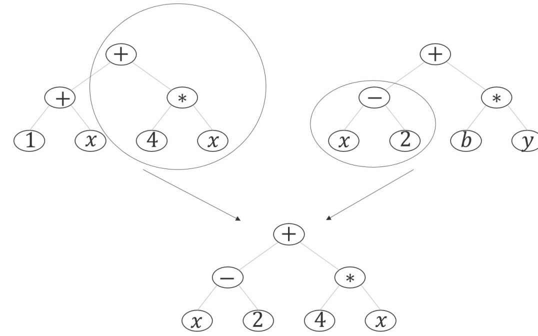

Figure 2: An example of generating a new individual by a combination of two others.

In the following, we list the four main steps of our technique.

Step 1: Modeling an MDP with a threshold-based policy: as a first step, one should model the problem as an MDP by defining the system states and the corresponding transition probabilities, associated operational costs and possible control decisions. Note that the studied MDP should have an optimal policy which is of a threshold type.

Step 2: Preparing the data: the regression program requires a dataset on which the individuals will be tested. Therefore, one needs to numerically solve a number of system instances, e.g., to obtain the optimal decision policy with regards to specific parameter values.

Step 3: Specifying the algorithm settings: there are many settings that determine the duration of the evolution, the initial population, and perhaps most importantly the way the generations evolve.

Step 4: Evaluating the results: different setting configurations lead to different results. There are certain choices that we believe are optimal, but nevertheless, we advise one to iteratively analyze the obtained expressions and further adjust certain settings that might lead to a better result.

To evaluate the performance of the algorithm, we examine two MDP models. The first one defines a system for which the analysis is very challenging and, to the best of our knowledge, there is still no efficient technique for obtaining the optimal policy. Furthermore, there is no analytic solution available even for specific cases of this system. Therefore, we take the corresponding MDP model as a running example and as a benchmark for evaluating the algorithm's accuracy in terms of achieved system performance. In contrast, the optimal policy for our second MDP example, namely, an $M|M|1$ queue, can be derived in closed-form. Hence, by comparing the expressions generated by our technique to the optimal formula, we can study how well the algorithm approximates the algebraic form of the threshold function.

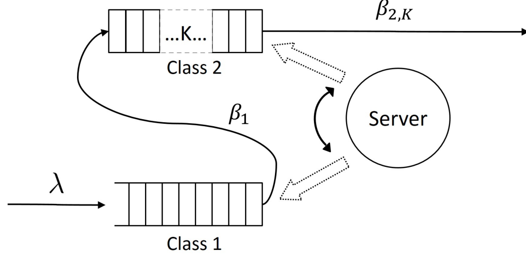

Figure 3: The 'write behind' mechanism as a queueing model.

3 MODELING AN MDP WITH A THRESHOLD-BASED POLICY

In this section, we present our first MDP problem. We consider a single-server queuing system with features inspired by the 'write behind' caching mechanism (see Fig. 3). Namely, the server has to complete a two-step process for each request that arrives in the system. Jobs receive an initial service in a First Come First Served (FCFS) fashion and are subsequently accumulated in a buffer. The second phase consists of serving those requests that are in the buffer as a batch, i.e., perform a flush of the cache. In such a way, the server can accumulate requests in the buffer to some level before serving them all together as a group.

In the following, we denote the maximum size of the server queue and the buffer as Q and B, respectively. Next to that, jobs are assumed to arrive according to a Poisson process with rate λ and join the queue if they find the server busy at this moment. Furthermore, the time to store a request in the buffer is taken to be exponentially distributed, with mean β_1. This corresponds to the required time to write a request to the cache. The service time required for the second phase is also assumed to be exponentially distributed, with mean $\beta_{2,K}$, and stands for the time required to write to the cache and consequently perform a flush of size K. We model $\beta_{2,K}$ to increase proportionally to the batch size and therefore we take $\beta_{2,K} = a + bK$, where a and b are parameters. One can interpret $a \geq \beta_1$ as the time required to write the K-th request to the cache and subsequently initialize the flush, i.e., the batch service. On the other hand, $b > 0$ stands for the average time to write a single update to the storage. As the flush requires an initialization time, accumulating jobs and serving them at once might greatly improve the overall performance by reducing the number of initialization steps. On the other hand, the bigger the group size is, the larger the waiting times during the batch service will be. This implies that requests can be grouped to an optimal level. Motivated by this, we analyze the control policy that minimizes the average waiting time.

