Value Function Discovery in Markov Decision Processes with Evolutionary Algorithms

Martijn Onderwater, Sandjai Bhulai, and Rob van der Mei

***Abstract*—In this paper we introduce a novel method for discovery of value functions for Markov Decision Processes (MDPs). This method, which we call Value Function Discovery (VFD), is based on ideas from the Evolutionary Algorithm field. VFD's key feature is that it discovers descriptions of value functions that are algebraic in nature. This feature is unique, because the descriptions include the model parameters of the MDP. The algebraic expression of the value function discovered by VFD can be used in several scenarios, e.g., conversion to a policy (with one-step policy improvement) or control of systems with time-varying parameters. The work in this paper is a first step towards exploring potential usage scenarios of discovered value functions. We give a detailed description of VFD and illustrate its application on an example MDP. For this MDP we let VFD discover an algebraic description of a value function that closely resembles the optimal value function. The discovered value function is then used to obtain a policy, which we compare numerically to the optimal policy of the MDP. The resulting policy shows near-optimal performance on a wide range of model parameters. Finally, we identify and discuss future application scenarios of discovered value functions.**

***Index Terms*—Markov Decision Processes, Evolutionary Algorithms, Value Function, Genetic Programming.**

I. Introduction

MARKOV Decision Processes (MDPs) form a popular modelling framework for scenarios involving sequential decision making under uncertainty. It has been applied to a wide range of stochastic control problems, such as inventory management, telephone call admission in a call center, routing in telecommunication networks, and financial portfolio management.

Once a scenario is modelled with an MDP, various techniques are available to, e.g., obtain optimal policies for decision making. These techniques fall into two categories, namely numeric and algebraic techniques. In the former category, the most well-known methods are value iteration, policy evaluation, and policy iteration [1]. Value iteration is an iterative technique for finding an optimal control policy and the corresponding costs. With policy evaluation one can find the costs of a given policy, and policy iteration improves and evaluates policies iteratively.

This work is part of the project Realisation of Reliable and Secure Residential Sensor Platforms of the Dutch program *IOP Generieke Communicatie*, number IGC1020, supported by the *Subsidieregeling Sterktes in Innovatie*.

Martijn Onderwater and Rob van der Mei are at the Center for Mathematics and Computer Science (CWI), Science Park 123, 1098 XG Amsterdam, The Netherlands. E-mail: {m.onderwater,r.d.van.der.mei}@cwi.nl

Sandjai Bhulai, and the other two authors, are at VU University Amsterdam, Faculty of Sciences , De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. E-mail: s.bhulai@vu.nl

The aforementioned techniques are numeric in nature, so when, e.g., the model parameters change they have to be reapplied to the updated scenario. Ideally, one would like to solve an MDP algebraically and obtain the optimal policy (with the model parameters included). This approach is, however, often not feasible due to complexities of the model. In those cases, algebraic techniques might be used to show structural properties of the value function, which in turn give information about the structure of the optimal policy. For instance, a convex value function might imply that the optimal policy is a switching curve (see [2] for an example).

If the optimal policy cannot be obtained algebraically, a near-optimal policy is often sufficient for practical purposes. In this context, a technique called *one-step policy improvement*, introduced in [3], can be applied. It starts with a policy for which the MDP can be solved algebraically, yielding the corresponding value function. Then, by applying one step of policy iteration algebraically, this results in an improved policy. Since policy iteration typically makes the largest improvements in the first steps (see [4]), this improved policy is often near-optimal.

In this paper we describe a novel method (dubbed VFD, for *value function discovery*) that is aimed at obtaining an algebraic description of a value function. VFD is based on a numeric technique from the Evolutionary Algorithm (EA) family known as Genetic Programming (GP). One of the standard applications of GP is discovering algebraic descriptions of functions based on samples of this function at various points. To be precise, suppose a function $V(x)$ is unknown, but that we do have samples $V(s_i)$ at various points s_i. Applying GP allows discovery of an approximate algebraic expression for $V(x)$ (denoted by $\widetilde{V}(x)$) such that $V(s_i) \approx \widetilde{V}(s_i)$ for all sample points $(s_i, V(s_i))$. VFD applies GP to sample points of the optimal value function of an MDP, thereby allowing discovery of an algebraic description of the optimal value function.

The aim of the current paper is to introduce the concept of value function discovery using GP, and to illustrate its potential with a simple use case. For this use case, we convert the value function discovered by VFD to a policy and show that it has near-optimal performance. Then, we discuss other scenarios where we expect discovered value functions to be of use.

In the remainder of this paper we describe VFD and illustrate it by applying VFD to an example MDP. We start with an review of related work in Section II, and an introduction to GP in Section III. Then, we continue with a detailed description of VFD in Section IV and of the example MDP in Section V. Numerical results are presented in Section VI, followed by

a discussion in Section VII and concluding remarks in Section VIII.

II. Related work

VFD is based on GP, details about which can be found in the books [5], [6], and Section III contains a short description as well. Other members of the EA family are described in various textbooks, such as [5] and [7]. An introduction to MDPs is given in, e.g., books [4] and [8].

The literature combining EAs and MDPs mostly uses EAs to learn policies, whereas VFD learns value functions. In [9] the authors introduce evolutionary policy iteration, where the policy improvement step is integrated with an EA to iteratively obtain better policies. This procedure is shown to have monotone convergence for finite action spaces. The authors of [10] enhance the work in [9] by generating policies in the population via sub-MDPs, thereby speeding up convergence. From an application perspective, [11] provides an example of how EAs and MDPs can be used in a practical scenario. [12] compares an EA to policy iteration, and provides a useful reminder that policy iteration typically converges quickly and thus often outperforms an EA-approach.

Closest to our research is [13] by Lin et al., where the authors construct a piecewise linear approximation of the value function. In this approach, the linear elements are learned using a Genetic Algorithm. Like VFD, Lin's approach results in an approximation of the value function. However, the value function discovered by VFD is a closed-form expression, whereas [13] finds a piecewise linear approximation. Having a closed-form expression is preferable when, e.g., studying the structure of the MDP using the discovered value function. Also, [13] focuses on convex value functions, and VFD does not make any assumptions about the structure of the value function. Another difference is the type of EA that is used: [13] employs a Genetic Algorithm, whilst VFD is based on a GP. In particular, [13] does not use the tree-based representation inherent to GP. Finally, [13] does not allow for the placement of model parameters in the approximate value function.

A paper that does use GP in an MDP-context is [14]. The authors loosely explore the combination of GP and MDPs on an example of a war game and show that it performs well compared to a pure MDP-based technique. Their approach differs from the one described in this paper, because they use GP to learn policies and not value functions, as VFD does.

Summarizing, the distinguishing feature of VFD is its focus on discovering value functions. Although existing methods in literature choose to learn policies, learning value functions has significant advantages as well. **In particular, VFD has the following benefits:**

- VFD applied to optimal value functions yields policies with near-optimal performance.
- For MDPs that allow for an explicit closed-form expression of the optimal value function, VFD can find this optimal value function with arbitrary precision. Thus, it can also find the optimal policy for such MDPs.
- VFD produces an algebraic expression of a policy that includes the parameters of the MDP. Consequently, this policy is still applicable if the parameters of the model change in value. This allows for dynamic control in time-varying systems, without making the underlying model time-dependent.
- Value functions discovered by VFD can help gain an understanding of the structure of the optimal value function, policy, and model.
- Alternative techniques for analyzing MDPs often require knowledge of structural properties of the value function (e.g., gradient-based method such as local search). These properties can be discovered by VFD.
- For many MDPs a near-optimal policy does not require a very accurate fit of the optimal value function. Thus, learning value functions can quickly result in good policies.
- VFD works with any MDP without requiring any changes to the algorithm.

