6

Value Function Discovery in Markov Decision Processes with Evolutionary Algorithms

In this chapter we introduce a novel method for discovery of relative 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 relative 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 relative 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 chapter is a first step towards exploring potential usage scenarios of discovered relative 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 relative value function that closely resembles the optimal relative value function. The discovered relative value function is then used to obtain a policy, which we compare numerically to the optimal policy of the MDP. The resulting policy has near-optimal performance on a wide range of model parameters. Finally, we identify and discuss future application scenarios of discovered relative value functions.

This chapter is based on the results presented in [5].

6.1 Introduction

When dealing with MDPs, 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 [130]. Value iteration is an iterative technique for finding an optimal control policy and the corresponding time-average cost. With policy evaluation one can find the time-average cost of a given policy, and policy iteration improves and evaluates policies iteratively. 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.

Motivated by this, we introduce a novel method called *Value Function Discovery* that is aimed at obtaining an algebraic description of a relative value function. In essence, VFD fits an algebraic function through several sample points of the relative value function. The fitting procedure is based on a technique from the Evolutionary Algorithm (EA) family known as Genetic Programming (GP). By including sample points for various model parameters, VFD can also include these parameters when discovering a relative value function. After applying VFD, the relative value function can be used to, e.g., obtain an algebraic policy. In the current chapter we use this to demonstrate that, for an example MDP, the relative value function discovered by VFD yields a near-optimal policy.

In the remainder of this chapter we describe VFD and illustrate it by applying VFD to an example MDP. We start with a review of related work in Section 6.2, and an introduction to GP in Section 6.3. Then, we continue with a detailed description of VFD in Section 6.4 and of the example MDP in Section 6.5. Numerical results are presented in Section 6.6, followed by a discussion in Section 6.7 and concluding remarks in Section 6.8.

6.2 Related work

The literature combining EAs and MDPs mostly uses EAs to learn policies (contrary to VFD, which learns relative value functions). In [35] 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 [71] enhance the work in [35] by generating policies in the population via sub-MDPs, thereby speeding up convergence. From an application perspective, [161] provides an example of how EAs and MDPs can be used in a practical scenario. [24] 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 [93] by Lin et al., where the authors construct a piecewise linear approximation of the relative 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 relative value function. However, the relative value function discovered by VFD is a closed-form expression, whereas [93] finds a piecewise linear approximation. Having a closed-form expression is preferable when, e.g., studying the structure of the MDP using the discovered relative value function. Also, [93] focuses on convex relative value functions, and VFD does not make any assumptions about the structure of the relative value function. Another difference is the type of EA that is used: [93] employs a Genetic Algorithm, whereas VFD is based on GP. In particular, [93] does not use the tree-based representation inherent to GP. Finally, [93] does not allow for the placement of model parameters in the approximate relative value function.

A paper that does use GP in an MDP-context is [53]. The authors loosely explore the combination of GP and MDPs on an example of a war game and demonstrate that it performs well compared to a pure MDP-based technique. Their approach differs from the one described in this chapter, because they use GP to learn policies and not relative value functions, as VFD does.

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

- VFD applied to an optimal relative value function yields policies with near-optimal performance.
- For MDPs that allow for an explicit closed-form expression of the optimal relative value function, VFD can find this optimal relative value function with arbitrary precision. Thus, it can also find the optimal policy for such MDPs. We present an illustration of this in Section 6.6.6.

- VFD produces an algebraic expression of a policy that includes the parameters of the MDP. Hence, the 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.
- Relative value functions discovered by VFD can help gain an understanding of the structure of the optimal relative value function, policy, and model.
- Alternative techniques for analyzing MDPs often require knowledge of structural properties of the relative value function (e.g., gradient-based methods such as local search). These properties can be discovered by VFD.
- For many MDPs a near-optimal policy does not require an extremely accurate fit of the optimal relative value function. Thus, learning relative value functions can quickly result in good policies.
- VFD works with any MDP without requiring any changes to the algorithm.

6.3 Genetic programming

Since VFD is based on GP, we give a short description of this technique in this section. Readers interested in a more detailed treatment of GP are referred to books [44, 125, 142]. The general idea of GP is to maintain a population of individuals and iteratively attempt to improve this population over several generations. In each generation (i.e., an iteration step), the current population form new offspring by combining individuals. The main idea underlying GP is that combining good individuals leads, over time, to offspring that are better than their predecessors. Below we describe this procedure in more detail, with particular attention for the *mutation* and *recombination* operators used for generating offspring. Determining the quality of an individual is related to VFD's application of GP to MDPs, so we postpone it until Section 6.4.

In GP, each individual in the population is an algebraic expression represented by a tree, and later we use such trees to represent the relative value function of an MDP. Figure 6.1A illustrates a tree representation of the function $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$ (the relative value function of an $M/M/1$ queue [28]). 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 chapter we only use the operators $\{/, *, +, -\}$ from the example, but the representation is flexible and also allows for, e.g., exponents, square roots, and logarithms. Also, note that a representation of a function by a

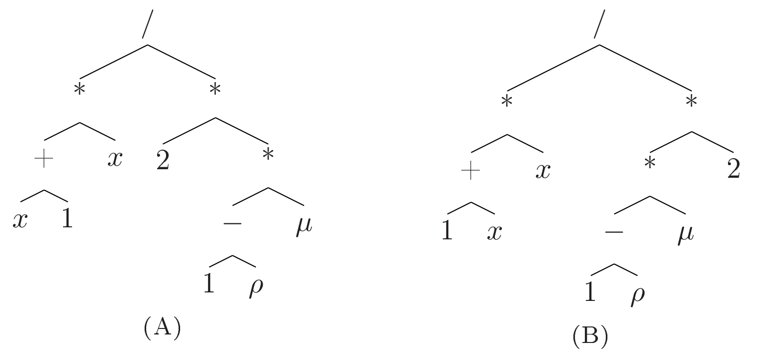


Figure 6.1: Two trees, each a representation of $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$.

tree is not unique: the tree in Figure 6.1B 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.