In addition, we note that the system load, ρ, is dependent on the batch size and hence the decision policy. Therefore, in the following as an approximation of ρ we use the case of a batch size 1. Hence, we take $\rho = 0.5\lambda(\beta_1 + a + b)$.

Optimizing the above described queuing system has proven to be challenging. Due to the complexity of the problem [8], there are studies on the simpler case of a static system control - the requests in the buffer are served whenever they reach a predefined number K, regardless of the number of jobs at the server queue. However, even in this case, there is still no analytic solution. Therefore, we believe that finding an analytic expression for the dynamic threshold policy would greatly facilitate managing such systems. At the same time, it promises a far better performance than the static one as it takes into account also the number of requests at the server queue.

4 PREPARING THE DATA

As discussed, we use SR to derive an expression for the threshold policy function for a given MDP. To produce an estimate, the regression needs training data set as an input. Once trained on the corresponding samples, the approximation's accuracy can be obtained by comparing the predictions on a given set of test samples with the actual, real values.

There are a few ways to solve an instance of a given MDP problem, e.g., by running the value iteration technique [13, 18] or by Temporal Difference (TD) learning [16]. Once the optimal threshold policy is obtained, one can transform it into a function $f(P_s, x) = y$, where P_s denotes the specific system parameters and y gives the corresponding threshold level for x. Note that for an MDP with an N-dimensional state space the vector x would be $(N-1)$-dimensional and y an integer. In our example, x is an integer and stands for the number of jobs at the server queue, whereas y is the number of requests that are already in the buffer. Next to that, we believe that in addition to the initial core parameters of the system (e.g., the local variables), one might also consider including self-composed ones (e.g., structured features). In such a way, one can facilitate the algorithm in discovering important dependencies between the parameters. For example, in most of the cases, the system load, ρ, greatly influences the behavior of the MDP, and hence, the optimal control. Therefore, in some of the generated SR instances we include ρ as a structured feature.

In the following, we outline the procedure we have performed to generate the data set for our running example. Our objective is to obtain approximation that can be used for a system with any parameters set, $P_s = (\lambda, \beta_1, a, b)$. Therefore, we design model instances with the idea of generating samples for systems as diverse as possible: $\beta_1 \in \{1.2, 2.4, 3.6, \ldots, 12\}$, $a/\beta_1 \in \{1, 1.4, 1.8, \ldots, 5\}$, and $b/\beta_1 \in \{0.01, 0.014, 0.18, \ldots, 0.5\}$. In all cases we take $\lambda = 1$ to reduce the number of parameters without loss of generality. In such a way, we produce examples of systems with a load ranging from 0.1 to 0.9. Note that for a given set of system parameters, P_s, we have $Q + 1$ samples in the data. Namely, one for each $0 \leq x \leq Q$ (the number of requests at the queue cannot exceed the queue length) together with the corresponding threshold value y indicating the optimal batch size.

Next to that, in the analyzed MDP system, scaling the arrival rate and the service rates does not influence the optimal decision policy. Therefore, we incorporate this insight by multiplying/dividing $(1/\lambda, \beta_1, a, b)$ by a factor of 100, append the already derived x and y and use the result as additional data samples. In such a way, we assist SR in finding an expression that is scale-free with regards to $(1/\lambda, \beta_1, a, b)$, and therefore less probable to be over-fitting the specific range of training values.

For the majority of cases, we took $Q = 100$ and $B = 50$, so that we could solve the above-described systems with the value iteration technique in terms of seconds. Nevertheless, we did also generate samples with $Q = 1000$ and $B = 500$ and included them in the test set. As a final remark, we note that we have split the obtained dataset in a training and test set by a 70/30 ratio, including MDP instances with various loads in both subsets.

