frontiers in Robotics and AI

BRIEF RESEARCH REPORT
published: 29 April 2021
doi: 10.3389/frobt.2021.625125

Off-Policy Evaluation of the Performance of a Robot Swarm: Importance Sampling to Assess Potential Modifications to the Finite-State Machine That Controls the Robots

Federico Pagnozzi and Mauro Birattari**

IRIDIA, Université libre de Bruxelles, Brussels, Belgium

Due to the decentralized, loosely coupled nature of a swarm and to the lack of a general design methodology, the development of control software for robot swarms is typically an iterative process. Control software is generally modified and refined repeatedly, either manually or automatically, until satisfactory results are obtained. In this paper, we propose a technique based on off-policy evaluation to estimate how the performance of an instance of control software—implemented as a probabilistic finite-state machine—would be impacted by modifying the structure and the value of the parameters. The proposed technique is particularly appealing when coupled with automatic design methods belonging to the AutoMoDe family, as it can exploit the data generated during the design process. The technique can be used either to reduce the complexity of the control software generated, improving therefore its readability, or to evaluate perturbations of the parameters, which could help in prioritizing the exploration of the neighborhood of the current solution within an iterative improvement algorithm. To evaluate the technique, we apply it to control software generated with an AutoMoDe method, Chocolate – 6S . In a first experiment, we use the proposed technique to estimate the impact of removing a state from a probabilistic finite-state machine. In a second experiment, we use it to predict the impact of changing the value of the parameters. The results show that the technique is promising and significantly better than a naive estimation. We discuss the limitations of the current implementation of the technique, and we sketch possible improvements, extensions, and generalizations.

Keywords: swarm robotics, control software architecture, automatic design, reinforcement learning, importance sampling

OPEN ACCESS

Edited by:
Savvas Loizou,
Cyprus University of Technology, Cyprus

Reviewed by:
Alan Gregory Millard,
University of Lincoln, United Kingdom
Heiko Hamann,
University of Lübeck, Germany

****Correspondence:***
Federico Pagnozzi
federico.pagnozzi@ulb.ac.be
Mauro Birattari
mbiro@ulb.ac.be

Specialty section:
This article was submitted to Multi-Robot Systems, a section of the journal Frontiers in Robotics and AI

Received: *02 November 2020*
Accepted: *17 February 2021*
Published: *29 April 2021*

Citation:
Pagnozzi F and Birattari M (2021) Off-Policy Evaluation of the Performance of a Robot Swarm: Importance Sampling to Assess Potential Modifications to the Finite-State Machine That Controls the Robots. Front. Robot. AI 8:625125. doi: 10.3389/frobt.2021.625125

1 INTRODUCTION

In this paper, we investigate the use of off-policy evaluation to estimate the performance of a swarm of robots. In swarm robotics (Dorigo et al., 2014), a group of robots act in coordination to perform a given mission. This engineering discipline is inspired by the principles of swarm intelligence (Dorigo and Birattari, 2007). The behavior of the swarm is determined by the local interactions of the robots