III. Genetic Programming

Since VFD is based on GP, we give a short description of this technique in this section. GP maintains a population of individuals and iteratively attempts to improve this population over several generations. In each generation, the current population generates new offspring by combining individuals. The underlying idea of GP is that combining good individuals leads, over time, to offspring that are better than their predecessors.

As mentioned in the introduction, one of the standard applications of GP is finding an algebraic description of a function based on numeric approximations. Hence, the individuals in the population used by GP have a specific representation that allow them to be interpreted as functions. This representation is discussed in more detail in Section III-A. The two operators involved in generating offspring are described in Sections III-B and III-C. Determining the quality of an individual is related to VFD's application of GP to MDPs, so we postpone it until Section IV.

A. GP *representation*

GP uses trees to represent a function, and several of these trees together form the population. Fig. 1a illustrates a tree representation of the function $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$. The operators $(/, *, +, -)$ from this expression are in the internal nodes of the tree, whereas the leafs contain the variables (x), parameters (ρ, μ), and constants $(1, 2)$. In this paper we only use the operators $(/, *, +, -)$ shown in the example, but the representation is flexible and also allows for, e.g., exponents, square roots, and logarithms. Finally, note that a representation of a tree is not unique: the tree in Fig. 1b is also a valid representation of $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$. Unicity of representation is, however, not required by VFD. In fact, this feature is used by VFD to include a preference for short trees.

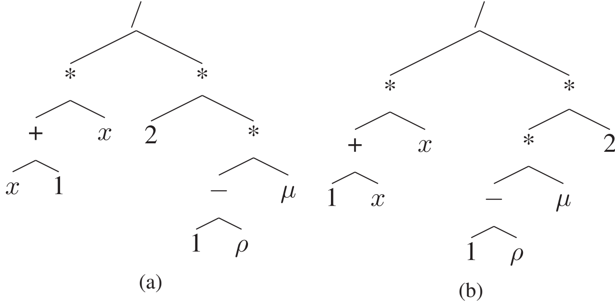

(a) (b)

Fig. 1. Two trees, each a representation of $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$

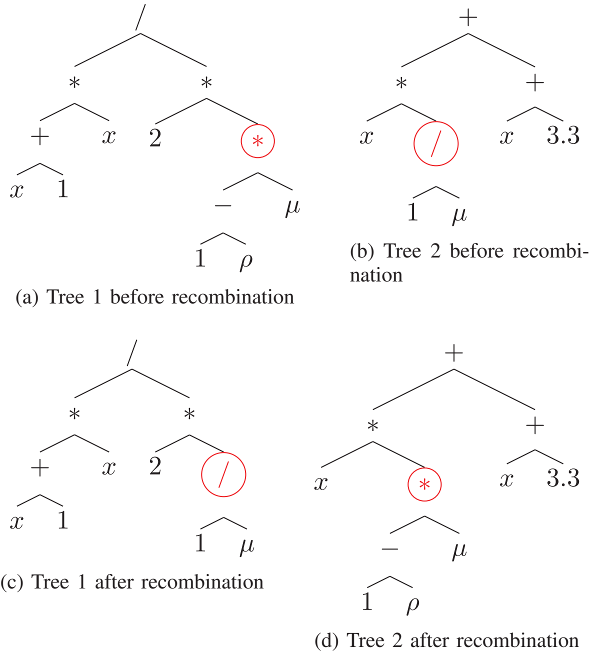

(a) Tree 1 before recombination
(b) Tree 2 before recombination
(c) Tree 1 after recombination
(d) Tree 2 after recombination

Fig. 2. The recombination operator illustrated on the two trees in Figs. 2a and 2b. The encircled subtrees are exchanged, resulting in the trees in Figs. 2c and 2d

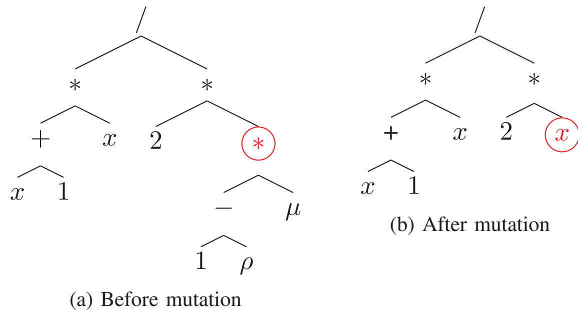

(a) Before mutation
(b) After mutation

Fig. 3. Mutation removes the subtree of the encircled node in Fig. 3a (representing the term $\mu(1-\rho)$) and replaces it by a randomly generated subtree. The new subtree contains, in this case, only the element x and is encircled in Fig. 3b

B. GP *recombination operator*

GP uses the recombination operator to generate two new offspring from two parents. The recombination operator takes the following two steps:

1) Randomly select a node in each of the two trees.
2) Exchange the two subtrees.

The procedure is applied in Fig. 2 to the two trees in Figs. 2a and 2b. The subtree with the encircled $*$ as root in Fig. 2a is exchanged with the subtree with root / (also encircled), resulting in the trees in Figs. 2c and 2d. This combines the functions $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$ and $V(x) = x\frac{1}{\mu} + x + 3.3$ to $V(x) = \frac{x(x+1)}{2\frac{1}{\mu}}$ and $V(x) = x(1-\rho)\mu + x + 3.3$.

C. GP *mutation operator*

Applying the GP paradigm with only the recombination operator already results in the desired improvement of the population over time. This improvement is, however, limited by the information present in the population at the start of the algorithm. The mutation operator discussed in this section is used by GP to insert new information into the population. The performance of GP is determined partly by carefully balancing the application of the mutation and recombination operators. Mutation of trees is done via the following procedure:

1) Select one of the nodes of the tree uniformly at random.
2) Remove this node and the subtree attached to it.
3) Randomly generate a new subtree.
4) Insert this new subtree in the place of the old subtree.

Fig. 3 illustrates the procedure for the tree for $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$ which we saw earlier, displayed again in Fig. 3a. The circled node is selected for mutation and removed from the tree, together with its subtree. It is replaced by a randomly generated subtree, in this case a simple tree with only one element (x). The result is shown in Fig. 3b, with the newly added tree encircled. Thus, mutation changes $V(x)$ from $\frac{x(x+1)}{2\mu(1-\rho)}$ to $\frac{x(x+1)}{2x}$.

IV. Value Function Discovery

The previous section illustrated how GP learns an algebraic description of a function. VFD relies on this technique to learn the value function of an MDP. To describe the algorithm we introduce some notation. The number of parameters of the MDP is denoted by m, the optimal value function of the MDP by $V(\cdot)$, and the VFD discovered function by $\widetilde{V}(\cdot)$.

A. *Preparing input from the* MDP

Before VFD starts, it requires input from the MDP in the form of sample point sets. Here, we describe how value iteration is used to generate sample points, but, generally speaking, other methods can be used as well. Sample points are generated using the following steps:

1) Generate random values for each of the m parameters.
2) Run value iteration for the MDP.
3) Select several sample points that together capture the shape of the value function. Each sample point is denoted by s, and the pairs $(s, V(s))$ together form the sample point set $\mathcal{S}_q$. The selection of sample points typically depends on the MDP.

4) Save these sample points into a file.
5) Repeat steps 1–4 for several combinations of the m parameters. Choosing the number of combinations is again MDP-specific, but we denote it by Q for now. This results in sample point sets $\mathcal{S}_q$, with $q \in [0, Q-1]$.

The purpose of having multiple sets is to allow VFD to position the parameters of the model. If we would only use one set, VFD could use the value of a parameter instead of the parameter itself and still discover a good value function. VFD would, however, most likely perform poorly on a set generated with different parameters.

Note that VFD starts by running value iteration on the MDP, which yields an optimal policy. So why not use this policy instead of running VFD? Well, the policy found by value iteration is numeric in nature, whereas VFD produces an algebraic policy. Consequently, the policy resulting from VFD can be applied to parameters that are not used to generate the sample points. This feature is illustrated later in this paper in Section VI-C.