5 SPECIFYING THE ALGORITHM SETTINGS

Once the MDP problem is modeled, and a dataset is generated by solving instances of this problem, one needs to specify the desired algorithm settings. An extensive list of the possible settings for the `gplearn`'s SR implementation is described in the corresponding package documentation (see [15]). Therefore, in this paper, we focus only on the features that we find particularly interesting with regards to our technique, i.e., when one uses the genetic program within an MDP framework. Namely, settings that one should consider adjusting in a way different than the default one are listed in the following subsections.

5.1 Set of operators

The set of operators contains a list of the mathematical operators that are allowed in building and evolving the trees. Due to a trade-off between the complexity of the formulas and their accuracy, our approach is to start with the four basic binary operations – addition, subtraction, multiplication, and division. Next to that, we believe that often the threshold function might contain square root operation, inverse function and/or logarithm. Therefore, we suggest running a few SR instances where trees can use various subsets of those mathematical operators. Comparing the outcomes of the configurations one can decide which results suit better one's goal. Note that there is no benefit in implicitly allowing exponentiation if one does not expect an exponent higher than 2, as the trees $exponent(a, 2)$ and $multiply(a, a)$ have the same depth and length.

5.2 Initial depth and parsimony coefficient

One can control the length and the depth of the trees, i.e., the complexity of the expressions, by adjusting the *initial depth* (*init_d*) and the *parsimony coefficient* (*pc*). More precisely, the *init_d* is given by a tuple that defines the minimum and the maximum size allowed for the first generation of trees, whereas the *pc* is influencing how the further generations evolve by penalizing longer expressions. In such a way, one can control the "bloating" effect – an increase in the trees' size which corresponds to a not significant increase in their fitness. Greater values of the *pc* penalize larger trees more and make them less favorable for selection. In our MDP example, we generated instances with values for the *pc* between 0.001 and 0.1 and *init_d* in ranges varying from as low as $(2, 3)$ to the higher values of $(7, 8)$. Based on the conducted tests, we believe that setting the *pc* to 0.01 in combination with *init_d* range containing the number of system parameters, $|P_s|$, would be a reasonable default choice for our technique. In particular, we conclude that for our MDP problem $init_d = (3, 6)$ tends to produce the best results as the first generation consists of relatively simple expressions that nevertheless are complex enough to capture all system parameters.

5.3 Fitness function

The fitness of a given tree is calculated based on the accuracy of its predictions on the training samples. This implies that there are two important components of the *fitness function* – the way the error on a single training sample is defined and the way the overall fitness is obtained from those accuracy scores. Note that if one's aim is to produce the best fit of the real threshold function solely, one can simply apply one of the common accuracy measurement metrics, e.g., mean absolute error or mean squared error. In contrast, in the following guidelines, we assume that the primary goal of a person using our technique is to minimize the costs associated with a certain MDP problem. In such a case, we find it crucial to assign appropriate weights to the various training samples. The reason is that the more probable a system state is, the greater impact an error in the corresponding decision has. Therefore, one might consider using the steady-state probability of each state as its weight. However, in some systems, it is difficult to obtain the steady-state distribution. As a consequence, in our running example, we tested the two simpler weight functions: ρ^x and $(Q - x)^2$, where ρ is the load of the system from the corresponding sample (P_s, x, y). Note that the second function does not require any additional computations as the maximum queue length Q is given as a system parameter.

Next to that, we believe that in the context of estimating a control policy of a threshold-type one should consider fitness function that computes the relative error on a given sample instead of the absolute one. In conclusion, we recommend the weighted version of a relative accuracy measurement, e.g., weighted mean absolute percentage error (wMAPE) or weighted root mean squared error (wRMSE).

6 EVALUATING THE RESULTS

As discussed, the output of the algorithm depends on the specific settings. Therefore, one has to decide whether it is possible to further adjust one or more of those features to produce an expression that fits better one's goal – a simpler threshold function or a more accurate one. Nevertheless, we believe that certain settings result in both better performing and less complex threshold decision policies. For that reason, next to the provided setting recommendations, we advise that one initially explores a larger number of various SR configurations and only afterward further evolve a few of the best ones. In such a case, it might become important to optimize the running time of each algorithm's instance. One way to achieve this is to train and/or test the first couple of configurations only on a subset of the corresponding data instead of the full one. Next to that, one can keep track of the best fitness score for each generation and terminate the instance earlier if there is not much of an increase in the score for a few generations in a row.