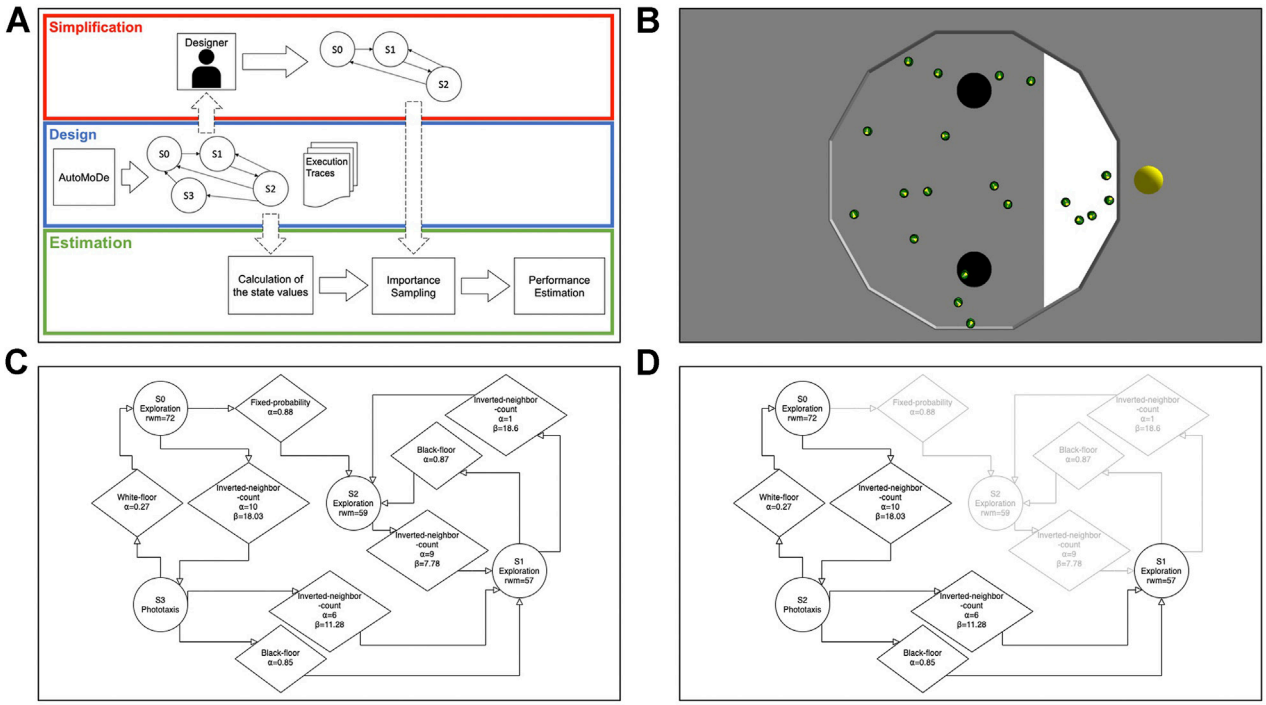

FIGURE 1 | In panel **(A)**, we show the workflow of the experiments. During the design process, AutoMoDe produces several finite-state machines—such as the one composed of four states shown in panel **(C)**—as well as the execution traces of each experiment executed during the design process. The finite-state machine is then modified by a designer—either a human or an automatic procedure—producing a modified finite-state machine—such as the one produced by removing a state shown in panel **(D)** where the state and the relative transitions that have been removed are shown in light gray—or by modifying the parameters. First, the state values of the original finite-state machine are calculated from the execution logs using the *first-visit* MC method. Importance sampling uses the state values, the modified finite-state machine, and the execution traces to calculate the estimated state values of the modified finite-state machine that are then used to produce an estimation of the performance of the modified finite-state machine. Finally, in panel **(B),** we show a screenshot from the ARGoS simulator where a swarm of 20 robots is performing the foraging task using the control software shown in panel **(C)**.

with each other and with the environment. In a robot swarm, there is no single point of failure and additional robots can be added to the swarm without changing the control software. Unfortunately, these same features make designing the control software of the individual robots comprised in a swarm a complex endeavor. In fact, with the exception of some specific cases (Brambilla et al., 2015; Reina et al., 2015; Lopes et al., 2016), a general design methodology has yet to be proposed (Francesca and Birattari, 2016). Typically, the design of the control software of the individual robots comprised in a swarm is an iterative improvement process based on trial and error and heavily relies on the experience and intuition of the designer (Francesca et al., 2014). Automatic design has shown to be a valid alternative to manual design (Francesca and Birattari, 2016; Birattari et al., 2019). Automatic design methods work by formulating the design problem as an optimization problem, which is then solved using generally available heuristic methods. The solution of the optimization problem is an instance of control software and the solution quality is a measure of its performance. In other words, the optimal solution of such optimization problem is the control software that maximizes an appropriate mission-dependent performance metric. Reviews of the swarm robotics literature can be found in Garattoni and Birattari (2016) and Brambilla et al. (2013), while in depth reviews of automatic design in swarm robotics can be found in Francesca and Birattari (2016); Bredeche et al. (2018); Birattari et al. (2020).

In this study, we focus on control software implemented as a probabilistic finite-state machine (PFSM): a graph where each node represents a low-level behavior of the robot and each edge represents a transition from a low-level behavior to another. When the condition associated to a transition is verified, the transition is performed and the current state changes. In a probabilistic finite-state machine, each transition whose associated condition is verified may take place with a certain probability. This control software architecture is human readable and modular—states and transitions can be defined once and easily reused or changed. Due to these characteristics, finite-state machines have been commonly used in manual design as well as in automatic design methods such as the ones belonging to the AutoMoDe family (Francesca et al., 2014).

In AutoMoDe, the control software is generated by combining pre-existing parametric software modules in a modular architecture, such as a probabilistic finite-state machine or a behavior tree. When considering finite-state machines, the software modules are either state modules and transition modules. The optimization algorithm designs control software by optimizing the structure of the PFSM—the number of states and how they are connected to each other—the behaviors, the

transitions, and their parameters. In **Figure 1C**, we show an example of a finite-state machine generated using AutoMoDe.

Regardless of the design method and control software architecture, once generated, the control software is improved through an iterative process where changes are evaluated and applied if considered promising. For instance, a human designer applies this process when fine-tuning the configuration of the control software. The optimization algorithms used in automatic design methods, and in particular iterative optimization methods, also work in this way. For instance, iterative optimization methods start from one or multiple solutions and explore the solution space in an iterative fashion. At each iteration, the optimization process generates new solutions by considering modifications of the previous solution(s). Having an estimate of the performance of such modifications can save valuable resources—access to appropriate computational resources can be expensive—and significantly speed up the design process. Furthermore, in the context of automatic design, such an estimate could be used to reduce the complexity of the generated control software. Indeed, the automatic design process often introduces artifacts in the generated control software, that is, there may be parts of the control software that do not contribute to the performance because they either do not influence the behavior of the robots or are never executed. These artifacts are generally ignored, but they add unnecessary complexity and hinder readability.