If it is not possible to run value iteration, for instance when the MDP is too large, other techniques can be used to generate sample points as well. An example of this is TD-learning (see [8]), which provides numerical approximations of the value function using simulations.

B. Overview

A pseudo code listing of VFD is shown in Algorithm 1, and in the following paragraphs we describe the steps involved. We start with a high-level description of Algorithm 1, and then move on to a detailed description of the functions involved (Algorithms 2 and 3). During these descriptions we encounter the first of several parameters of VFD, which are listed in Table I (together with assigned values that we use later in an example MDP in Section V). Functions and parameters are written in SMALLCAPS throughout the text, including trailing brackets () for functions.

The algorithm starts on line 2 by loading the sample point sets of the MDP from the files. These are used later to determine the error of a tree. Next, the population is initialized by filling it with LAMBDA randomly generated trees. Lines 4–20 describe the steps taken by GP: first, MU children are generated using mutation and recombination (lines 6–11). Then, their error is calculated, they are added to the population, and the population is sorted from smallest error to largest (lines 13-15). Survivor selection removes LAMBDA trees from the population, leaving MU individuals (line 16). This procedure is repeated until convergence (line 4).

Repeating the GP-like procedure described above eventually leads to a population where most trees are the same or similar. When this happens, the algorithm loses its ability to learn and evolve, and the population is said to have lost *diversity*. VFD deals with this by checking the level of diversity in each generation (with the ISPOPULATIONDIVERSE() function in line 17). When this check indicates that too much diversity has been lost, VFD reinitializes the population (line 18) with random trees and restarts the search process. Upon convergence VFD returns the discovered tree (line 21).

Algorithm 1 Value function discovery (VFD)

```
1:  function VFD( )
2:      samplePointSets ← readSamplePointSets()
3:      population ← initPopulation()
4:      while not isConverged() do
5:          repeat
6:              if apply mutation then
7:                  children ← mutate(selectParent());
8:              else
9:                  children ←recombine(selectParent(),
10:                                     selectParent());
11:             end if
12:         until MU children generated
13:         setError(children)
14:         population ← population + children
15:         sort(population)
16:         survivorSelection()
17:         if not isPopulationDiverse() then
18:             initPopulation()
19:         end if
20:     end while
21:     return population[0]
22: end function
```

C. Mutation, recombination, diversity, and convergence

The remaining paragraphs in this section describe the functions used in Algorithm 1 in more detail. We start with the MUTATE() function on line 1 of Algorithm 2. Mutation occurs according to the GP paradigm, as described in Section III-C: a random point in the tree is selected (line 2) and the subtree at that point is replaced by a randomly generated subtree (lines 3 and 4). Similarly, the recombination operator from Section III-B is reflected in the RECOMBINE() method. Both functions rely on a numbering of the nodes in a tree, which VFD assigns using a root-left-right walk of the tree.

Every time that VFD generates one or two new individuals, it decides whether to use mutation or recombination. This is done probabilistically via the command line parameters APPLYMUTATIONPROB: with probability APPLYMUTATIONPROB VFD uses mutation, with probability 1-APPLYMUTATIONPROB it uses recombination.

Checking for diversity is done in ISPOPULATIONDIVERSE(). It finds the error of the best tree (the first in the population) and the worst tree (the last in the population) at lines 26 and 27 respectively. Diversity is then calculated via "error of worst tree - error of best tree"/ "error of best tree" at line 28, which is then compared to a threshold DIVERSITY_THRESHOLD, another parameter of VFD. If diversity drops below the threshold, diversity is considered to be lost (line 29).

The next function is INITPOPULATION(), which periodically reinserts diversity into the population. The entire population is cleared (line 17) and reinitialized with randomly

generated trees (lines 18–20). The final steps in lines 21 and 22 calculate the error of each tree and sort the population (on error). Readers familiar with GP most likely notice that VFD's treatment of diversity differs from common practice in GP. We added a paragraph on the reasons for this difference in Section VII-A.

The final function in Algorithm 2 is the ISCONVERGED() function, which determines whether the current best individual is good enough to allow stopping of VFD. If its error is lower than the threshold value MIN_ERROR (specified by the user), VFD stops.

Algorithm 2 VFD continued

```
 1: function MUTATE(parent)
 2:     z ← randint [0,numElements(parent)−1]
 3:     newSubtree ← generateRandomTree()
 4:     parent→setSubtree(z, newSubtree)
 5: end function
 6:
 7: function RECOMBINE(parent1, parent2)
 8:     z1 ← randint [0,numElements(parent1)−1]
 9:     z2 ← randint [0,numElements(parent2)−1]
10:     subTree1 ← parent1→getSubtree(z1)
11:     subTree2 ← parent2→getSubtree(z2)
12:     parent1→setSubtree(z1, subtree2)
13:     parent2→setSubtree(z2, subtree1)
14: end function
15:
16: function INITPOPULATION( )
17:     population ← List()
18:     for k ← 0, . . . ,LAMBDA−1 do
19:         population[k] ← generateRandomTree()
20:     end for
21:     setError(population)
22:     sort(population)
23: end function
24:
25: function ISPOPULATIONDIVERSE(population)
26:     min ← population[0]→getError()
27:     max ← population[MU−1]→getError()
28:     div ← (max-min)/min
29:     return div > DIVERSITY_THRESHOLD
30: end function
31:
32: function ISCONVERGED( )
33:     return population[0]→getError() < MIN_ERROR
34: end function
```

D. Bloat in GP

Note that if recombination exchanges, for instance, the root of the first tree with the leaf of the second tree, the second tree can increase in depth and in number of elements. Over time, this typically leads to very large and deep trees, with negative effects on both speed and memory usage. This problem, common to all GP instances, is called *bloat* and must be dealt with by VFD. It does this by enforcing a maximum on the number of elements in the tree, as specified by the command line parameter MAXELEMENTSINTREE. This feature is not shown in the MUTATE() and RECOMBINE() functions in Algorithm 2 to keep the listing readable, but it is present in the implementation of VFD. Additionally, the SORT() function, which sorts a given set of trees by error in ascending order, has a built-in preference for trees with a small number of elements. Specifically, if two trees have equal error, the sort function puts the tree with the fewest elements in front. This gives VFD a slight inclination to discover short trees and prevent bloat.

E. Parent selection and survivor selection

We continue with the SELECTPARENT() function in Algorithm 3, which is used by the mutation and recombination operators to determine which parent(s) to act upon. Following convention in the GP community, VFD relies on a strategy called *over-selection* when selecting parents. In this strategy the population is split into two groups, one containing 'good' parents and the other with 'bad parents'. The two groups are determined by taking the sorted population and defining the first 'GOODPCT' percent individuals as good parents, and the remaining trees as bad parents.

The parameter GOODPCT is automatically determined by VFD from the size of the population MU. For this, VFD again relies on GP-conventions and uses values ranging from $4 - 32\%$, as described in [5, Table 6.4]. Once the split point z_1 is known (line 2), a parent is selected from the good parents with probability SELECTFROMGOODPROB and from the bad parents otherwise. The selection is done in lines 4 and 6. Note that for recombination the SELECTPARENT() function is called twice.

The SURVIVORSELECTION() function is used by VFD in each generation after the LAMBDA children have been generated. Its purpose is to select MU survivors from among the MU+LAMBDA individuals currently in the population. VFD uses a greedy approach and simply removes the LAMBDA children with the worst error from the population (line 12).