GP uses the mutation and recombination operator to generate new offspring from an existing population. In particular, the recombination operator generates two new offspring from two parents, and the mutation operator produces one new offspring from one parent. The recombination operator takes the following two steps:

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

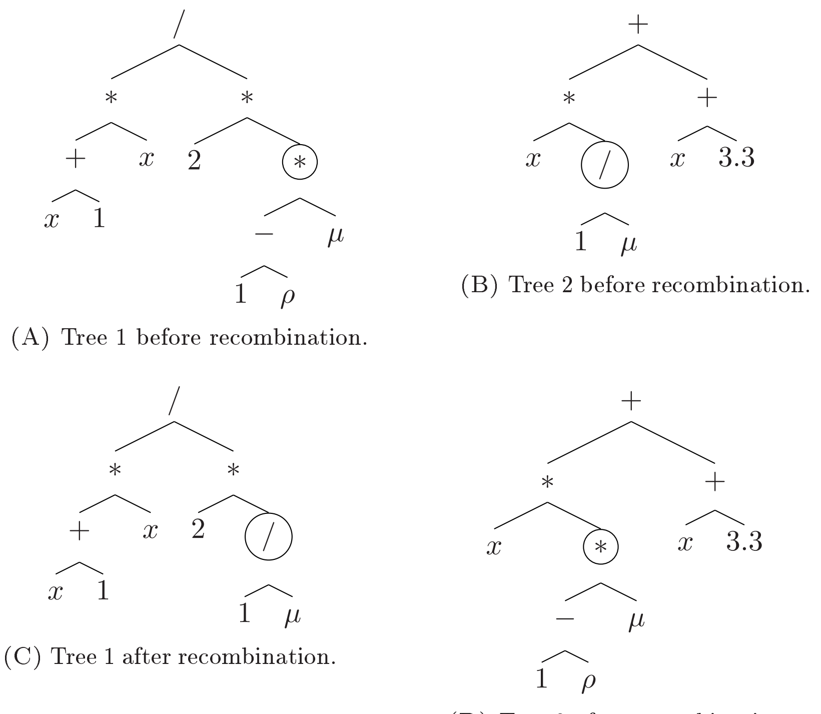
The procedure is illustrated in Figure 6.2, where recombination is applied to the two trees in Figures 6.2A and 6.2B. The subtree with the encircled $*$ as root in Figure 6.2A is exchanged with the subtree with root / (also encircled), resulting in the trees in Figures 6.2C and 6.2D. This combines the functions

$$V(x) = \frac{x(x+1)}{2\mu(1-\rho)} \quad \text{and} \quad V(x) = x\frac{1}{\mu} + x + 3.3,$$

to, respectively,

$$V(x) = \frac{x(x+1)}{2\frac{1}{\mu}} \quad \text{and} \quad V(x) = x(1-\rho)\mu + x + 3.3,$$

Mutation of trees is similar to recombination, except that a selected subtree is removed and replaced by a randomly generated subtree. The procedure is:



(A) Tree 1 before recombination.

(B) Tree 2 before recombination.

(C) Tree 1 after recombination.

(D) Tree 2 after recombination.

FIGURE 6.2: The recombination operator illustrated on the two trees in Figures 6.2A and 6.2B. The encircled subtrees are exchanged, resulting in the trees in Figures 6.2C and 6.2D.

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.

Figure 6.3 illustrates the procedure for the tree for $V(x) = \frac{x(x+1)}{2\mu(1-\rho)}$ which we saw earlier, displayed again in Figure 6.3A. 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 Figure 6.3B, with the newly added tree encircled. Thus, mutation changes $V(x)$ from $\frac{x(x+1)}{2\mu(1-\rho)}$ to $\frac{x(x+1)}{2x}$.

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

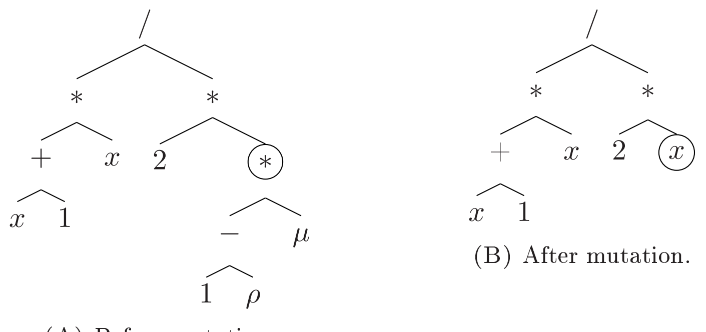


Figure 6.3: Mutation removes the subtree of the encircled node in Figure 6.3A (representing the term $\mu(1-\rho)$) and replaces it by a randomly generated subtree. The new subtree contains, in this case, only the element x and is encircled in Figure 6.3B.

start of the algorithm. The mutation operator 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.