Our proposal is to use off-policy evaluation to estimate the impact of a modification from data collected during the execution of the control software. Off-policy evaluation, is a technique developed in the context of reinforcement learning (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 2018), to estimate the performance of a *target* policy from the observation of the one of a *behavior* policy. In a policy, the world is represented as a set of states—possible configurations of the environment, including the robot itself—connected by actions—the possible interactions with the world. Each state is associated with a set of possible actions that can be taken to transition from one state to the other. The target policy may be deterministic—given a state it always executes the same action—or stochastic—the action to be performed is chosen with a certain probability; while the behavior policy must be stochastic (Sutton and Barto, 2018).

Almost all off-policy methods use a technique called importance sampling (Hammersley and Handscomb, 1964; Rubinstein and Kroese, 1981; Sutton and Barto, 2018). This technique is used to compute a statistic of a distribution based on a sample extracted from another. In this way, data generated by one policy can be used—after being appropriately weighted through importance sampling—to evaluate a different policy. For instance, one may execute a policy that performs actions randomly and use the observed policy performance to estimate the performance of a deterministic policy that always chooses the action with the highest expected reward. Given a set of actions executed by the behavior policy, this technique estimates the performance of the target policy from the one observed by performing the behavior policy by weighting the latter with the ratio between the probability of executing each action under the target policy and the one of executing them under the behavior policy. As a consequence, the behavior policy must contain all the states and actions of the target policy. Moreover, under the behavior policy, the probabilities of executing each action under each state must be strictly positive.

Off-policy methods based on importance sampling have been studied in reinforcement learning for a long time (Hammersley and Handscomb, 1964; Powell and Swann, 1966; Rubinstein and Kroese, 1981; Sutton and Barto, 2018). Recent works focused on combining importance sampling with temporal difference learning and approximation methods (Precup et al., 2000, Precup et al., 2001), as well as reducing the variance of the estimation (Jiang and Li, 2016) and improving the bias-variance trade-off (Thomas and Brunskill, 2016).

The control software of a robot is indeed the implementation of a policy. In fact, a robot uses information acquired through its sensors to acquire information on the world and execute an action by properly operating its actuators and motors. Depending on the control software architecture, the state might be explicitly reconstructed or not, and the set of actions available in each state might be defined in a more or less explicit way. In the case of probabilistic finite-state machines, the similarities of this control software architecture with policies makes it ideal for this study. By considering additional information from the sensors of the robot, the states and transitions of a PFSM are directly translated in states and actions of a policy. In the technique we propose in this paper, the relevant data is the execution traces—that is, the sequence of internal states traversed by the controller and the sequence of sensor readings—from each robot in the swarm during multiple experimental runs.

To evaluate the technique we propose, we use control software generated with a variant of Chocolate (Francesca et al., 2015) that we modified to allow the generation of more complex finite-state machines composed of up to six states. In order to avoid confusion, we call this variant $Chocolate-6S$. A further advantage of using AutoMoDe is that collecting the execution data needed for the estimation does not require additional experimental runs because the technique we propose can operate on the data produced within the design process. In the experiments, we consider two kind of modifications, one concerning the structure and one the parameters of the control software. In the first, we estimate the performance after removing one of the states of the control software. In the second, we evaluate the impact of modifying the values of two parameters of the two most executed transitions. In both experiments, we compared the estimation provided by the proposed technique with a naive estimation made assuming that the applied modification would not change the performance. The results show that the proposed technique is better than the naive estimation and that the difference between the two is statistically significant.

The structure of the paper is the following. In **Section 2**, we present off-policy evaluation and how it can be applied to finite-state machines. The experiments, their setup, and results are presented in **Section 3**. Finally in **Section 4**, we discuss the limitations of the proposed technique, possible ways to improve the estimation and how it can be extended to other control software architectures.

2 METHOD

2.1 Background: Off-Policy Evaluation

The main challenge in reinforcement learning is estimating how desirable it is for an agent to be in a state. The value $v_\pi(s)$ of a state s under a policy π is the expected reward obtainable by starting in state s and then following the policy π. The reward is calculated over an episode e that can be defined as an entire interaction between an agent and the environment, from an initial to a final step. The reward obtained after step t can be calculated as

$$G_t(s;e) \doteq R_{t+1} + \gamma G_{t+1}. \tag{1}$$

In **Eq. 1**, R_{t+1} is the reward obtained at step $t+1$ and $\gamma \in [0,1]$ is a parameter called discount rate, which allows us to model the fact that rewards may depend less and less on states visited earlier in the episode.

Using Monte Carlo (MC) methods, the value function can be estimated from a sample of episodes. In these methods, the value $v_\pi(s)$ of a state s is calculated as the average of the returns following the visit to state s. In the *first-visit* MC method only the reward following the first visit is considered while in *every-visit* all the visits contribute to the average. As the two methods are similar and both converge to $v_\pi(s)$ when the number of visits tends to infinity, to simplify calculations in this work we use the *first-visit* MC method.