In the following, we discuss the accuracy of our technique for our running example. We configure the SR algorithm in accordance with the guidelines that we described in the previous sections. In such a way, we incorporate the insights from analyzing the system as an MDP model. To evaluate the added value of our approach we compare it to SR instances with default settings. Next to that, we also include cases where only part of our recommendations were implemented. In Table 1, we list some of these instances, numbered

Table 1: Setting configurations

N	Operators	ρ feature?	Fitness	Weights
I	$+, -, *, /$	No	RMSE	
II	$+, -, *, /$	No	wMAPE	ρ^x
III	$+, -, *, /$	Yes	wMAPE	ρ^x
IV	$+, -, *, /$	Yes	wRMSE	ρ^x
V	$+, -, *, /$	Yes	wRMSE	$(Q-x)^2$
VI	$+, -, *, /, \surd$	Yes	wRMSE	$(Q-x)^2$
VII	$+, -, *, /, \surd$	Yes	wRMSE	$(Q-x)^2$

Table 2: Numerical results

N	Threshold function	E_r perc. 50$^{\text{th}}$	E_r perc. 95$^{\text{th}}$
I	$\frac{3a\lambda-0.37}{\lambda\beta_1} - \frac{\lambda}{\lambda-1/\beta_1} + 0.27x + 0.96$	1.31	5.05
II	$\lambda(\beta_1 + a) + \frac{a-bx}{\beta_1} + x$	0.32	1.65
III	$\lambda(\beta_1 + a + b) + \frac{a-2b}{\beta_1} + 0.75x + 0.4$	0.30	1.10
IV	$2\lambda(\beta_1 + a + b) + \frac{0.75(a-2b)}{\beta_1} + 0.5x$	0.26	1.28
V	$1.5\lambda(\beta_1 + a + b) + a(2\lambda + 1/\beta_1) + 0.33x$	0.27	0.93
VI	$\sqrt{0.4(a\lambda + x)}(1.6\sqrt{a/\beta_1} + 0.6\lambda)$	0.03	0.34
VII	$a\lambda + \sqrt{x}(a\lambda + 1.5) + \sqrt{\sqrt{x}(a\lambda + 1.5)}$	0.01	0.18

with Roman numerals and the corresponding parameters. The first column, N, is used for a reference purpose, whereas the other columns are self-explanatory.

As it is also most often the case in practice, our goal is to find the decision policy which minimizes the average costs of the system. Therefore, although the algorithm is approximating the threshold function, in this section, we will not examine how good of a fit the estimation is. Instead, we present the relative difference, E_r, between the acquired costs if one uses the threshold policy from the generated expression, g^{est}, and the optimal one, g^{opt}. More precisely, we calculate

$$E_r = \frac{|g^{est} - g^{opt}|}{g^{opt}} \times 100\%.$$

The generated expressions and their accuracy are shown in Table 2. Next to the median, we also state the 95$^{\text{th}}$ percentile of E_r. Based on the results for formulas I to V, we conclude that incorporating the various guidelines in accordance with the MDP framework greatly improves the accuracy of the expressions without adding complex terms. Next to that, we note that including the square root operator can decrease the error even further. However, the remarkable accuracy of the threshold functions VI and VII come with the cost of expressions that are hard to interpret.

7 OBTAINING ALGEBRAIC RESULTS

Next to numerically evaluating the performance of the generated functions, we are also interested in how well these symbolic expressions resemble the optimal one. Therefore, in this section, we apply our technique to an MDP system that has a closed-form solution. In such a way, we can compare the derived expressions to the real one. Moreover, we use this example to once again go through the steps of our approach. In the following, we implement the algorithm according to the guidelines described in Sections 4 and 5. Furthermore, we apply the default settings in order to examine our technique in its general form, without any adjusting and tailoring to a specific model.