6.4 Value Function Discovery

VFD applies GP to relative value functions of an MDP. Each individual in the population corresponds to a potential relative value function, and by repeatedly modifying these trees with the mutation and recombination operators, VFD discovers new relative value functions. In order to judge the quality of a tree in the population, VFD compares it to several sample points of the actual relative value function. If a tree 'closely' matches the sample points, it is considered a 'good' relative value function (this is made precise in Section 6.4.7). By including sample points for various MDP parameter values, VFD is able to include these parameters in the tree representation, and thus in the discovered relative value function.

In the remainder of this section we describe VFD in more detail, with particular attention for the preparation of the sample points, several GP-specific aspects that were omitted from Section 6.3, and determining the quality of the fit of a tree to the sample points.

6.4.1 Preparing sample points

Before VFD starts, it requires input from the MDP in the form of sample point sets. The sample points in each set are generated by fixing the values of the model parameters, finding the relative value function numerically (with, e.g., value iteration), and then selecting appropriate sample points. By repeating these steps for multiple values of the model parameters, they can be included in VFD. The steps are described below, along with some supporting notation:

1. Fix values for each of the m model parameters.
2. Find a numeric approximation of the relative value function of the MDP (by, e.g., value iteration).
3. Select several sample points that together capture the shape of the relative value function. Each sample point is denoted by s, and the pairs $(s, V(s))$ together form the sample point set $\mathcal{S}_q$.
4. Save these sample points into a file.
5. Repeat steps 1–4 for several combinations of the m parameters. We denote the resulting number of sample point sets by Q, and each sample point set by $\mathcal{S}_q$, with $q \in [0, Q-1]$.

In these steps, several choices are made based on the MDP that VFD is applied to: (a) the values for the m parameters in each sample point set in step 1, (b) the selection of the sample points in each set in step 3, and (c) the number of sample point sets Q in step 5. Section 6.5.2 illustrates the considerations for making these choices on an example MDP.

Preparing the sample point sets requires running, e.g., 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 model parameters that are not used to generate the sample points. This feature is illustrated later in this chapter in Section 6.6.4, when we apply VFD to an example MDP.

In this chapter we use value iteration for generating the sample point sets, but for applying VFD other techniques can be used as well. For instance, when the MDP is too large for running value iteration, one can also use TD-learning [148], which provides numerical approximations of the relative value function using simulations.

Algorithm 6.1 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 LAMBDA 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
```

6.4.2 Overview

A pseudo code listing of VFD is shown in Algorithm 6.1, and in the following paragraphs we describe the steps involved. We start with a high-level description in Algorithm 6.1, and then move on to a detailed description of the functions involved (Algorithms 6.2 and 6.3). During these descriptions we encounter the first of several parameters of VFD, which are listed in Table 6.1 (together with assigned values that we use later in the example MDP in Section 6.5). Functions and parameters are written in SMALLCAPS throughout the text, including trailing brackets () for functions.

The algorithm starts at 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 MU randomly generated trees. Lines 4–20 describe the steps taken by GP: first, LAMBDA 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 at 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).

6.4.3 Mutation, recombination, diversity, and convergence

Next we describe the functions used in Algorithm 6.1 in more detail, starting with the MUTATE() function at line 1 of Algorithm 6.2. Mutation occurs according to the GP paradigm, as described in Section 6.3: 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 is represented by 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.

Each 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 DIVERSITY_THRESHOLD, another parameter of VFD. If diversity drops below this 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

Algorithm 6.2 VFD continued

```
 1: function MUTATE(parent)
 2:     $z \leftarrow$ randint [0,numElements(parent)$-$1]
 3:     newSubtree $\leftarrow$ generateRandomTree()
 4:     parent$\rightarrow$setSubtree($z$, newSubtree)
 5: end function
 6:
 7: function RECOMBINE(parent1, parent2)
 8:     $z_1 \leftarrow$ randint [0,numElements(parent1)$-$1]
 9:     $z_2 \leftarrow$ randint [0,numElements(parent2)$-$1]
10:     subTree1 $\leftarrow$ parent1$\rightarrow$getSubtree($z_1$)
11:     subTree2 $\leftarrow$ parent2$\rightarrow$getSubtree($z_2$)
12:     parent1$\rightarrow$setSubtree($z_1$, subtree2)
13:     parent2$\rightarrow$setSubtree($z_2$, subtree1)
14: end function
15:
16: function INITPOPULATION( )
17:     population $\leftarrow$ List()
18:     for $k \leftarrow 0, \ldots,$MU$-1$ do
19:         population[$k$] $\leftarrow$ generateRandomTree()
20:     end for
21:     setError(population)
22:     sort(population)
23: end function
24:
25: function ISPOPULATIONDIVERSE(population)
26:     min $\leftarrow$ population[0]$\rightarrow$getError()
27:     max $\leftarrow$ population[MU$-$1]$\rightarrow$getError()
28:     div $\leftarrow$ (max-min)/min
29:     return div $>$ DIVERSITY_THRESHOLD
30: end function
31:
32: function ISCONVERGED( )
33:     return population[0]$\rightarrow$getError() $<$ MIN_ERROR
34: end function
```

with randomly generated trees (lines 18–20). The final steps at 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 6.7.

The final function in Algorithm 6.2 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.

6.4.4 Bloat in GP

When recombination exchanges, for instance, the root of the first tree with a leaf of the second tree, the second tree can increase in depth and in number of elements. Over time, this typically leads to large and deep trees, with negative effects on both speed and memory usage. This problem 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 6.2 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.

6.4.5 Parent selection and survivor selection

We continue with the SELECTPARENT() function in Algorithm 6.3, 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 separated 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 [44, 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. SELECTFROMGOODPROB is set to 0.8, again following conventions in the GP community. The selection is done at lines 4 and 6. Note that for recombination the SELECTPARENT() function is called twice.

Algorithm 6.3 VFD continued