F. Goodness of fit (error)

So far we have not yet discussed how the error of a tree is defined. This definition ties the GP approach of VFD to the MDP setting of finding a good value function. The error of a tree must be chosen in such a way that a low error corresponds to a good fit of the function described by the tree on the sample points obtained from the MDP. For VFD the error $\mathcal{E}_q$ on sample point set q is calculated via

$$\mathcal{E}_q = \max_{(s,V(s))\in\mathcal{S}_q} \frac{|\widetilde{V}(s) - V(s)|}{V(s)}. \tag{1}$$

Here, $\widetilde{V}(\cdot)$ is the function discovered by VFD and $V(\cdot)$ the optimal value function found by value iteration. The error $\mathcal{E}_q$ is calculated in the function CALCERROR() in line 19 in Algorithm 3. The error of a tree is then defined as

$$\mathcal{E} = \max_{q\in[0,Q-1]} \mathcal{E}_q,$$

Algorithm 3 VFD continued

```
1: function SELECTPARENT( )
2:     z_1 ← floor(MU·GOODPCT)
3:     if select from good then
4:         z_2 ← randint[0, z_1 − 1]
5:     else
6:         z_2 ← randint[z_1, MU−1]
7:     end if
8:     return population[z_2]
9: end function
10:
11: function SURVIVORSELECTION(population)
12:     remove population[MU:MU+LAMBDA−1]
13: end function
14:
15: function SETERROR(trees)
16:     for tree in trees do
17:         maxError ← 0
18:         for q ← 0, . . . , Q − 1 do
19:             err ← calcError(samplePointSets[q], tree)
20:             maxError ← max (err, maxError)
21:         end for
22:         tree→setError(maxError)
23:     end for
24: end function
```

i.e., the error of the tree is its worst error achieved on all the sample point sets (see also steps 15-24 of function SETERROR() in Algorithm 3).

The error in Eq. (1) uses a relative measure of error by dividing by $V(s)$, contrary to, e.g., the mean squared error. This ensures that sample points that naturally have large values for $V(s)$ do not dominate the search process of VFD. Also, we use "$\max_{(s,V(s))\in\mathcal{S}_q}$" rather than "$\text{mean}_{(s,V(s))\in\mathcal{S}_q}$" (i.e., MAPE). With MAPE, a large relative error for a small sample point s can be mitigated by a small relative error of large sample points. In the context of MDPs, however, small states are usually visited more often, so we require a better fitting value function in such states. At larger states we want to allow larger errors. Therefore, using "$\max_{(s,V(s))\in\mathcal{S}_q}$" in VFD is preferable over MAPE.

V. Example MDP

We illustrate the configuration, application, and output of VFD on an example MDP. This MDP is suitable for demonstrating VFD because

- No known expression for the optimal policy or value function exists, so we have no prior knowledge that VFD can capture the optimal value function.
- The system resembles a combination of an $M/M/1$ and $M/M/2$ system, which helps us when generating sample point sets and when choosing MAXELEMENTSINTREE.
- The system is relatively simple and easy to understand.
- The state space is small, which keeps run times of VFD short.

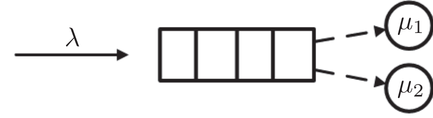


Fig. 4. An M/M/2 system with control, where jobs (arriving with rate λ) from the queue have to be assigned to either a fast server S_1 (with service rate μ_1) or to a slow server S_2 (with service rate $\mu_2 < \mu_1$)

In the following paragraphs we describe how VFD is configured and, in doing so, we have to choose the parameters of VFD. Table I contains all parameters available to VFD, and the values we assign to them in this section. When choosing these values, the aim is to discover a value function for the system with the objective to obtain a near-optimal policy. Thus, in particular, we are not looking for the best parameter settings (which we postpone to future research).

A. Model formulation

Fig. 4 shows a queue with Poisson arrivals (rate λ) and two servers with exponential service rates μ_1 and μ_2 ($\mu_1 > \mu_2$). The jobs in the queue have to be assigned non-preemptively to either the fast server (S_1) or the slower server (S_2), assuming one is available. This decision is taken after a job completion, as well as after a job arrival. We model this scenario as an MDP, with state $(x, i) \in \mathcal{X} = \mathbb{N} \times \{0, 1\}$. Here, x denotes the number of jobs in the queue and S_1, and i the number of jobs in S_2. Our aim is to minimize the average number of jobs in the system. From [15] we have the optimality equation

$$\begin{aligned} g + V(x, i) = x &+ i \\ &+ \lambda W(x + 1, i) \\ &+ \mu_1 W((x - 1)^+, i) \\ &+ \mu_2 W(x, 0), \end{aligned} \tag{2}$$

with

$$\begin{aligned} W(x, 0) &= \min\{V(x, 0); V(x - 1, 1)\} \quad \text{if } x > 0, \\ W(0, i) &= V(0, i), \\ W(x, 1) &= V(x, 1). \end{aligned} \tag{3}$$

The function $W(x, i)$ reflects the decision to be taken after the occurrence of an event. In particular, if S_2 is empty the decision is between leaving the job in the queue ($V(x, 0)$) or moving one job from the queue to S_2 ($V(x-1, 1)$), as shown in Eq. (3). If the queue and S_1 are empty then moving a job is not possible and the state of the system does not change ($W(0, i) = V(0, i)$). Also, if the second server is busy the state does not change ($W(x, 1) = V(x, 1)$). In Eq. (2), the first line reflects the number of jobs in the system ($x+i$). The second, third, and fourth line correspond to the decision upon a job arrival, a job completion at S_1, and a job completion at S_2, respectively. Finally, the constant g is the time-average costs of the system.

Note that this formulation allows preemptive behavior, since $W(1, 0) = \min\{V(1, 0); V(0, 1)\}$ can result in moving a job in service at S_1 to S_2. However, since $\mu_1 > \mu_2$ and rates are exponential, such a move would result in a longer

TABLE I
THE PARAMETERS AVAILABLE TO VFD, THE VALUES ASSIGNED TO THEM FOR THE EXAMPLE MDP IN SECTION V, AND THE VALUES ALLOWED BY VFD

Parameters Name	**In example**	**Allowed values**
Command line		
SEED	3151492	[0,MAXINT]
MU	1000	[1,MAXINT]
LAMBDA	500	[1,MAXINT]
MAXELEMENTSINTREE	125	[1,MAXINT]
MIN_ERROR	0.2	[0,1]
APPLYMUTATIONPROB	0.2	[0,1]
DIVERSITY_THRESHOLD	0.01	[0,MAXDOUBLE]
Parent selection		
GOODPCT	0.32	[0,1]
SELECTFROMGOODPROB	0.8	[0,1]
Random tree creation		
PROB_PLUS	0.3	[0,1]
PROB_MINUS	0.3	[0,1]
PROB_MULTIPLY	0.3	[0,1]
PROB_DIVIDE	0.1	[0,1]
PROB_PARAMETER	0.45	[0,1]
PROB_VARIABLE	0.45	[0,1]
PROB_CONSTANT	0.1	[0,1]
MAXCONSTANTVALUE	1	[0,MAXDOUBLE]

expected service time for the job than when it is left at S_1. Hence, the optimal policy automatically enforces non-preemptive behavior. Finally, in Eq. (2) and (3) we assume that the parameters are normalized such $\lambda + \mu_1 + \mu_2 = 1$.

B. Generating sample point sets

The first step to running VFD is preparing the sample point sets. The steps were described in Section IV-A and we repeat them here for convenience:

1) Generate random values for each of the 3 parameters.
2) Run value iteration for the MDP.
3) Select sample points that together capture the shape of the value function.
4) Save these sample points into a file.
5) Repeat steps 1–4 for Q pairs of the 3 parameters.

First we decide upon the number of sample point sets Q that we intend to generate, and on the parameters used to generate these sets. The purpose of having multiple sets is to allow VFD to position the parameters of the model in the discovered value function. If we would only use one set, VFD could use the value of a parameter instead of the parameter itself and still achieve a good fit on the single set. VFD would, however, most likely perform poorly on a set generated with different parameters.