Off-policy evaluation estimates the value function of a policy π from episodes generated by another policy b. To be able to use episodes from b to estimate values for π, b must *cover* π, that is, it must be possible—under b—to take every action that can be taken under π. Formally, if $\pi(a|s) > 0$ then $b(a|s) > 0$, where $\pi(a|s)$ indicates the probability under policy π of taking action a when in state s. Assuming that b covers π, importance sampling can be used to weight the returns of b considering that—between the policies—each action may be taken with a different probability. Defining $\tau(s)$ as the sequence of states and actions $S_t, A_t, S_{t+1}, A_{t+1}, \ldots, S_T$—where S_t is the first visit to state s, A_t is the action taken when in S_t and S_T is the final state—the ratio between the different probabilities—called importance sampling ratio—can be expressed as

$$\rho_{\tau(s)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \tag{2}$$

In **Eq. 2**, $\pi(A_k|S_k)$ and $b(A_k|S_k)$ indicate the probability of taking action A_k when in state S_k under the target policy and the behavior policy, respectively. Given a set of episodes E generated with policy b, there are two main ways of using $\rho_{\tau(s)}$ to estimate $v_\pi(s)$: *ordinary importance sampling* and *weighted importance sampling* (WIS). *Ordinary importance sampling* is defined as

$$v_\pi(s) = \frac{\sum_{e \in E} \rho_{\tau(s)}(e) G_t(s;e)}{|E|}. \tag{3}$$

Weighted importance sampling instead is defined as

$$v_\pi(s) = \frac{\sum_{e \in E} \rho_{\tau(s)}(e) G_t(s;e)}{\sum_{e \in E} \rho_{\tau(s)}(e)}. \tag{4}$$

The main difference between **Eqs. 3**, **4** is that, in the latter equation, $\rho_{\tau(s)}$ is in the denominator. Both *ordinary importance sampling* and *weighted importance sampling* tend to the exact state value when increasing the number of episodes considered but the two methods show a different variance and bias trade-off. *Ordinary importance sampling* is unbiased but shows a very high variance, *weighted importance sampling* introduces some bias resulting in a far lower variance (Sutton and Barto, 2018). We selected WIS for our implementation because during some preliminary experiments the variance of the estimation was found to impact significantly the results.

2.2 Our Technique: Applying Off-Policy Evaluation to Finite-State Machines

Applying off-policy evaluation to finite-state machines requires establishing what are states and actions in a PFSM, what is an episode and how to calculate the final reward, G_T, from the score attained by the whole swarm at the end of an experimental run. In finite-state machines, each state represents a behavior that is executed until an event triggers a transition to another state. Considering that the robot does not change its behavior until a transition is triggered, our working hypothesis is that a state of a PFSM–together with the information from the robot sensors saved in the execution traces—can be simplified in a single state of a policy and, consequently, transitions acquire the same function as actions in a policy. With these assumptions—considering a PFSM π—$\pi(a|s)$ can be defined as the probability that transition a is triggered when in state s.

Naturally, an episode should correspond to an experimental run with the reward being the final score, but in swarm robotics there are several robots—typically all running the same control software. For this reason, we divide an experimental run involving a swarm of n robots in a set of n parallel episodes. Similarly, assuming F is the final score of the experimental run, the reward awarded to each robot corresponds to F/n. In calculating G_T using **Eq. 1**, because assigning a per-time-step reward is not always possible, we set $R_{t+1} = 0$—that is, the reward is given only at the end of the episode—and we do not consider any discount—that is, $\gamma = 1$—resulting in $G_T = F/n$. Using the *first-visit* method, a state will get a reward of G_T if it is executed at least once during the episode. In this case, a state that is executed once—for instance, the initial state—will have the same reward as a state that is executed for almost the entirety of the episode. We also considered another way of calculating G_T that consists in weighting the reward by the relative execution time of each state. This proportional G_T is calculated per state so that a state s that has been executed for k time steps gets a reward $GP_T(s) = (F/n) \cdot (k/\text{steps})$ where *steps* is the total number of time steps in the episode. The pseudo code showing how to estimate the state values is shown in **Algorithm 1**.

Algorithm 1 Pseudo code showing how the state values are calculated. The inputs are the finite-state machine and its execution traces generated during the design process. Each execution trace contains the recording of all the robots in the swarm as well as the final score awarded to the swarm at the end of the experiment. The procedure iterates over each execution trace and for each execution trace considers each robot separately. For each robot, the value of each state is calculated using the *first-visit* MC method and considering two ways of calculating the reward. In the first, the reward is equal for each state and is equal to the reward per robot.