```
function SELECTPARENT( )
    $z_1 \leftarrow$ floor(MU·GOODPCT)
    if select from good then
        $z_2 \leftarrow$ randint[0, $z_1 - 1$]
    else
        $z_2 \leftarrow$ randint[$z_1$, MU$-1$]
    end if
    return population[$z_2$]
end function

function SURVIVORSELECTION(population)
    remove population[MU:MU+LAMBDA$-1$]
end function

function SETERROR(trees)
    for tree in trees do
        maxError $\leftarrow 0$
        for $q \leftarrow 0, \ldots, Q-1$ do
            err $\leftarrow$ calcError(samplePointSets[$q$], tree)
            maxError $\leftarrow$ max (err, maxError)
        end for
        tree$\rightarrow$setError(maxError)
    end for
end function
```

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 individuals with the worst error from the population (line 12).

6.4.6 Creating random trees

Mutation and initialization of the population use function GENERATERANDOMTREE() for creating new trees. The type of operator in an internal node is determined randomly: it is a '+', '-', '*', or '/' with probability PROB_PLUS, PROB_MIN, PROB_MULTIPLY, and 1-PROB_PLUS-PROB_MIN-PROB_MULTIPLY, respectively. Similarly, leaf nodes are a variable, MDP parameter, or constant with probability PROB_VARIABLE, PROB_PARAMETER, and 1-PROB_VARIABLE-PROB_PARAMETER. We expect that the division operator '/' is needed less often than the others, and this is reflected in the values of VFD's parameters in the bottom part of Table 6.1.

6.4.7 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 relative 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 S_q is calculated via

$$\mathcal{E}_q = \max_{(s,V(s))\in\mathcal{S}_q} \frac{|\widetilde{V}(s) - V(s)|}{V(s)}. \tag{6.1}$$

Here, $\widetilde{V}(\cdot)$ is the function discovered by VFD and $V(\cdot)$ the optimal relative value function found by value iteration. The error $\mathcal{E}_q$ is calculated in the function CALCERROR() at line 19 in Algorithm 6.3. The error of a tree is then defined as

$$\mathcal{E} = \max_{q\in[0,Q-1]} \mathcal{E}_q, \tag{6.2}$$

i.e., the error of the tree is its worst error achieved on all the sample point sets. The error in Eq. (6.1) 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 relative 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 to MAPE.

Parameters Name	In example	Allowed values
Command line		
SEED	3,151,492	[0,MAXINT]
MU	1,000	[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_PARAMETER	0.45	[0,1]
PROB_VARIABLE	0.45	[0,1]

TABLE 6.1: The parameters available to VFD (first column), the values assigned to them for the example MDP in Section 6.5 (second column), and the values allowed by VFD (third column).

6.5 Example MDP

In the following paragraphs we illustrate VFD on an example MDP. We describe how VFD is configured and, in doing so, we have to choose the command line parameters of VFD, listed in the top part of Table 6.1. It shows the values we are going to choose in this section, as well as the range of values that is allowed. For completeness, it also has the parameters that were discussed in Section 6.4 and the values assigned to them by VFD. Before starting, we emphasize that we make reasonable choices for VFD's parameters and the sample point sets, rather than seeking choices that lead to, e.g., fast run times. Our focus is on demonstrating that VFD can indeed be used to learn a relative value function that yields a near-optimal policy.

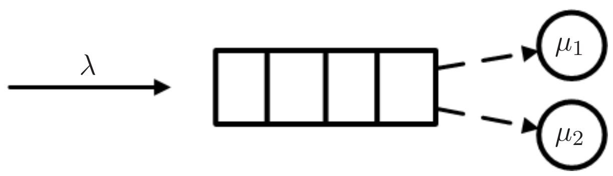


FIGURE 6.4: An M/M/2 system with control, where jobs (arriving with rate λ) from the queue have to be assigned to either a fast server S_1 (with service rate μ_1) or to a slow server S_2 (with service rate $\mu_2 < \mu_1$).

The MDP in this section is suitable for demonstrating VFD because:

- No known expression for the optimal policy or the relative value function exists, so we have no prior knowledge that VFD can capture the optimal relative 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.

6.5.1 Model formulation

Figure 6.4 shows a queue with Poisson arrivals (rate λ) and two servers with exponential service rates μ_1 and μ_2 (without loss of generality we take the service rates such that $\mu_1 > \mu_2$). Arriving jobs are put into the queue and, when they reach the head of the queue, have to be assigned non-preemptively to either the fast server (S_1) or the slower server (S_2). This decision is taken after a job completion, as well as when a new job arrives at the queue. We model this scenario as an MDP, with states $(x, i) \in \mathcal{X} = \mathbb{N} \times \{0, 1\}$. Here, x denotes the number of jobs in the queue and at S_1, and i the number of jobs at S_2. Our aim is to minimize the average number of jobs in the system. From [81] we have the optimality equation