1. Modeling an MDP with a threshold-based policy: We study a single server queue with Poisson arrivals with rate λ and exponentially distributed service times with rate μ. There are holding costs, c_h, associated with each customer in the queue. Furthermore, one can decide to reject a customer upon arrival. In such a case, there is a rejection cost, c_r acquired. Even for such a simple system, obtaining the closed-form expression of the decision policy is very challenging. In [2], it was shown that the long-run average cost, g, for a system with $c_h = c_r = 1$ is given by:

$$g = \frac{\rho - \frac{(\tau+1)(1-\rho)\rho}{(1/\rho)^\tau - \rho}}{1-\rho} + \frac{(1-\rho)\lambda}{\left(\frac{1}{\rho}\right)^\tau - \rho},$$

where $\rho = \lambda/\mu$ is the system load and τ is the threshold value. Now, minimizing g with respect to τ gives the optimal threshold value τ_{opt}:

$$\begin{aligned}\tau_{opt} &= \mu - \lambda - 1 \\ &\quad - \frac{1}{\log(\rho)} \\ &\quad - \frac{L\left(-\rho \exp\left(\log\left(\rho\right)\left(\mu - 1 - \lambda\right) - 1\right)\right)}{\log(\rho)},\end{aligned} \tag{1}$$

where $L(.)$ is the Lambert-W function.

2. Preparing the data: Note that the state space of this MDP is one dimensional, i.e., the number of customers in the queue, and therefore each system configuration results in exactly one data sample (λ, μ, ρ, y) where y denotes the threshold value. We generated samples for systems with 100 equally spread values of μ in the range [1..1000] and 100 values of λ for each μ resulting in ρ from 0.05 to 0.95.

3. Specifying the algorithm settings: Three different setting configurations were tested. In the first one we used only the four basic mathematical operators (addition, subtraction, multiplication, and division), whereas in the second configuration we added two more that are present in the closed-form expression: inversion and natural logarithm. Finally, we generated a third program instance including also the Lambert-W function as a possible mathematical operator. In all of the configurations, the *init_d* and the *pc* were assigned to the suggested default values, namely $(1, 3)$ and 0.01, respectively. Furthermore, since each system instance is associated with exactly one sample, there is no need of using weights in the *fitness function*.

4. Evaluating the results: The outcomes of the above-described three setting configurations are given by the following approximations τ_1, τ_2, and τ_3 for the optimal threshold policies, respectively:

$$\tau_1 = \mu - \lambda - 0.657 + \frac{\lambda}{\mu - \lambda}; \tag{2}$$

$$\tau_2 = \mu - \lambda - \frac{\mu - \lambda}{\mu - \lambda} - \frac{1}{\log(\rho)} = \mu - \lambda - 1 - \frac{1}{\log(\rho)}; \tag{3}$$

$$\tau_3 = \mu - \lambda - 0.597 + \frac{\lambda}{\mu - \lambda}. \tag{4}$$

The results show that the technique is able to find the most influential terms (namely, $\mu - \lambda$) in all three configurations. Furthermore, in the first case (Eq. (2)) it approximated $(-1 - 1/\log(\rho))$ using the term $\lambda/(\lambda - \mu)$ and a constant. It is interesting to note that this is indeed a very good estimation as it is exactly the first (and most important) term, $\rho - 1$, from the Taylor expansion of $\log(\rho)$ around 0. Namely:

$$\frac{1}{\log(\rho)} \approx \frac{1}{\rho - 1} = \frac{\mu}{\lambda - \mu} = \frac{\lambda}{\mu - \lambda} - 1.$$

In addition, we believe that the high accuracy of this approximation led the algorithm to use it also in τ_3, although the log and the inverse operands were allowed by the third settings configuration. The fact that τ_2 contains the exact term $-1 - 1/\log(\rho)$ shows that given more evolutionary time (e.g., more generations, and/or different random seeds) would have helped the third configuration to discover this term.