TABLE 1 | Description of Chocolate – 6S. This method targets the e-puck robot and specifically the reference model RM1.1 of which we report the key features at the end of the table. The finite-state machines are generated by choosing from six behaviors—for which we provide a brief description—and six conditions—for which we report how the activation probability p is calculated with α and β being parameters. Chocolate – 6S can generate finite-state machines of maximum six states, while Chocolate allows a maximum of four and each state can have a maximum of four transitions. The optimization algorithm is iterated F-Race which uses the ARGoS simulator to perform evaluations.

Chocolate – 6S		
Modules		
Behaviors	Exploration	Move randomly
	Stop	Stop moving
	Phototaxis	Move toward the light
	Anti-phototaxis	Move away from the light
	Attraction	Move toward other robots
	Repulsion	Move away from other robots
Conditions	Black-floor	If the floor is black, $P = \alpha$; 0 otherwise
	White-floor	If the floor is white, $P = \alpha$; 0 otherwise
	Gray-floor	If the floor is gray, $P = \alpha$; 0 otherwise
	Neighbor-count	With n neighbors $P = 1/(1 + e^{\beta \cdot (\alpha - n)})$
	Inverted-neighbor-count	With n neighbors $P = 1 - 1/(1 + e^{\beta \cdot (\alpha - n)})$
	Fixed-probability	$P = \alpha$
Constraints		
Number of states	6 (max)	
Transitions per state	4 (max)	
Tools		
Optimization algorithm Simulator	iterated F-Race implemented in the irace package (Balaprakash et al. (2007); López-Ibáñez et al. (2016)) ARGoS3 (Pinciroli et al. (2012))	
Robot platform	e-puck reference model RM1.1 (Hasselmann et al. (2020))	
Input	8 proximity sensors	
	8 light sensors	
	3 ground sensors	
	Number of neighboring robots perceived	
	Attraction vector for each perceived robot	
Output	Left and right wheel target linear velocity	

In the second, each state has a reward calculated as a fraction of the reward per robot proportional to the total time the state was active.

Input A PFSM composed of S states and T transitions.
Input A set of execution traces
Output $V(s)$ state values using the full reward
Output $V_p(s)$ state values using the proportional reward
for each $s \in S$ **do**
  Initialize Rewards(s) as an empty list
  Initialize Rewards$_p(s)$ as an empty list
loop over each execution trace
  F = final score of the swarm
  n =number of robots in the swarm
  $G_T = F/n$ reward per robot
  steps = number of time steps in the execution trace
  for each robot in the swarm **do**
    for each state $s \in S$ **do**
      if s has been executed in the episode **then**
        k = number of time steps s has been executed
        Append G_T to Rewards(s)
        Append $G_T \cdot k$/steps to Rewards$_p(s)$

$V(s) = \text{average}(\text{Rewards}(s))$
$V_p(s) = \text{average}(\text{Rewards}_p(s))$

Given these definitions, we can use *weighted importance sampling* to estimate the state values of a *target* finite-state machine from the execution traces of a *behavior* finite-state machine, provided that the states of the behavior finite-state machine are a superset of those of the target one and are connected by transitions in the same way. To calculate a performance estimation for the whole swarm when executing the target control software, F/n has to be derived from the state values estimated using *weighted importance sampling*. Let $v_b(s)$, with $s \in S$ be the state values of the behavior finite-state machine and $v_t(j)$, with $j \in J$ and $J \subseteq S$, be the state values of the target finite-state machine estimated using *weighted importance sampling*. When considering the reward calculated as $G_T = F/n$, the performance F_e of the target finite-state machine is

$$F_e = \frac{\sum_J \left(v_t(j)/v_b(j) \right)}{|J|} \cdot G_T. \tag{5}$$

In **Eq. 5**, the estimation is calculated as the per-robot reward multiplied by a factor that combines the estimated state values of the target finite-state machine weighted by the state values of the behavior finite-state machine. The proportional reward already considers the relative contribution of each state so the calculation of F_e is defined as follows:

$$F_e = \sum_J \nu_t(j) GP_T. \tag{6}$$

F_e represents the estimation of the average performance of a single robot in the swarm. The estimated performance of the swarm, considering the assumption that all the robots contribute equally, is calculate as $F_e \cdot n$.

The experiments presented in this paper are based on finite-state machines generated with Chocolate – 6*S*, which builds PFSM composed of a maximum of six states and four transitions per state. Each state can assume one of six behaviors and each transition can have one of six conditions. The key characteristics of Chocolate – 6*S*, as well as a brief description of the behaviors and the conditions, are given in **Table 1**. To produce execution traces for each experimental run, we modified AutoMoDe so that the control software of each robot logs an execution trace containing, for each time step, the current state, the active transition(s) and the information needed to calculate the activation probability of each condition—that is, the ground color and the number of neighboring robots perceived.