$$g + V(x,i) = x + i + \lambda W(x+1,i) + \mu_1 W((x-1)^+, i) + \mu_2 W(x,0) \qquad (6.3)$$

with

$$\begin{aligned} W(x,0) &= \min\{V(x,0); V(x-1,1)\} \quad \text{if } x > 0, \\ W(0,i) &= V(0,i), \\ W(x,1) &= V(x,1). \end{aligned} \qquad (6.4)$$

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. (6.4). 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. (6.3), $x + i$ reflects the number of jobs in the system, $W(x + 1, i)$ the decision upon a job arrival, $W((x-1)^+, i)$ the decision when a job is completed at S_1, and $W(x, 0)$ the decision when a job is completed at S_2. Finally, the constant g is the time-average cost of the system.

Note that this formulation allows preemptive behavior, since the expression $W(1, 0) = \min\{V(1, 0); V(0, 1)\}$ 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 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. (6.3) and (6.4) we assume that the parameters are normalized such that $\lambda + \mu_1 + \mu_2 = 1$.

6.5.2 Generating sample point sets

The first step to running VFD is preparing the sample point sets. The steps were described in Section 6.4.1 and we repeat them here for convenience. In these steps, we include the fact that $m = 3$ for the example MDP, and that we use value iteration for generating the sample points.

1. Fix values for each of the three parameters λ, μ_1, and μ_2.
2. Run value iteration for the MDP.
3. Select sample points that capture the shape of the relative value function.
4. Save these sample points into a file.
5. Repeat steps 1–4 for Q pairs of the three parameters.

First we decide upon the number of sample point sets Q (for step 5) that we will generate, and on the MDP parameter values used for each set (for step 1). We make our choice for a worst-case scenario where S_2 is never used (i.e., an $M/M/1$ system) and choose parameters for the sample point sets based on the load $\rho = \lambda/\mu_1 \in [0, 1]$. Then, we generate parameters μ_1 and μ_2 uniformly from $[0, 1]$, set $\lambda_1 = \rho\mu_1$, at the same time ensuring that $\mu_1 > \mu_2$ and that $\lambda + \mu_1 + \mu_2 = 1$.

Set	ρ	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

TABLE 6.2: Model parameters per sample point set.

In the region $0 \leq \rho \leq 0.4$ the load on the system is low, and possible wrong decisions in a policy have little impact. Hence, we expect that an accurate relative value function in that region is not required, and we cover it by just two sample point sets: one at $\rho = 0.1$ and another at $\rho = 0.4$. Following similar reasoning, we choose two sample point sets 'close together' at $\rho = 0.9$ and $\rho = 0.95$ to cover scenarios with a high load. The region $0.4 < \rho < 0.9$ is then covered by $Q - 4$ sample point sets distributed evenly over the interval. Short experiments suggest that $Q = 7$ is a reasonable choice. The resulting ρ-values are $\{0.1, 0.4, 0.525, 0.65, 0.775, 0.9, 0.95\}$, and the model parameters of each set are listed in Table 6.2.

Note that, generally speaking, using many sample point sets (i.e., a large Q) ensures that VFD discovers a well-fitting relative 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 relative 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 relative value function, and the run time of VFD.

Now that the number of sets is chosen, the sample points in each set can be selected. 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 point set we then take 2×10 points, with the ten x-values evenly distributed over $[0, 0.75 \cdot L]$ and i both 0 and 1 (recall that $(x, i) \in \hat{\mathcal{X}}$ is a point in the state space). These sample points capture the shape of the relative 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 ten. Finally, we stop value iteration once the span of two consecutive iterations is less than 10^{-6}.

With these 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 relative 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 $\hat{\mathcal{X}}$, thus minimizing the effect of VFD's inability to extrapolate. In general, when applying VFD the user should keep in mind that it 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.

6.5.3 Determining command line parameters

