SIFTER: Space-Efficient Value Iteration for Finite-Horizon MDPs

Konstantinos Skitsas
Aarhus University
201903630@post.au.dk

Ioannis G. Papageorgiou
NTU Athens
johnpapageorgiou0@gmail.com

Mohammad Sadegh Talebi
University of Copenhagen
m.shahi@di.ku.dk

Verena Kantere
NTU Athens
verena@mail.ntua.gr

Michael N. Katehakis
Rutgers University
mkatehakis@gmail.com

Panagiotis Karras
Aarhus University
piekarras@gmail.com

ABSTRACT

Can we solve finite-horizon Markov decision processes (FHMDPs) while raising low memory requirements? Such models find application in many cases where a decision-making agent needs to act in a probabilistic environment, from resource management to medicine to service provisioning. However, computing optimal policies such an agent should follow by dynamic programming value iteration raises either prohibitive space complexity, or, in reverse, non-scalable time complexity requirements. This scalability question has been largely neglected. In this paper, we propose SIFTER (Space Efficient Finite Horizon MDPs), a suite of algorithms that achieve a golden middle between space and time requirements. Our former algorithm raises space complexity growing with the square root of the horizon's length without a time-complexity overhead, while the latter's space requirements depend only logarithmically in horizon length with a corresponding logarithmic time complexity overhead. A thorough experimental study under diverse settings confirms that SIFTER algorithms achieve the predicted gains, while approximation techniques do not achieve the same combination of time efficiency, space efficiency, and result quality.

PVLDB Reference Format:
Konstantinos Skitsas, Ioannis G. Papageorgiou, Mohammad Sadegh Talebi, Verena Kantere, Michael N. Katehakis, and Panagiotis Karras. SIFTER: Space-Efficient Value Iteration for Finite-Horizon MDPs. PVLDB, 16(1): 90 - 98, 2022.
doi:10.14778/3561261.3561269

PVLDB Artifact Availability:
Source code, data, and other artifacts have been made available at https://github.com/constantinosskitsas/Space-Efficiency-in-Finite-Horizon-MDPs.

1 INTRODUCTION

Markov Decision Processes (MDPs) [25] model real-world problems where an agent learns [21] how to maximize a notion of reward in a probabilistic environment. An MDP is defined by states and available actions per state; each action yields a reward, which may be positive or negative, and induces a transition to a new state with some probability. Once MDP parameters are known, the agent may use them to compute an *optimal policy*.

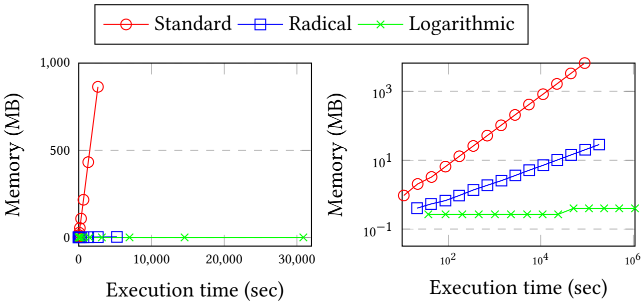

Figure 1: Memory vs. runtime; L: lin-lin plot; R: log-log plot.

Infinite-horizon MDPs model interactions that continue indefinitely or terminate at a *terminal state*; *finite*-horizon MDPs (FHMDPs) terminate once the agent executes a fixed number of steps. In both cases, the agent aims to maximize a reward its actions incur.

FHMDPs are used, e.g., to maximize battery life on mobile phones by deciding when to perform background actions [8], to schedule how patients use tomography scanner in a hospital [14], to plan medical treatments [2], to decide which path to follow in car-sharing services [7], and to plan airline meal provisioning [15].

Bellman's algorithm [3] finds the *exact* solution to an FHMDP; it has served as the standard since 1957; still, its *space complexity* grows linearly in *both* the number of states and horizon length; astoundingly, it remains unchanged since 1957 [25]. Still, several FHMDP use cases [2, 7, 8, 14, 15, 29–31] raise high memory requirements or run on constrained-memory devices [7, 8]. In these circumstances, the standard algorithm cannot run within available resources; it thus makes sense to trade off a *drastic* reduction in space complexity for a *minimal* runtime overhead.

In this paper we introduce SIFTER: a suite of algorithms that reduce the space-complexity of FHMDP value iteration [3] from linear to *radical* or *logarithmic* in horizon length, via limited recomputation. Despite the easing of memory needs, one algorithm incurs no, and the other only a logarithmic, time-complexity overhead. Table 1 reviews the time and space complexity of state-of-the-art exact FHMDP methods and our proposals. Figure 1 shows how our schemes treat the space-time complexity tradeoff compared to the standard method, summing up results reported in Section 5.