3 RESULTS

In the experiments presented here, the execution traces are collected during the generation of the control software from the executions performed by the optimization algorithm used in Chocolate – 6*S*, Iterated F-Race implemented in the irace package (Birattari, 2009; López-Ibáez et al., 2016). In a nutshell, I/F-race works in an iterated fashion by generating, testing and discarding solutions—i.e., finite-state machines. In each iteration, I/F-race keeps a set of solutions that are executed and compared with each other. A solution is discarded when proved worse than the others by means of a statistical test. The algorithm ends when the maximum amount of executions is reached, returning the set of surviving solutions. **Figure 1A** shows a description of how a finite-state machine is generated and modified as well as how the performance of the modified control software is estimated.

We applied off-policy evaluation to 20 finite-state machines generated with Chocolate – 6*S* to perform a foraging mission as defined by Francesca et al. (2014). In this mission, a swarm of 20 robots, confined in an dodecagonal arena, must retrieve as many objects as possible from two sources and transport them to the nest. A screenshot of an experimental run is shown in **Figure 1B**. The e-puck robot is not able to manipulate objects so the interaction with objects is abstracted. We consider that a robot collects an object by entering a source and deposits it by entering the nest. The sources are two black circles roughly in the middle of the arena while the nest is a white area placed at the edge of the arena. Additionally, a light source is placed behind the nest. The performance metric is defined as the number of objects retrieved. A video of an experimental run showcasing the mission as well as the source code and the experimental data are available in the supplementary page (Supplementary Material).

All the experiments were conducted in a simulated environment using ARGoS3 Pinciroli et al. (2012). We executed Chocolate – 6*S* ten times with a budget of 10,000 evaluations, generating 93 finite-state machines. From this group, we removed the finite-state machines that had less than three active states according to the execution traces. From the remaining ones, we formed the final group of 20 finite-state machines by selecting the ones showing the greatest number of active states. During the execution of Chocolate – 6*S*, these finite-state machines were executed between seven and ten times. Considering that each experimental run involves twenty robots, we collected from 140 to 200 episodes for each finite-state machine.

We devised two experiments: one, called "state pruning," in which we consider changes to the structure of the finite-state machines and one, called "parameter variation," where we consider variations in its configuration. In state pruning, for each finite-state machine, we use the proposed technique to estimate the performance of all the finite-state machines that can be generated by removing one state—for instance, three finite-state machines composed of two states can be generated from one composed of three states. **Figure 1D** shows a finite-state machine composed of three states generated by removing state S2 from the finite-state machine in **Figure 1C**. In parameter variation, for each finite-state machine, we generate four variations by changing two parameters of the two most active transitions. For each parameter, we considered two different values generating four combinations per finite-state machine. For instance, we can generate four finite-state machines by considering two new values for the α parameters of the two transitions connecting the states S0 and S3 in the finite-state machine in **Figure 1C**. From these 80 combinations, we discarded the ones unable to change significantly the performance of the control software after five experimental runs, resulting in a total of 20 parameter variations. In both experiments, we compared the performance estimations calculated with weighted importance sampling—both with and without the proportional reward—with a naive estimation implemented as the average performance of the unmodified finite-state machine as reported in the execution traces. In other words, the naive estimation always assumes that the changes done to the control software will not influence its performance. We measured the accuracy of each estimation using the normalized squared error (SE):

$$\text{normalized SE} = \frac{\left(\pi_i(E) - P_{i,b}(E)\right)^2}{\pi_i(E)}. \tag{7}$$

In the equation, $\pi_i(E)$ is the measured average performance of the finite-state machine i over the set of executions E and $P_{i,b}(E)$ is the estimated performance calculated from the traces generated by the finite-state machine b within the set of executions E. Moreover, we tested the results for significance using the Friedman rank sum test.

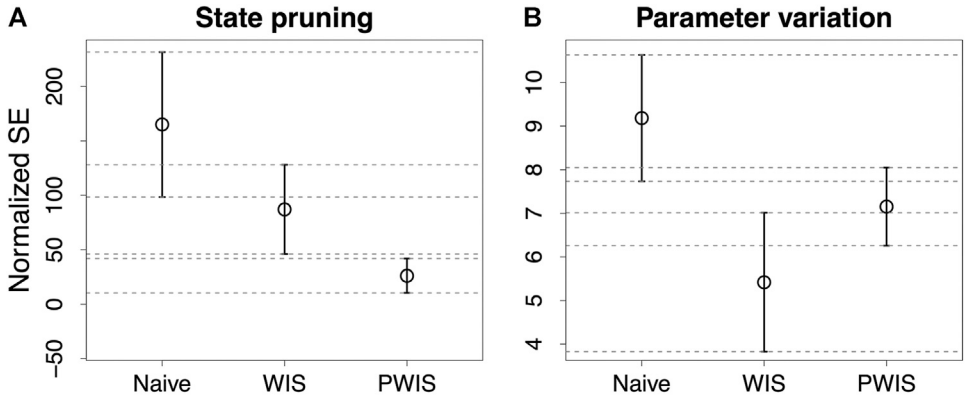

FIGURE 2 | Results of the experiments measured as normalized mean squared error. In **(A)**, we report the results of estimating the impact of removing a state; in **(B)**, we report those of estimating the impact of changing the probabilities of the two most active transitions. Naive indicates the naive estimation, while WIS and PWIS indicate the results obtained with the weighted importance sampling using, respectively, the full and the proportional per state reward.