The next step is determining the command line parameters, as listed in the top part of Table 6.1. The first, SEED, can be set to any desired integer value, as it is only used 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. In [44] populations with several thousands of individuals are suggested, but since our MDP has fairly low dimensionality we conservatively set MU$=$ 1,000 and LAMBDA$=$ 500.

For the parameter MAXELEMENTSINTREE we manually count the number of elements needed for the $M/M/1$ relative value function (13) and the $M/M/2$ relative value function ($\approx$ 90), based on the expressions in [28]. 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 relative value function depends on the decision $\min\{\widetilde{V}(x, 0); \widetilde{V}(x-1, 1)\}$. Even if $\widetilde{V}(x, i)$ is not highly accurate, the decision can still be correct. Hence, we choose MIN_ERROR quite large and set MIN_ERROR$=$ 0.20.

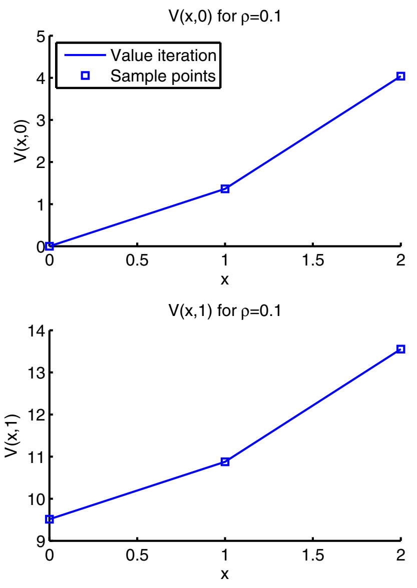
The value of DIVERSITY_THRESHOLD is determined by visually observing the progress made by VFD in terms of error in several short experimental runs. VFD should have sufficient time to discover good functions in between reinitialisations of the population, but should stop as soon as error stops decreasing 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).

Parameter APPLYMUTATIONPROB is used by VFD to decide between using the mutation or recombination operators. The GP literature (see [44, Sec. 6.4] and the references therein) suggests using a mutation probability in the order of 0.05. However, experiments on the current MDP indicate that setting APPLYMUTATIONPROB to 0.2 yields better results.

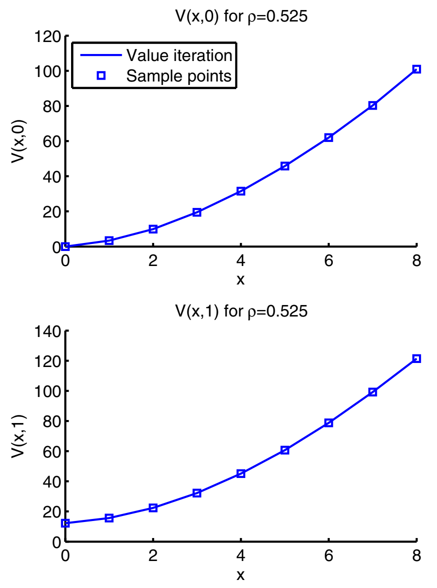
6.6 Numerical results

6.6.1 Sample points

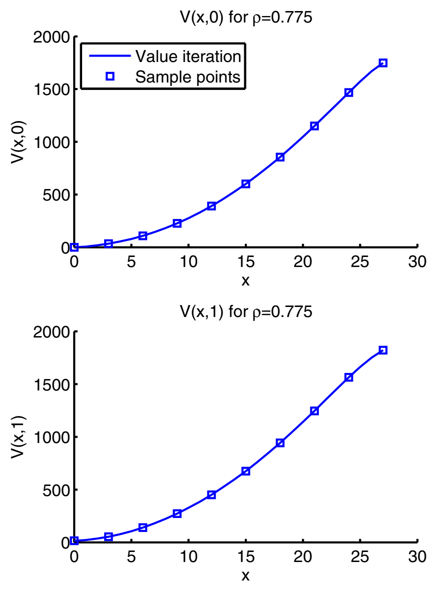
Section 6.5.2 describes how the sample points for our MDP example are generated. The values for the model parameters per sample point set are outlined in Table 6.2. For each of the model parameters in Table 6.2 we then run value iteration to find the sample points. Figure 6.5 shows the resulting sample points (marked by squares) for several of the sets. Note that the system with high load (Figure 6.5D) the optimal relative value function attains values in the order of 10^4, whereas for lower loads in Figures 6.5A and 6.5B these values are significantly smaller. Also, in Figure 6.5A the boundary $\lceil 0.75 \cdot L \rceil$ for value iteration is smaller than the number of desired sample points (ten), in which case only $\lceil 0.75 \cdot L \rceil$ sample points are retained. This results in three sample points for both $i = 0$ and $i = 1$, i.e., six sample points in total.



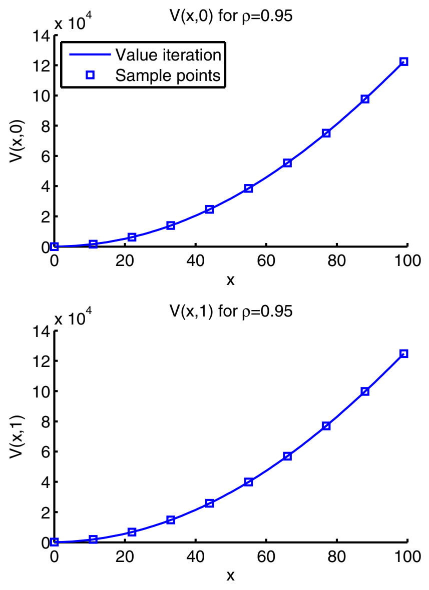
(A) Set 0.



(B) Set 2.

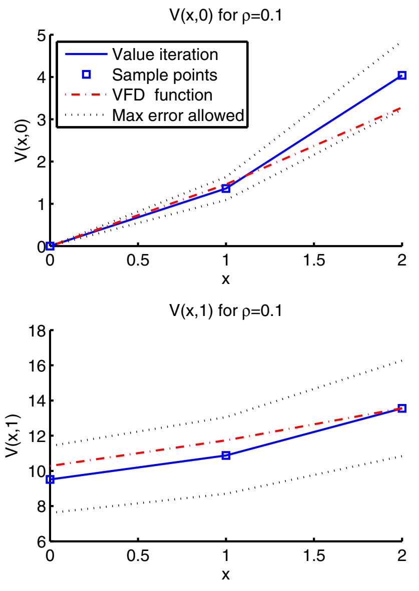


(C) Set 4.

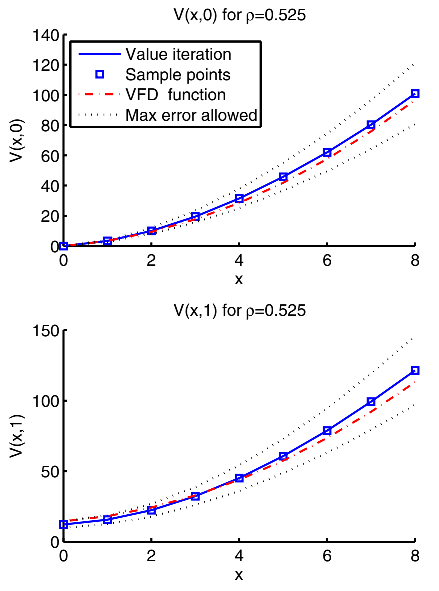


(D) Set 6.

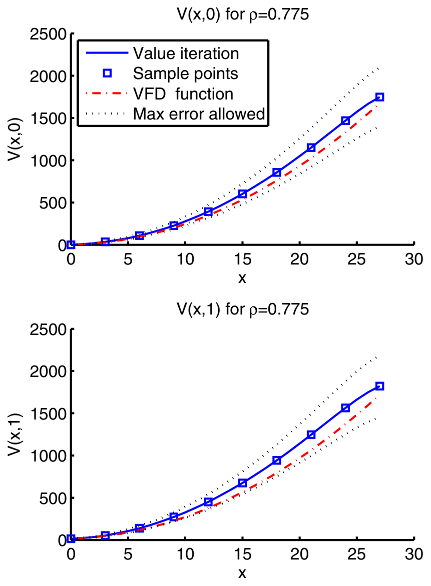
Figure 6.5: Sample point sets $0, 2, 4$, and 6.



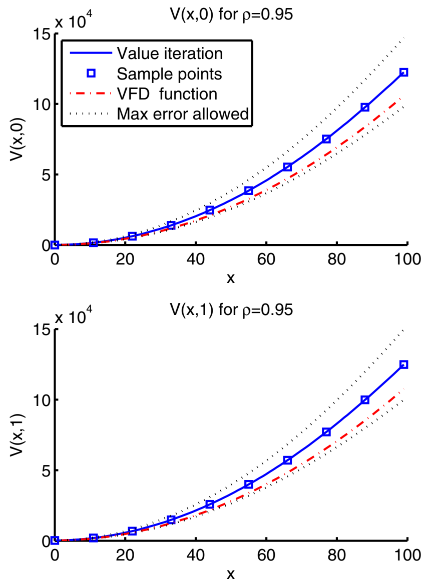
(A) Set 0.



(B) Set 2.



(C) Set 4.



(D) Set 6.

Figure 6.6: $\widetilde{V}(x,i)$ for sets $0, 2, 4$, and 6.

6.6.2 The discovered relative value function

Having specified all the input for VFD, it is ready to run. The relative value function $\widetilde{V}(x,i)$ discovered by VFD is

$$\begin{aligned}\widetilde{V}(x,i) = i/\Big[&0.28\mu_2(2\lambda\mu_2(i+\mu_1)(2\lambda+\mu_1)-i+\mu_2)\\ &\cdot\left((i+\lambda)\left(\frac{\lambda^2}{\mu_1}+\mu_2\right)+i-\mu_1\right)+\mu_2\Big]+x\\ &-\lambda\left(\lambda^2+1\right)x\Big[\lambda^2-\lambda\frac{\lambda^2\left(\frac{3.58i\lambda}{\mu_1}+3.58\lambda^2x+x\right)+\mu_2x}{\mu_2}\\ &-3.58(\lambda+\mu_1)-3.58\lambda x-\mu_1x-2\mu_2-x\Big].\end{aligned} \tag{6.5}$$

$\widetilde{V}(x,i)$ is plotted in Figure 6.6 (dash-dotted line) together with the sample points for the same sets as in Figure 6.5. 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. (6.1) and by the parameter MIN_ERROR. When running, VFD continues looking for a relative value function until one is found that lies completely between this upper and lower bound. Figure 6.6 demonstrates that $\widetilde{V}(x,i)$ resembles $V(x,i)$ well, and that it 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 6.4.7).

6.6.3 The policy derived from $\widetilde{V}(x,i)$

Next, we convert $\widetilde{V}(x,i)$ to a (algebraic) policy using a technique called *one-step policy improvement*, introduced in [118]. 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. (6.4) 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

Set	ρ	g	$\tilde{g}$	**Policy**
0	0.100	0.1107	0.1107	Use S_2 if $x > 25.5719$
1	0.400	0.6643	0.6643	Use S_2 if $x > 10.2648$
2	0.525	1.0589	1.0665	Use S_2 if $x > 5.9715$
3	0.650	1.7107	1.7368	Use S_2 if $x > 6.4751$