Table 1: Complexity of *exact* FHMDP methods.

method	**Time**	**Space**
Standard	$O(\lvert S\rvert^2\lvert A\rvert N)$	$O(\lvert S\rvert N)$
InPlace	$O(\lvert S\rvert^2\lvert A\rvert N^2)$	$O(\lvert S\rvert)$
Radical (ours)	$O(\lvert S\rvert^2\lvert A\rvert N)$	$O(\lvert S\rvert\sqrt{N})$
Logarithmic (ours)	$O(\lvert S\rvert^2\lvert A\rvert N\log N)$	$O(\lvert S\rvert\log N)$

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:10.14778/3561261.3561269

2 BACKGROUND AND RELATED WORK

A Markov Decision Process (MDP) models a problem where a decision-making agent interacts with a probabilistic environment. Each decision of the agent triggers a response from the environment. The agent aims to make optimal decisions with regard to some performance metric. Decisions take place over time, either continuously, whereby the agent makes a decision when some event occurs, or in discrete *epochs*. We focus on the discrete case. The number of decision epochs, or *horizon* N, could be finite or infinite.

A *stationary* Finite-Horizon MDP (FHMDP) M is specified by a tuple $M = (S, A, p, R, N)$ [25]; S is a finite *state space*, $A = \cup_{s \in S} A(s)$ a finite *action space* such that $A(s)$ is the set of actions available in state $s \in S$, P a *Markovian* (memory-less) *transition function* such that for each $s \in S$ and $a \in A(s)$, $p(s'|s, a)$ is the *probability* of transiting to s' when the agent chooses action a at state s; thus, $p(\cdot|s, a)$ is a *probability distribution* over S, $\sum_{s' \in S} p(s'|s, a) = 1$; R is a *reward function* such that executing action $a \in A(s)$ in state s yields a (possibly random) reward r drawn from $R(s, a)$, i.e., $r \sim R(s, a)$; lastly, N denotes the finite horizon, assumed to be fixed.

The agent interacts with the environment M in N rounds (or epochs). In each epoch $t \in [N] := \{1, \ldots, N\}$, the agent occupies a *state* $s_t \in S$, fully observable to the agent, and chooses an *action* $a_t \in A(s_t)$ by some *policy* π, to receive a *reward* $r_t \sim R(s_t, a_t)$ and transits to a state $s_{t+1} \sim p(\cdot|s_t, a_t)$. The goal of the agent is to maximize the expected total reward collected during the interaction, i.e., to maximize $\mathbb{E}[\sum_{t=1}^{N} r_t]$, where the expectation is taken with respect to the possibly random choice of the initial state, randomness in the rewards and state transitions, and the possible randomization in the action selection policy π. By a slight abuse of notation, we use R to denote both a reward distribution and its mean.

2.1 Exact FHMDP Solution

Actions in MDPs are chosen according to *policies*. A randomized (or stochastic) FHMDP policy is a mapping $\pi : S \times [N] \rightarrow \Delta(A)$, where $\Delta(A)$ is the set of all probability distributions over A. For each state $s \in S$ and *remaining* steps $k \in [N]$, $\pi(s, k)$ defines a probability distribution over $A(s)$, according to which the agent samples an action $a \sim \pi(s, k)$. A *deterministic* policy π prescribes one action $a = \pi(s, k)$ for each $s \in S$ and $k \in [N]$. The *value function* of a policy π is a function $V^\pi : S \times [N] \rightarrow \mathbb{R}$ such that $V^\pi(s, k)$ corresponds to the *expected* sum of $k - 1$ rewards received under policy π when starting from $s_k = s$. Formally, for $s \in S$ and $k \in [N]$,

$$V^\pi(s, k) := \mathbb{E}\left[\sum_{t=1}^{k-1} r_t \middle| s_k = s \right],$$

where the expectation is taken with respect to the randomness in the transitions, rewards, and possible randomization in π. For brevity, we define $V_k^\pi(s) := V^\pi(s, k)$.