The results for the two experiments are shown in **Figure 2** where, we indicate with WIS the result obtained using weighted importance sampling and with PWIS the result obtained using weighted importance sampling and the proportional reward. In both cases, PWIS and WIS have better results than the naive estimation with PWIS having significantly better results in the state pruning experiment—shown in **Figure 2A**—and WIS being significantly better in the parameter variation experiment—shown in **Figure 2B**. When comparing the results, the larger estimation error shown by all methods in the state pruning experiment can be explained by the fact that, in this experiment, the finite-state machines undergo substantial modifications which may invalidate the execution traces leading to performance estimation of 0. This is the case, for instance, when removing a state generates a finite-state machine with no transitions.

Overall, the results indicate that PWIS gives a better estimation when changing the structure of a finite-state machine while WIS is better suited to estimate the effect of variations to the parameters. In the first experiment, the proportional reward used in PWIS—that includes a measure of the relative execution time of each state—makes it better suited to estimate how the performance would change when removing a state. On the contrary, in the parameter variation experiment, the changes to the parameters influence directly the execution time of the states, making PWIS less accurate than WIS.

4 DISCUSSION

In this paper, we applied off-policy evaluation to estimate the performance of a robot swarm where the control software is represented as a finite-state machine. Although the experiments deliver promising results, further experimentation is needed, considering different missions as well as different sets of software modules. However, the results indicate that this line of research is promising with several developments that could be explored such as different reward calculations, different estimators. The execution traces can be modified to also trace the performance metric of the swarm so that more complex reward calculations can be implemented. The estimation can be improved by employing importance sampling methods such as the ones proposed by Jiang and Li (2016); Thomas and Brunskill (2016).

Moreover, this technique is not necessarily limited to finite-state machines and it could be extended with some modifications to other modular control software architecture such as, for instance, behavior trees (Kuckling et al., 2018). Another interesting application of this technique would be in automatic design methods using iterative optimization algorithms. The execution time of these methods might be reduced by running simulations only if newly generated solutions have an estimated performance that is better than the current best one.

DATA AVAILABILITY STATEMENT

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number (s) can be found as follows: http://iridia.ulb.ac.be/supp/IridiaSupp2020-012/index.html.

AUTHOR CONTRIBUTIONS

MB and FP discussed and developed the concept together. FP implemented the idea and conducted the experiments. Both authors contributed to the writing of the paper. The research was directed by MB.

FUNDING

The project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020

research and innovation programme (DEMIURGE Project, grant agreement no. 681872) and from Belgium's Wallonia-Brussels Federation through the ARC Advanced Project GbO-Guaranteed by Optimization. MB acknowledges support from the Belgian Fonds de la Recherche Scientifique–FNRS.

REFERENCES

Balaprakash, P., Birattari, M., and Stützle, T. (2007). "Improvement strategies for the F-Race algorithm: sampling design and iterative refinement," in *Hybrid metaheuristics, 4th international workshop, HM 2007*. Editors T. Bartz-Beielstein, M. J. Blesa, C. Blum, B. Naujoks, A. Roli, G. Rudolph, et al. (Berlin, Germany: Springer), 4771. 108–122. doi:10.1007/978-3-540-75514-2-9

Bertsekas, D. P., and Tsitsiklis, J. N. (1996). *Neuro-dynamic programming*. Nashua NH: Athena Scientific.

Birattari, M., Ligot, A., Bozhinoski, D., Brambilla, M., Francesca, G., Garattoni, L., et al. (2019). Automatic off-line design of robot swarms: a manifesto. *Front. Robot. AI.* 6, 59. doi:10.3389/frobt.2019.00059

Birattari, M., Ligot, A., and Hasselmann, K. (2020). Disentangling automatic and semi-automatic approaches to the optimization-based design of control software for robot swarms. *Nat. Mach Intell.* 2, 494–499. doi:10.1038/s42256-020-0215-0

Birattari, M. (2009). *Tuning metaheuristics: a machine learning perspective*. Berlin, Germany: Springer. doi:10.1007/978-3-642-00483-4

Brambilla, M., Brutschy, A., Dorigo, M., and Birattari, M. (2015). Property-driven design for robot swarms. *ACM Trans. Auton. Adapt. Syst.* 9 (4), 1–28. doi:10.1145/2700318

Brambilla, M., Ferrante, E., Birattari, M., and Dorigo, M. (2013). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intell.* 7, 1–41. doi:10.1007/s11721-012-0075-2

Bredeche, N., Haasdijk, E., and Prieto, A. (2018). Embodied evolution in collective robotics: a review. *Front. Robot. AI.* 5, 12. doi:10.3389/frobt.2018.00012

Dorigo, M., Birattari, M., and Brambilla, M. (2014). Swarm robotics. *Scholarpedia* 9, 1463. doi:10.4249/scholarpedia.1463