4	0.775	2.4684	2.5085	Use S_2 if $x > 4.6315$
5	0.900	7.3973	7.7279	Use S_2 if $x > 15.5992$
6	0.950	12.8241	13.5369	Use S_2 if $x > 17.4802$

TABLE 6.3: time-average cost $\tilde{g}$ for the policy based on the relative value function in Eq. (6.5) discovered by VFD. These costs are compared to costs g of the optimal policy. The policy in the last column indicates for which states $(x, 0)$ a job should be assigned to server S_2.

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 6.2, and list the policies in the last columns of Table 6.3. 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 [81] for a proof). The time-average cost $\tilde{g}$ of the discovered are in Table 6.3, and demonstrate that the policy yields good results for the various model parameter values.

6.6.4 VFD and interpolation

The time-average cost in Table 6.3 are based on the model parameters in Table 6.2, which were given to VFD as input. As mentioned in Section 6.5.2, we expect that VFD is able to interpolate well in the range $[0.100, 0.950]$ for ρ. To investigate this, we fix new values for ρ within that range (the second column in Table 6.4) 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. (6.5). The last two columns of Table 6.4 demonstrate 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 other values of the parameter SEED, and VFD continually yielded similar good results.

Set	ρ	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

Table 6.4: time-average cost $\tilde{g}$ for the policy based on the relative value function discovered by VFD, compared to costs g of the optimal policy. The model parameters (λ, μ_1, μ_2) and loads (ρ) are different from the ones VFD was given as input.

6.6.5 Computational complexity

With the model parameter values from Table 6.2 and the corresponding sample point sets VFD requires 2 minutes and 7 seconds to discover the $\widetilde{V}(x,i)$ from Eq. (6.5). 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 chapter the run time is quite short, which is mainly due to the small state space of the MDP in Eq. (6.3). 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 extremely challenging, time consuming, and does not always yield results. VFD, however, is easy to set up and run.

6.6.6 VFD applied to $M/M/1$

In Section 6.2 we claimed that for MDPs that allow for an explicit closed-form expression of the optimal relative value function, VFD can find this optimal relative value function. As an illustration, we let VFD discover the relative value function of an $M/M/1$ queue. To this end, 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 relative value function of an $M/M/1$ queue [28], and demonstrates that VFD is able to discover the closed-form expression that we expected.

6.7 Discussion

The results from the previous section demonstrate that VFD is able to discover relative value functions that closely resemble the optimal relative value function, and that the policy derived from a discovered relative value function perform wells. The good results in this chapter indicate that VFD is a promising technique and shows great potential. The research on VFD so far is, however, an initial step of exploring the idea of combining GP with MDPs. In particular, using VFD to gain valuable insights into the structure of an optimal relative value function is still unexplored. Another interesting application scenario of VFD is that of the control of an MDP with time-varying parameters. Having an algebraic policy prevents the need to make and analyze a time-dependent model. In the remainder of this section we list several potential directions for future research.

Shorter descriptions. In this chapter we showed the relative value function discovered by VFD in Eq. (6.5), but we did not analyze it further. It can, however, provide useful insights. For instance, $\widetilde{V}(x, i)$ in Eq. (6.5) 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 relative value function, because the expression in Eq. (6.5) is somewhat long. We even expect that it is acceptable to sacrifice some accuracy in return for shorter trees. An inclusion of this feature in VFD might help in discovering relative value functions that are simpler and easier to interpret.

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