For the current MDP, we make our choice for a worst-case scenario where S_2 is never used (i.e., a $M/M/1$ system) and choose parameters for the sample point sets based on the load $\rho_1 = \lambda/\mu_1 \in [0, 1]$. In the region $0 \le \rho_1 \le 0.4$ the load on the system is low, and possible wrong decisions in a policy have little impact. Hence, we expect that an accurate value function in that region is not required, and we cover it by just two sample point sets: one at $\rho_1 = 0.1$ and another at $\rho_1 = 0.4$. Short experiments with VFD indicate that this is indeed sufficient. Following similar reasoning, we choose two sample point sets 'close together' at $\rho_1 = 0.9$ and $\rho_1 = 0.95$ to cover scenarios with a high load. The region $0.4 < \rho_1 < 0.9$ is then covered by $Q - 4$ sample sets distributed evenly over the interval. We did short experiments with Q ranging from 5 until 9, and using $Q = 7$ yielded the best policies. The resulting ρ_1-values are $\{0.1, 0.4, 0.525, 0.65, 0.775, 0.9, 0.95\}$. Then, we generate parameters μ_1 and μ_2 uniformly from $[0, 1]$, and set $\lambda_1 = \rho_1\mu_1$. In generating these values we also ensure that $\mu_1 > \mu_2$ and that $\lambda + \mu_1 + \mu_2 = 1$. The parameters of each set are shown in Table II.

TABLE II
MODEL PARAMETERS PER SAMPLE POINT SET

Set	ρ_1	λ	μ_1	μ_2
0	0.100	0.0814	0.8135	0.1051
1	0.400	0.2688	0.6719	0.0594
2	0.525	0.3158	0.6015	0.0827
3	0.650	0.3701	0.5693	0.0606
4	0.775	0.4028	0.5198	0.0774
5	0.900	0.4662	0.5180	0.0159
6	0.950	0.4804	0.5057	0.0139

Note that, generally speaking, using many sample point sets (i.e., a large Q) ensures that VFD discovers a well-fitting value function. On the other hand, the points in each sample point set are used many times to evaluate trees, contributing significantly to the computational complexity. Moreover, VFD has to discover a value function that closely fits each sample point set, so using many sets increases the time needed by VFD to discover such a function. Consequently, choosing Q is a trade-off between the goodness of fit of the discovered value function, and the run time of VFD.

Now that the number of sets is chosen, the sample points in each set can be found by value iteration. To run value iteration we must decide on a boundary for the first dimension of the state space $\mathcal{X} = \mathbb{N} \times \{0, 1\}$. We use a value L to limit the state space to $\hat{\mathcal{X}} = [0, L] \times \{0, 1\}$, where L is the smallest value such that $\mathbb{P}(x > L) < 0.001$ in the worst case $(M/M/1)$ scenario. For each sample set we then take 2×10 points, with 10 x-values evenly distributed over $[0, 0.75 \cdot L]$ and i both 0 and 1. Here, $(x, i) \in \hat{\mathcal{X}}$ is a point in the state space. These sample points easily capture the shape of the value function and avoid boundary effects of value iteration (by using $\lceil 0.75 \cdot L \rceil$ instead of L). If $0.75 \cdot L < 10$ then we take only $\lceil 0.75 \cdot L \rceil$ points instead of 10. Finally, we stop value iteration once the span of two consecutive iterations is less than 10^{-6}.

With this choice of sample points, the part of the state space outside $\hat{\mathcal{X}}$ is not covered by sample points. Most likely, VFD will not discover a value function that extrapolates well

outside $\hat{\mathcal{X}}$. By choosing L such that $\mathbb{P}(x > L) < 0.001$, we ensure that it is unlikely that the system reaches states outside $\hat{\mathcal{X}}$, and thus we minimize the effects of VFD's inability to extrapolate. In general, when applying VFD to an MDP, the user should keep in mind that VFD is good at interpolating between sample points, and not at extrapolating. Hence, the sample points should cover the area in the state space that the user is most interested in. A similar argument holds for the placement of the Q sample point sets in the parameter space.

C. Determining command line parameters

The next step towards running VFD is determining the command line parameters, as listed in the first part of Table I. The first of these, SEED, can be set to any desired integer value, as its only purpose is to initialize the random number generator. For the population size MU and the number of children LAMBDA we follow current trends in GP and choose them such that LAMBDA<MU. The authors of [5] suggest populations with several thousands of individuals, but since our MDP is fairly small we conservatively set MU to 1000 and LAMBDA to 500.

For the parameter MAXELEMENTSINTREE we manually count the number of elements needed for the $M/M/1$ value function (13) and the $M/M/2$ value function (≈ 90), based on the expressions in [16]. Then, we set MAXELEMENTSINTREE to a value somewhat higher than 90 (125), and ran some short experiments to see how large the resulting trees where. These experiments suggest that using 125 elements is sufficient. In general it is wise to set MAXELEMENTSINTREE to a slightly bigger value than expected, since that gives VFD some more freedom. Also, the SORT() function prefers smaller trees, so this tends to counteract a possibly too large value of MAXELEMENTSINTREE.

Next is MIN_ERROR, which influences the stopping criterion of VFD. Large values for MIN_ERROR let VFD stop quickly (but with a badly fitting tree), smaller values allow VFD to search longer (with a better fitting tree). Note that for the current MDP the performance of a discovered value function depends on the decision $\min\{\widetilde{V}(x,0); \widetilde{V}(x-1,1)\}$. Even if $\widetilde{V}(x,i)$ is not very accurate, the decision can still be correct. Hence, we choose MIN_ERROR quite large and set MIN_ERROR$= 0.20$.

The value of DIVERSITY_THRESHOLD is determined by visually observing the progress made by VFD in terms of error in several short experimental runs. VFD should have sufficient time to discover good functions in between reinitialisations of the population, but should stop as soon as error stops decreasing significantly. This means that DIVERSITY_THRESHOLD should not be too high. After some experiments we set it to 0.01, i.e., diversity is lost when the worst tree differs by at most 1% from the best tree (in terms of error).

Finally, we discuss the parameter APPLYMUTATIONPROB, which is used by VFD to decide between using the mutation or recombination operators. The GP literature (see [5, Sec. 6.4] and the references therein) suggests using a small mutation probability in the order of 0.05. However, experiments on the current MDP (see Section VI-E) indicate that setting APPLYMUTATIONPROB to 0.2 yields better results.

D. Parent selection parameters

As described, parent selection utilizes a strategy called over-selection which relies on parameters GOODPCT and SELECTFROMGOODPROB. For these we rely on conventions from the GP community, as specified in [5]. Parameter GOODPCT depends on the population size, which for our MDP results in 0.32, and SELECTFROMGOODPROB is typically set to 0.8.

E. Random tree generation parameters

Generating random trees depends on parameters for determining the type of operator and the contents of leafs (model parameter, variable, or constant). Test runs indicate that the operator $/$ does not need to occur that often, so the values in Table I reflect this (PROB_DIVIDE=0.1, whereas the others are set to 0.3). Similar test runs suggest that constants are needed less often, so PROB_CONSTANT is set to 0.1 and the two others are set to 0.45. Parameter MAXCONSTANTVALUE is set to 1, since by combining constants using the operators $/, *, +, -$ each value in $\mathbb{R}$ can be attained.