Dorigo, M., and Birattari, M. (2007). Swarm intelligence. *Scholarpedia* 2, 1462. doi:10.4249/scholarpedia.1462

Francesca, G., and Birattari, M. (2016). Automatic design of robot swarms: achievements and challenges. *Front. Robot. AI.* 3, 1–9. doi:10.3389/frobt.2016.00029

Francesca, G., Brambilla, M., Brutschy, A., Garattoni, L., Miletitch, R., Podevijn, G., et al. (2015). AutoMoDe-Chocolate: automatic design of control software for robot swarms. *Swarm Intell.* 9, 125–152. doi:10.1007/s11721-015-0107-9

Francesca, G., Brambilla, M., Brutschy, A., Trianni, V., and Birattari, M. (2014). AutoMoDe: a novel approach to the automatic design of control software for robot swarms. *Swarm Intell.* 8, 89–112. doi:10.1007/s11721-014-0092-4

Garattoni, L., and Birattari, M. (2016). "Swarm robotics," in *Wiley encyclopedia of electrical and electronics engineering*. Editor J. G. Webster (Hoboken, NJ, United States: John Wiley & Sons), 1–19. doi:10.1002/047134608X.W8312

Hammersley, J. M., and Handscomb, D. C. (1964). *Monte Carlo methods*. North Yorkshire, United Kingdom: Methuen.

Hasselmann, K., Ligot, A., Francesca, G., Garzón Ramos, D., Salman, M., Kuckling, J., et al. (2020). *Reference models for AutoMoDe. Tech. Rep. TR/IRIDIA/2018-002, IRIDIA*. Belgium: Université Libre de Bruxelles.

Jiang, N., and Li, L. (2016). "Doubly robust off-policy value evaluation for reinforcement learning," in International conference on machine learning. New York, NY, June 19–24, 2016 (Burlington, Massachusetts: Morgan Kaufmann), 652–661.

Kuckling, J., Ligot, A., Bozhinoski, D., and Birattari, M. (2018). "Behavior trees as a control architecture in the automatic modular design of robot swarms," in *Swarm intelligence – ants*. Editors M. Dorigo, M. Birattari, C. Blum, A. L. Christensen, A. Reina, and V. Trianni (Cham, Switzerland: Springer), Vol. 11172. 30–43. doi:10.1007/978-3-030-00533-7-3

Lopes, Y. K., Trenkwalder, S. M., Leal, A. B., Dodd, T. J., and Groß, R. (2016). Supervisory control theory applied to swarm robotics. *Swarm Intell.* 10, 65–97. doi:10.1007/s11721-016-0119-0

López-Ibáñez, M., Dubois-Lacoste, J., Pérez Cáceres, L., Birattari, M., and Stützle, T. (2016). The irace package: iterated racing for automatic algorithm configuration. *Operations Res. Perspect.* 3, 43–58. doi:10.1016/j.orp.2016.09.002

Pagnozzi, F., and Birattari, M. (2020). *Supplementary material for the paper: Off-policy evaluation of the performance of a robot swarm: importance sampling to assess potential modifications to the finite-state machine that controls the robots.* IRIDIA - Supplementary Information. ISSN: 2684–2041. Brussels, Belgium

Pinciroli, C., Trianni, V., O'Grady, R., Pini, G., Brutschy, A., Brambilla, M., et al. (2012). ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intell.* 6, 271–295. doi:10.1007/s11721-012-0072-5

Powell, M. J., and Swann, J. (1966). Weighted uniform sampling - a Monte Carlo technique for reducing variance. *IMA J. Appl. Math.* 2, 228–236. doi:10.1093/imamat/2.3.228

Precup, D., Sutton, R. S., and Dasgupta, S. (2001). "Off-policy temporal-difference learning with function approximation," in International conference on machine learning. New York, NY, June 2, 2001 (Burlington, Massachusetts: Morgan Kaufmann), 417–424.

Precup, D., Sutton, R. S., and Singh, S. (2000). "Eligibility traces for off-policy policy evaluation," in International conference on machine learning. New York, NY, June 13, 2000 (Burlington, MA: Morgan Kaufmann). 759–766.

Reina, A., Valentini, G., Fernández-Oto, C., Dorigo, M., and Trianni, V. (2015). A design pattern for decentralised decision making. *PLOS ONE.* 10, e0140950. doi:10.1371/journal.pone.0140950

Rubinstein, R. Y., and Kroese, D. P. (1981). *Simulation and the Monte Carlo method*. Hoboken, New Jersey: John Wiley & Sons.

Sutton, R. S., and Barto, A. G. (2018). *Reinforcement learning: an introduction*. Cambridge, MA: MIT Press.

Thomas, P., and Brunskill, E. (2016). "Data-efficient off-policy policy evaluation for reinforcement learning," in International conference on machine learning. New York, NY, United States, June 19–24, 2016 (Burlington, MA: Morgan Kaufmann), 2139–2148.

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.