Different error criterion for large MDPs. Prior to running VFD, it has to be supplied with sample point sets. For the MDP in this chapter, each set contains several points $((x,i), V(x,i))$ that together capture the shape of the optimal relative value function. The $V(x,i)$ are obtained by value iteration, which is computationally feasible for the small example MDP. For larger MDPs, however, this might not be possible. Earlier, we already mentioned the use of TD-learning as an alternative to value iteration. Another potential solution is to determine the error of a tree using the Bellman error of the optimality equation of the MDP, instead of with Eq. (6.1). For the MDP in this chapter, the modified error of a tree would be

$$\begin{aligned}\mathcal{E}_q = \sum_{(x,i)\in\mathcal{S}_q} \beta^{x+i}\Big| - \widetilde{V}(x,i) + x + i + \lambda\widetilde{W}(x+1,i) \\ + \mu_1\widetilde{W}((x-1)^+,i) + \mu_2\widetilde{W}(x,0)\Big|.\end{aligned} \tag{6.6}$$

This expression is based on the optimality equation in Eq. (6.3) and Eq. (6.4), with $V(x,i)$ replaced by $\widetilde{V}(x,i)$. Note that $\mathcal{E}_q \geq g$, because that is the error reached by the optimal relative value function. Also, the sample point sets $\mathcal{S}_q$ no longer contain $V(x,i)$, only a number of points (x,i) in the state space. Finally, to avoid overfitting to sample points farther away in the state space, the sum is weighted by a factor β^{x+i}, with β a suitable constant (in Eq. (6.1) overfitting was avoided by dividing by $V(x,i)$, but in this modified error criterion $V(x,i)$ is no longer available). We expect that with this modification, VFD is better able to handle large MDPs.

Additional operators. 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 deleting and inserting nodes.

Determining VFD's parameters. In Section 6.5 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 [143] to select good parameters, or to learn parameters on the go with, e.g., a co-evolutionary algorithm (see [47] for an example).

Improve diversity handling. 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 chapter, such a naive attitude towards diversity is sufficient to get a good relative value function quickly. However, for MDPs with larger state spaces, or MDPs that require a smaller error, this approach most likely does not yield a sufficiently good relative value function in a reasonable amount of time. Traditionally, GP algorithms employ a diversity maintenance scheme, e.g., a temporary increase of APPLYMUTATION-PROB 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 chapter 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 relative value functions and the resulting policies in the context of MDPs.

Learning policies. With certain MDPs it is also possible to use VFD to learn policies directly. In [81] the author proves that the optimal policy for the MDP in this chapter is a switching curve, i.e., there is 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. This experiment is the topic of the next chapter. Another example is the improved policy $\pi'(i, j, N)$ from Chapter 5, shown in Eq. (5.26). This policy includes a parameter $\hat{\alpha}$ that is determined with a numerical procedure. Instead of this procedure, VFD can be applied to sample points of $g'(\alpha)$ and thus help discover an expression for $g'(\alpha)$, which can then be minimized with respect to α. This yields an expression for the parameter $\hat{\alpha}$ in terms of the model parameters $\lambda_1, \lambda_2, \mu_1$, and μ_2. With this expression for $\hat{\alpha}$, the policy $\pi'(i, j, N)$ no longer needs a numerical procedure.

Consistency of discovered value functions. VFD is a stochastic process, and running VFD with different values for the parameter SEED should ideally result in the same discovered relative value function. This is particularly useful when analyzing the structure of the value function discovered by VFD in order to learn something about the true relative value function. We expect that such consistency is difficult to achieve, because VFD imposes no restrictions on the description of a discovered value function, and VFD is satisfied with any tree that has a sufficiently low error. Analyzing the relative value function of an MDP is, however, still possible with VFD, by inspecting the discovered trees from several runs of VFD for common structures.

6.8 Conclusion

In this chapter we introduced VFD, a novel method for discovering algebraic descriptions of relative 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 detailed description of VFD, discussed an example MDP, and applied VFD to that MDP to discover a relative value function. To illustrate how a discovered relative value function can be used, we obtained a policy from it via one-step policy improvement. Numerical experiments demonstrated 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.