VI. Numerical Results

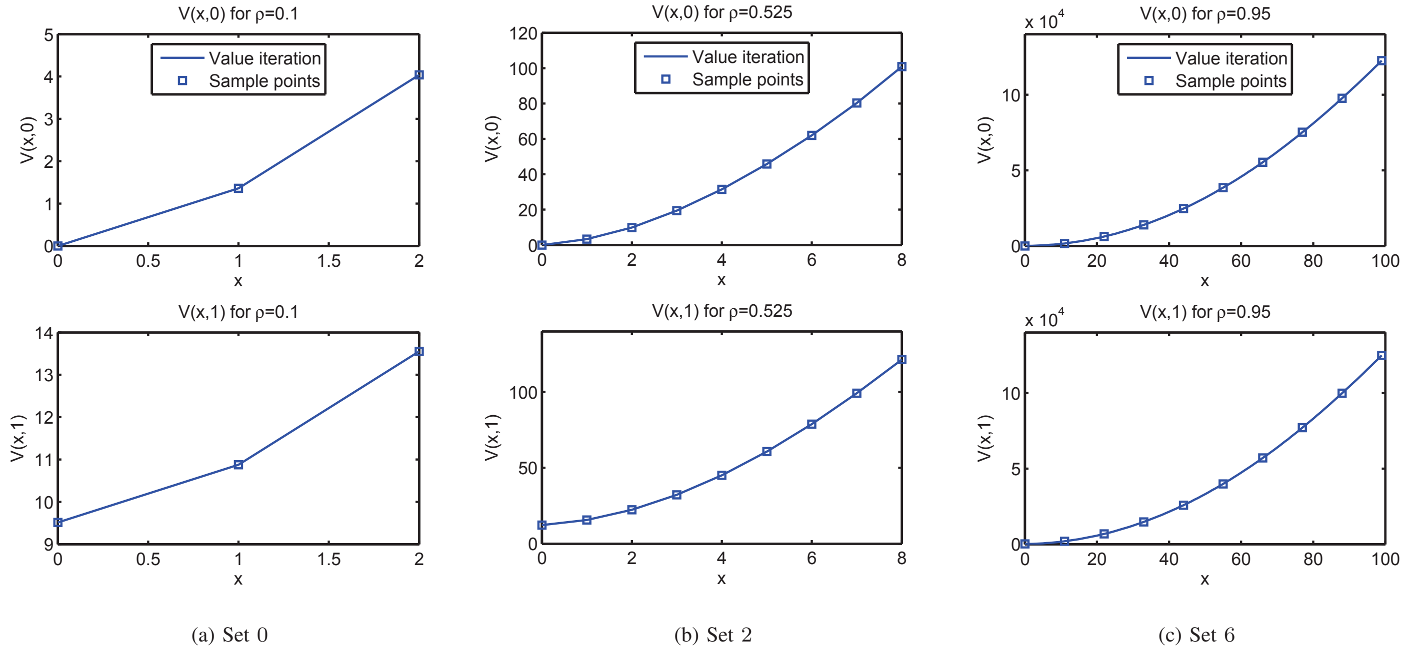
A. Sample points

Section V-B describes how the sample points for our MDP example are generated. The values for the model parameters per sample set are outlined in Table II. For each of the model parameters in Table II we then run value iteration to find the sample points. Fig. 5 shows the resulting sample points for several of the sets. Note that the system with high load (Fig. 5c) the optimal value function attains values in the order of 10^4, whereas for lower loads in Figs. 5a and 5b these values are significantly smaller. Also, in Fig. 5a the boundary $\lceil 0.75 \cdot L \rceil$ for value iteration is smaller than the number of desired sample points (10), in which case only $\lceil 0.75 \cdot L \rceil$ sample points are retained. This results in 3 sample points for both $i = 0$ and $i = 1$ (i.e., 6 sample points in total).

B. The discovered value function

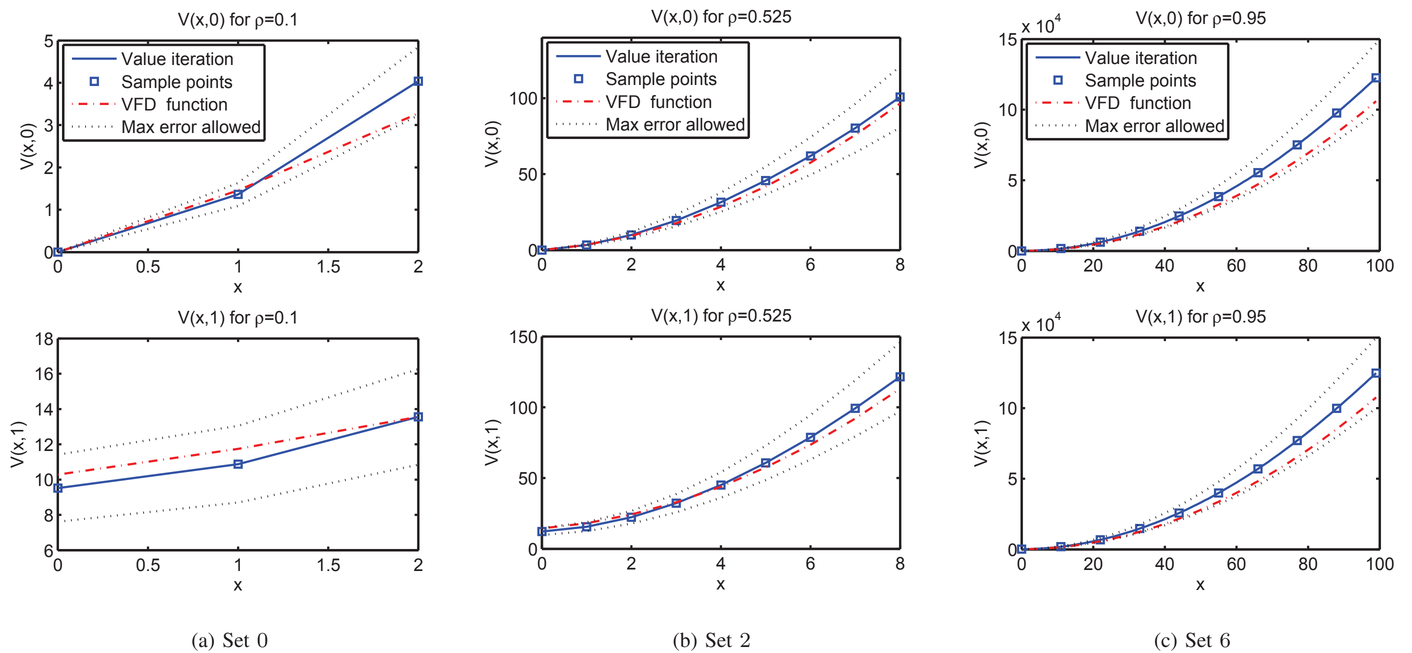
Having specified all the input for VFD, it is ready to run. The value function $\widetilde{V}(x,i)$ discovered by VFD is shown in Eq. (4). In Fig. 6 it is plotted (dash-dotted line) together with the sample points for the same sets as in Fig. 5. The discovered $\widetilde{V}(x,i)$ resembles $V(x,i)$ well. Additionally, the figure contains two lines (dotted) above and below the sample points that indicate how much $\widetilde{V}(x,i)$ is allowed to differ from the sample points, as specified by the error criterion in Eq. (1) and by the parameter MIN_ERROR. When running, VFD continues looking for a value function until one is found that lies completely between this upper and lower bound. Fig. 6 demonstrates that $\widetilde{V}(x,i)$ indeed lies between the specified bounds. By modifying the parameter MIN_ERROR, the user of VFD can control the distance between the upper and lower bounds, and thus the accuracy of $\widetilde{V}(x,i)$. Also, observe that the distance between the upper and lower bound increases as x gets larger, as a consequence of our choice for a relative error criterion (as discussed in Section IV-F).

$$\widetilde{V}(x,i) = \frac{i}{0.28\mu_2(2\lambda\mu_2(i+\mu_1)(2\lambda+\mu_1)-i+\mu_2)\left((i+\lambda)\left(\frac{\lambda^2}{\mu_1}+\mu_2\right)+i-\mu_1\right)+\mu_2} + x \dots$$
$$-\lambda\left(\lambda^2+1\right)x\left(-\lambda\left(\frac{\lambda^2\left(\frac{3.58i\lambda}{\mu_1}+3.58\lambda^2 x+x\right)}{\mu_2}+x\right)+\lambda^2-3.58(\lambda+\mu_1)-3.58\lambda x-\mu_1 x-2\mu_2-x\right) \quad (4)$$