Finally, we note that the last part of the exact threshold expression that involves the Lambert-W function was not included by the algorithm. After further analysis, we found that the mean and the variance of this term across the sampled systems were -0.005 and 0.003, respectively. Therefore, given that the mean threshold was 1252, we believe that this additional term is indeed negligible. Based on these findings, we conclude that one might use our technique not only for deriving a very well-performing decision policy, but also the reason for its structure, and therefore, the importance of certain parameters and the relations between them.

8 CONCLUSION

This paper is a pioneering contribution, and presents a new and promising technique to obtain an analytic solution to MDPs that have a threshold-based optimal policy. The method makes use of a specific machine learning algorithm - the symbolic regression. Therefore, we showed how one can apply and tailor this genetic program to the MDP framework. Although the obtained solution might not be the optimal one, the decision policy is nevertheless near-optimal and furthermore given in a closed-form expression.

The technique introduced in this paper was tested on two MDP models, resulting in highly accurate approximations both in terms of the achieved system performance and the form of the expression. We believe that the first next step would be to extend the algorithm to a broader range of MDP problems.

The results also raise a number of other questions for further research. For example: (1) How do the generated approximations relate to the size of the training data set?, (2) What is a good experimental setup for the specific parameter settings used in the training set?, (3) How does the choice of the hyper-parameters (the set of basis functions, the set of operators, the set of features) of the SR-algorithm influence the approximations?, (4) How does inaccuracy in the calculations of of the value functions affect the quality of the approximations?, and (5) To what extent is this general methodology applicable to other models, and beyond approximating control policies in MDPs?

REFERENCES

[1] D. Barash. A genetic search in policy space for solving Markov decision processes. In *AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information*. AAAI Press, 1999.
[2] S. Bhulai. Markov decision processes: The control of high-dimensional systems, 2002.
[3] S. Bhulai and G. Koole. On the structure of value functions for threshold policies in queueing models. *J. Appl. Probab.*, 40(3):613–622, 09 2003.
[4] H.S. Chang, H.-G. Lee, M.C. Fu, and S.I. Marcus. Evolutionary policy iteration for solving Markov decision processes. *IEEE Transactions on Automatic Control*, 50(11):1804–1808, Nov 2005.
[5] M.W. Khan and M. Alam. A survey of application: Genomics and genetic programming, a new frontier. *Genomics*, 100(2):65 – 71, 2012.
[6] G. Koole. A simple proof of the optimality of a threshold policy in a two-server queueing system. *Syst. Control Lett.*, 26(5):301–303, December 1995.
[7] J.R. Koza, D. Andre, F.H. Bennett, and M.A. Keane. *Genetic Programming III: Darwinian Invention & Problem Solving*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.
[8] Y. Levy. A class of scheduling policies for real-time processors with switching system applications. 1985.
[9] Z.-Z. Lin, J.C. Bean, and C.C. White. A hybrid genetic/optimization algorithm for finite-horizon, partially observed Markov decision processes. *INFORMS Journal on Computing*, 16(1):27–38, 2004.
[10] J.M. Norman. *Heuristic procedures in dynamic programming*. Manchester University Press Manchester, 1972.
[11] M. Onderwater, S. Bhulai, and R.D. van der Mei. Value function discovery in Markov decision processes with evolutionary algorithms. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46(9):1190–1201, Sept 2016.
[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[13] M.L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. New York, NY, USA: John Wiley & Sons, 1994.
[14] D. Roubos and S. Bhulai. Approximate dynamic programming techniques for the control of time-varying queuing systems applied to call centers with abandonments and retrials. *Probab. Eng. Inf. Sci.*, 24(1):27–45, January 2010.
[15] T. Stephens. Gplearn version 0.2.0. https://gplearn.readthedocs.io/en/stable/, 2016.
[16] R.S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, 3(1):9–44, August 1988.
[17] R.S. Sutton and A.G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.
[18] H. Tijms. *A First Course in Stochastic Models*. Wiley, 2003.
[19] A. Yener and C. Rose. Genetic algorithms applied to cellular call admission: local policies. *IEEE Transactions on Vehicular Technology*, 46(1):72–79, Feb 1997.