Solving an FHMDP calls for finding a policy of highest value, i.e., finding $\max_\pi V_N^\pi(s)$ for each initial state $s \in S$. A fundamental result in FHMDP theory states that there exists a deterministic policy $\pi^\star$ that is optimal for all initial states [25]. More precisely, there exists $V^{\pi^\star} : S \times [N] \rightarrow \mathbb{R}$ such that $V_N^{\pi^\star}(s) = V_N^\star(s) = \max_\pi V_N^\pi(s)$ for all s. Furthermore, $V^{\pi^\star}$ satisfies the *Bellman optimality equation*: starting from $V_0^\star = 0$, we have for all $s \in S$, $k \in [N]$,

$$V_k^\star(s) = \max_{a \in A(s)} \Big(R(s, a) + \sum_{s' \in S} p(s'|s, a) V_{k-1}^\star(s') \Big). \quad (1)$$

Equation (1) computes $V^\star$ by *dynamic programming* performing *backward induction* [4], which resembles *value iteration* for infinite-horizon MDPs; we use the term *value iteration* as an all-encompassing term for such computations.

2.2 Turnpike Integers

A Finite-Horizon MDP problem becomes too memory-demanding as the length of the horizon N and the state space S grow, as it needs $O(N|S|)$ space. On the other hand, an Infinite-Horizon MDP is solved in $O(|S|)$ space by value iteration on one in-place array. Past works [20, 25, 26] proposed approximating finite-horizon solutions via those for an infinite horizon. For an FHMDP and a discount factor γ there exists a *turnpike integer* $N^*(\gamma)$, such that decisions made in epochs $t \geq N^*(\gamma)$ are the same as in the infinite-horizon case. Thus, the agent may follow a time-independent optimal policy for a γ-discounted infinite-horizon MDP for the first $N^* - 1$ epochs of N to $N - N^* + 1$ remaining steps, coupled with a solution for an $(N - N^*)$-horizon FHMDP for as many final steps. Yet finding a turnpike integer is challenging, even while there are bounds therefor [20, 25]. A *heuristic* is to solve an FHMDP for the whole horizon, but only store and use the optimal policy for the full length N.

2.3 Non-stationary MDPs

A *Non-Stationary MDP* (NSMDP) [19] is one whose transition and reward functions depend on the decision epoch. Formally, it is specified as $M = (S, A, \{p_t\}_{t \in [N]}, \{R_t\}_{t \in [N]}, N)$, where S, A, and N are as in the case of FHMDP, and for each $t \in [N] = \{1, \ldots, N\}$, p_t and R_t denote the transition and reward function at step t, respectively. Executing action a at state s in round t yields a reward $r \sim R_t(s, a)$ and a next-state sampled from $p_t(\cdot|s, a)$. NSMDPs find applications in real-world problems, as described in Section 1. Assuming the agent knows how functions change with time, FHMDP solutions work in the non-stationary case.

2.4 Space Efficiency

Memory is a critical resource for a FHMDP. Some works reduce MDP space requirements by aggregating states into bins [15]. *External memory value iteration* [12] utilizes external memory at the cost of execution time; its *partitioned* variant [11] partitions the state space into blocks and loads those to main memory, gaining in I/O efficiency. However, these solutions are case-dependent heuristics pertaining to the model's memory footprint; they do not reduce the algorithm's space complexity requirements. By contrast, we aim to reduce the space complexity of the *dynamic-programming value iteration* algorithm for FHMDPs.

3 BASIC FHMDP SOLUTIONS

Unfortunately, existing algorithms that apply the Bellman equations to solve FHMDPs are either time-efficient but space-demanding, or space-efficient but time-demanding. Here, we present these solutions; in the next section, we propose algorithms that achieve both time- and space-efficiency, inspired by cost amortization techniques used in synopsis construction [18] and graph processing [23].

3.1 Standard Solution

The *standard* FHMDP solution [3] computes and stores in memory every policy for every decision epoch, i.e., number of remaining finite-horizon steps. When the agent commences decisions, it recalls the policy for the current number of available steps and uses it to make its choice. The process is repeated until completing a number of steps equal to the horizon.

Algorithm 1 Standard solution

```
procedure STANDARDSOLUTION(horizon)
    $V_{tmp} \leftarrow []$
    for $i = 0$ to $horizon$ do
        $policyArray \leftarrow []$
        $V_{aux} \leftarrow []$
        for $s \in S$ do
            for $a \in A(s)$ do
                $Q(s,a) \leftarrow 0$
                for $s' \in S(s)$ do
                    $Q(s,a) \leftarrow Q(s,a) + R(s,a) + p(s'|s,a)V_{tmp}(s')$
            $V_{aux}(s) \leftarrow \max_a Q(s,a)$
            $policyArray[s] = \text{argmax}_a Q(s,a)$
        $V_{tmp} \leftarrow V_{aux}$
        $policyStack.push(policyArray)$
    for $i = 0$ to $horizon$ do
        $chosenAction = policyStack.top()[currentState]$
        $takeAction(chosenAction)$
        $policyStack.pop()$
```