(a) Set 0 (b) Set 2 (c) Set 6

Fig. 5. Sample points of sets 0, 2, and 6



(a) Set 0 (b) Set 2 (c) Set 6

Fig. 6. $\widetilde{V}(x,i)$ for sets 0, 2, and 6

TABLE III
POLICY DERIVED FROM THE VALUE FUNCTION IN EQ. (4) DISCOVERED BY VFD. THE TABLE INDICATES FOR WHICH STATES $(x, 0)$ A JOB SHOULD BE ASSIGNED TO SERVER S_2

Set	ρ_1	Policy
0	0.100	Use S_2 if $x > 25.5719$
1	0.400	Use S_2 if $x > 10.2648$
2	0.525	Use S_2 if $x > 5.9715$
3	0.650	Use S_2 if $x > 6.4751$
4	0.775	Use S_2 if $x > 4.6315$
5	0.900	Use S_2 if $x > 15.5992$
6	0.950	Use S_2 if $x > 17.4802$

TABLE IV
TIME-AVERAGE COSTS $\tilde{g}$ FOR THE POLICY BASED ON THE VALUE FUNCTION IN EQ. (4) DISCOVERED BY VFD. THESE COSTS ARE COMPARED TO COSTS g OF THE OPTIMAL POLICY

Set	ρ_1	g	$\tilde{g}$
0	0.100	0.1107	0.1107
1	0.400	0.6643	0.6643
2	0.525	1.0589	1.0665
3	0.650	1.7107	1.7368
4	0.775	2.4684	2.5085
5	0.900	7.3973	7.7279
6	0.950	12.8241	13.5369

Next, we convert $\widetilde{V}(x, i)$ to a (algebraic) policy using one-step policy improvement. Observe that for states $(x, 1)$ it is not possible to assign a job to server S_2, so the policy is trivial in these states. Therefore, we focus on states $(x, 0)$. To obtain the policy, we take the term $\min\{V(x, 0); V(x-1, 1)\}$ in Eq. (3) and substitute $\widetilde{V}(x, i)$ for $V(x, i)$. Evaluating the minimum results in an action for each state $(x, 0)$, i.e., server S_2 is used when $\widetilde{V}(x, 0) > \widetilde{V}(x-1, 1)$. Unfortunately, the resulting inequality is lengthy and challenging to interpret. Instead, we simplify the inequality for parameters λ, μ_1, μ_2 of the sample point sets in Table II, and show the policies in Table III. The policies indicate for which states $(x, 0)$ the second server S_2 should be used. All policies are of threshold type, and the same structure holds for the optimal policy (see [15] for a proof).

For the policy derived from $\widetilde{V}(x, i)$ we can find the time-average costs $\tilde{g}$ with policy evaluation for each parameter combination from Table II. The results are in Table IV, and show that the policy consistently yields good results for the various model parameter values.

C. Performance on different model parameters

The time-average costs in Table IV are based on the model parameters in Table II, which were given to VFD. To further investigate the performance of VFD, we again compute the time-average costs $\tilde{g}$ of the policy based on $\widetilde{V}(x, i)$, but now for model parameters that VFD has not seen before. To this end, we fix new values for ρ_1 (the second column in Table V) and generate new values for the model parameters λ, μ_1, and μ_2 (columns $3-5$). Then, we rerun value iteration to get the costs g of the optimal policy, and apply policy evaluation to find the costs $\tilde{g}$ of the policy based on $\widetilde{V}(x, i)$ from Eq. (4). The last two columns of Table V shows that g and $\tilde{g}$ are consistently close and that VFD performs well on these new model parameters. We repeated this experiment several times for several values of the parameter SEED, and VFD continually yielded similar good results.

TABLE V
TIME-AVERAGE COSTS $\tilde{g}$ FOR THE POLICY BASED ON THE VALUE FUNCTION DISCOVERED BY VFD, COMPARED TO COSTS g OF THE OPTIMAL POLICY. THE MODEL PARAMETERS (λ, μ_1, μ_2) AND LOADS (ρ_1) ARE DIFFERENT FROM THE ONES VFD WAS GIVEN AS INPUT

Set	ρ_1	λ	μ_1	μ_2	g	$\tilde{g}$
0	0.010	0.0088	0.8832	0.1080	0.0101	0.0101
1	0.200	0.1533	0.7663	0.0805	0.2496	0.2496
2	0.300	0.2094	0.6981	0.0924	0.4270	0.4270
3	0.450	0.2848	0.6329	0.0823	0.8067	0.8100
4	0.600	0.3686	0.6143	0.0171	1.4930	1.4930
5	0.700	0.3823	0.5462	0.0715	1.9669	2.0080
6	0.825	0.4443	0.5385	0.0172	4.3761	4.4744
7	0.875	0.4567	0.5219	0.0215	5.7497	5.9840
8	0.925	0.4571	0.4942	0.0487	5.8536	6.0514

D. Computational complexity

With the model parameter values from Table II and the sample point sets from Fig. 5 VFD requires 2 minutes and 7 seconds to discover $\widetilde{V}(x, i)$ from Eq. (4). Since VFD relies on several sources of randomness (controlled via command line parameter SEED), we inspect whether this run time is representative of VFD in general. To this end, we run VFD for 25 different values of SEED, record the run times, and compute the median of these run times. This results in a median run time of 2 minutes and 21 seconds, which corresponds well with the previously observed run time.

For the MDP in this paper the run time is quite short, which is mainly due to the small state space of the MDP in Eq. (2). On MDPs with larger state spaces the run time will be longer, but we feel that this is well worth the effort. Obtaining near-optimal policies for large MDPs via mathematical procedures is very challenging, time consuming, and does not always yield results. VFD, however, is easy to set up and run.

E. Sensitivity analysis of APPLYMUTATIONPROB

For the experiments in this paper we use APPLYMUTATIONPROB$=$ 0.2, even though the GP literature suggest a lower value of 0.05. To illustrate why we deviate from GP conventions, we analyze the effect of changing APPLYMUTATIONPROB on VFD. Note that APPLYMUTATIONPROB only affects the run time of VFD, but not the goodness of fit of the discovered value function. The latter is controlled with parameter MIN_ERROR, and with the sample point sets. Therefore, we analyze the effect of APPLYMUTATIONPROB

TABLE VI
MEDIAN AND STANDARD DEVIATION (σ) OF THE RUN TIME OF VFD OVER 25 RUNS, FOR SEVERAL DIFFERENT VALUES OF APPLYMUTATIONPROB.

value	median	σ	value	median	σ
0	4.93	6.07	0.30	5.40	5.51
0.05	5.57	7.18	0.40	6.07	6.91
0.10	11.90	6.56	0.50	11.02	8.65
0.15	3.37	4.42	0.60	6.02	16.21
0.20	2.35	3.57	0.70	5.62	9.66
0.25	3.18	2.31	0.80	6.34	7.95

on the run time of VFD. We vary APPLYMUTATIONPROB from 0 to 0.8, as seen in the first and fourth column of Table VI. Then, for each value of APPLYMUTATIONPROB we run VFD 25 times and record the median run time (second and fifth column) and the standard deviation σ (third and sixth column). The lowest median run time is achieved for APPLYMUTATIONPROB$= 0.2$, and the corresponding standard deviation is low as well. Hence, APPLYMUTATIONPROB$= 0.2$ is the value we use in this paper.

F. VFD *applied to* $M/M/1$

In Section II we claimed that for MDPs that allow for an explicit closed-form expression of the optimal value function, VFD can find this optimal value function. To illustrate this, we let VFD discover the value function of a $M/M/1$ queue. Thereto, we set $\mu_2 = 0$, regenerate the sample point sets, and run VFD with parameter MIN_ERROR set to 0.0001 (slightly bigger than 0 to allow for small numerical inaccuracies in value iteration). VFD discovers the function

$$\widetilde{V}(x) = \frac{x(\lambda + \mu + x)}{-2\lambda + 2\mu},$$

which simplifies to

$$\widetilde{V}(x) = \frac{x(x+1)}{2(\mu - \lambda)}.$$

This is indeed the value function of a $M/M/1$ queue.

VII. DISCUSSION

A. *Improvements to* VFD

The paragraphs below contain several potential improvements to VFD. In this paper we showed the value function discovered by VFD in Eq. (4), but we did not analyze it further. It can, however, provide useful insights. For instance, $\widetilde{V}(x, i)$ in Eq. (4) contains the element λ/μ_1, the load of an $M/M/1$ system. It does, however, not contain $\frac{\lambda}{\mu_1+\mu_2}$, the load on an $M/M/2$ system. At the moment it is quite difficult to interpret the discovered value function, because the expression in Eq. (4) is somewhat long. We even expect that it is acceptable to sacrifice some accuracy in return for shorter trees.

VFD does not utilize any prior knowledge about the structure of the value function in the population. However, it might speed up the search process or result in better value functions if this knowledge is included. For the MDP in this paper, we could for instance add several elements of the $M/M/1$ and $M/M/2$ value function to the population: λ/μ_1 (the load of an $M/M/1$ system), $\lambda/(\mu_1 + \mu_2)$ (the load of an $M/M/2$ system), and x^2 (both the $M/M/1$ and $M/M/2$ value functions are quadratic in x).

A modification of VFD might eliminate the need for sample point sets before the start of the algorithm. If we make VFD work directly with the MDP optimality equations and construct a suitable error measure, then VFD does not need sample points anymore.

The current version of VFD uses only operators $/, *, +, -$, but the representation of a function in GP is flexible enough to also allow for, e.g., exponents, square roots, logarithms, and rounding. Additionally, we could add other genetic operators besides mutation and recombination, such as dropping and inserting nodes.

In Section V we determined values for the parameters of VFD. We wanted to set the parameters of VFD to values that yield good policies. In particular, we were not looking for the best parameter settings. The current, basic, MDP does not require too much consideration for the VFD parameters, but for larger systems we expect the parameter values to be more important. A potential improvement is to use a parameter tuning tool such as Bonesa [17] to select good parameters, or to learn parameters on the go with, e.g., a co-evolutionary algorithm (see [18] for an example).

The current setup of VFD reinitializes the entire population when diversity is lost, so it does not attempt to maintain diversity of a population. Upon loss of diversity the search is simply restarted elsewhere. With the basic MDP we used in this paper, such a naive attitude towards diversity is sufficient to get a good value function quickly. However, for MDPs with larger state spaces, or MDPs that require a smaller error, this approach will most likely not yield a sufficiently good value function in a reasonable amount of time. Traditionally, GP algorithms employ a diversity maintenance scheme, e.g., a temporary increase of APPLYMUTATIONPROB upon loss of diversity. We expect that VFD will also need a diversity maintenance strategy, as we continue our experiments with VFD in the near future. For the current paper we decided not to include such a scheme, because that would have resulted in even more parameters for VFD. This would have clouded our focus on discovery of value functions in the context of MDPs.

B. *Applications of* VFD

Recall that VFD yields an algebraic expression for a value function, and consequently for a policy as well. As stated in the introduction, this is particularly useful for large time-varying systems that require a control policy, since there is no need to make and analyze a time-dependent model. An example of such a system is presented in [19].

VFD discovers a value function, which is then turned into a policy. For the current MDP, however, another approach might speed VFD up. In [15] the author proves that the optimal policy is a switching curve, i.e., there exists a threshold T such that only S_1 is used for $x \leq T$ and both S_1 and S_2 are used

for $x > T$. We can thus apply VFD to sample points of this threshold T and learn an expression for T in terms of the model parameters. Note that this requires only one sample point per set and thus significantly improves the run time of VFD. Initial experiments suggest that this approach works well.

Applying VFD can also be convenient in other situations than when searching for a value function. For instance, in [20] the authors of the current paper apply one-step policy improvement to an MDP, starting with a Bernoulli policy. This policy includes a parameter α that must be determined after policy improvement, ideally such that the time-average costs $g'(\alpha)$ are minimized. An expression for $g'(\alpha)$ is, however, not available so the minimization is done using a numerical heuristic. VFD can be applied in this scenario to sample points of $g'(\alpha)$ and thus help discover an expression for $g'(\alpha)$, which can then be minimized with respect to α.

VIII. Conclusions

In this paper we introduced VFD, a novel method for discovering algebraic descriptions of value functions of MDPs using a GP approach. We started with a description of GP, in particular of the representation used in GP, and of the mutation and recombination operators. Then we gave a high-level overview of the workings of VFD, followed by a more detailed treatment of the algorithm. We applied VFD to a basic yet interesting MDP, and let it discover a value function. To illustrate how a discovered value function can be used, we obtained a policy from it via one-step policy improvement. Numerical experiments showed that this policy has near-optimal performance, both for model parameters that VFD was given a priori, and for new parameters. We identified several opportunities for future research, containing both improvements to VFD and alternative applications of the algorithm.

Acknowledgment

We thank SURFsara [21] for the support in using the LISA Compute Cluster, and the reviewers for their in-depth comments during the peer-review process.

References

[1] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[2] J. Walrand, *An introduction to queueing networks*. Prentice Hall Englewood Cliffs, N.J, 1988.

[3] J. M. Norman, *Heuristic procedures in dynamic programming*. Manchester University Press, 1972.

[4] H. Tijms, *A First Course in Stochastic Models*. Wiley, 2003.

[5] A. E. Eiben and J. E. Smith, *Introduction to evolutionary computing*. Berlin Heidelberg New York: Springer, 2003.

[6] R. Poli and J. Koza, *Genetic Programming*. Springer, 2014.

[7] D. Simon, *Evolutionary optimization algorithms*. John Wiley & Sons, 2013.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[9] H. S. Chang, H. Lee, M. C. Fu, and S. I. Marcus, "Evolutionary policy iteration for solving Markov decision processes," *IEEE Transactions on Automatic Control*, vol. 50, no. 11, pp. 1804–1808, 2005.

[10] J. Hu, M. C. Fu, V. R. Ramezani, and S. I. Marcus, "An evolutionary random policy search algorithm for solving Markov decision processes," *INFORMS Journal on Computing*, vol. 19, no. 2, pp. 161–174, 2007.

[11] A. Yener and C. Rose, "Genetic algorithms applied to cellular call admission: local policies," *IEEE Transactions on Vehicular Technology*, vol. 46, no. 1, pp. 72–79, 1997.

[12] 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*, 1999.

[13] 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*, vol. 16, no. 1, pp. 27–38, 2004.

[14] C. Gearhart, "Genetic programming as policy search in Markov decision processes," *Genetic Algorithms and Genetic Programming at Stanford*, pp. 61–67, 2003.

[15] G. Koole, "A simple proof of the optimality of a threshold policy in a two-server queueing system," *Systems & Control Letters*, vol. 26, no. 5, pp. 301–303, 1995.

[16] S. Bhulai and G. Koole, "On the structure of value functions for threshold policies in queueing models," *Journal of Applied Probability*, pp. 613–622, 2003.

[17] S. Smit and A. Eiben, "Multi-problem parameter tuning using BONESA," in *Artificial Evolution*, 2011, pp. 222–233.

[18] C. M. Fernandes, J. J. Merelo, and A. C. Rosa, "Controlling the parameters of the particle swarm optimization with a self-organized criticality model," in *Parallel Problem Solving from Nature-PPSN XII*. Springer, 2012, pp. 153–163.

[19] 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.*, vol. 24, no. 1, pp. 27–45, Jan. 2010.

[20] M. Onderwater, S. Bhulai, and R. D. v. d. Mei, "On the control of a queueing system with aging state information," *Stochastic Models (under review)*, 2014.

[21] SURFsara, "http://www.surfsara.nl," 2013.