Algorithm 1 shows the pseudocode. We store policies, each represented as an array (*policyArray*) of length equal to the number of states $|S|$, in a stack (*policyStack*). At any time, the top of the stack returns the policy for the remaining number of steps. The application-dependent *takeAction* function takes a chosen action as argument and executes it, rewarding the agent and transferring them to the next state in a probabilistic way. Time complexity is dominated by the four nested loops iterating over the horizon N, states S, actions $A(s)$ available at the current state s, and neighboring states accessible at s, $S(s)$, respectively. Thus, time complexity is $O(N|S|^2|A|)$. Throughout its execution, the algorithm stores N policy arrays of length $|S|$. Thus, its space complexity is $O(N|S|)$, growing linearly in the horizon's length for a fixed state space.

3.2 In-place Solution

Another way to solve a fully known FHMDP is the *in-place solution*, whereby the agent repetitively recalculates the policy array from scratch for each remaining number of steps, without storing it. Space complexity is $O(|S|)$, independent of the horizon, as one policy array and at most two value function arrays (V_{tmp}, V_{aux}) of size $|S|$ are in memory at any time. Nevertheless, the space complexity advantage is counterbalanced in execution time. The algorithm reiterates over four loops as in Algorithm 1 in each step; each iteration needs $O(j|S|^2|A|)$ time, where j is the number of steps remaining, hence the total time complexity is $O(N^2|S|^2|A|)$.

4 SIFTER: ADVANCED FHMDP SOLUTIONS

The two basic solutions fail to resolve the tradeoff between time and space efficiency. In this section, we introduce FHDMP solutions that strike a balance between these two desiderata, achieving either the same or slightly higher space complexity as the in-place solution along with the same or slightly increased time complexity as that of the state-of-the-art *standard* solution [3]. We emphasize that these solutions retain all correctness and convergence properties of the original dynamic programming solution, as they perform the same computations [17]; they recompute some parts, yet these re-computations produce the same results as the original, as all steps of the dynamic programming algorithm are deterministic.

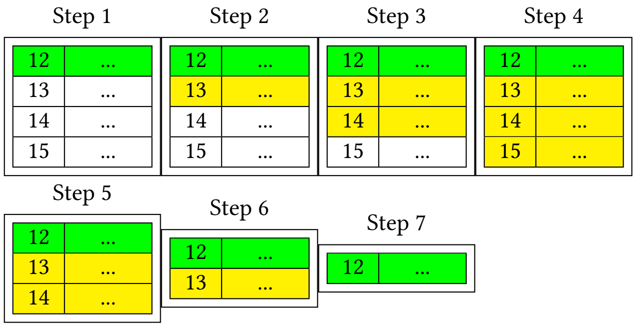

Figure 2: Computations at indices 15 . . . 12; array 12 is stored in advance; arrays 13, 14, 15 are stored temporarily and used.

4.1 Radical Solution

Our first solution improves on the state-of-the-art *standard* [3] solution in terms of space complexity, while maintaining the same time complexity. Instead of storing every policy and value function array in memory, we store a few *checkpoint* arrays, recompute intermediate ones when needed, and maintain arrays in the *active* interval between checkpoints. We first calculate every array up to the horizon's end, storing those with indices that are multiples of the horizon's square root $\sqrt{N}$, hence $O(\sqrt{N})$ arrays. Thereafter, whenever the agent reaches a checkpoint, we compute and store the $\sqrt{N}$ arrays in the length-$\sqrt{N}$ interval between that checkpoint and the next. We use each of the inter-checkpoint arrays to obtain optimal actions, and discard it. As we calculate each array at most twice, we need $O(\sqrt{N}|S|^2|A|)$ time per intervals; as there are $\sqrt{N}$ intervals, the total time complexity is $O(N|S|^2|A|)$. As no more than $2\sqrt{N}$ arrays of length $|S|$ are stored in memory at any time, space complexity is $O(\sqrt{N}|S|)$.

Figure 2 visualizes a snapshot of how arrays are stored for a horizon of length 16. Starting from index 12, we compute each of the arrays from index 13 to 15, and keep them in memory. The agent uses the value function in each computed array, from index 15 downwards, to make decisions and execute actions and thereafter discards it. We repeat this process at each interval of length $\sqrt{N}$.

Algorithm 2 Radical solution

```
procedure ROOTSOLUTION(N)
    $V \leftarrow zeros(N); steps = N$
    for $i = 0$ to $N$ step $\lfloor\sqrt{N}\rfloor$ do
        $calculateValues(i + \lfloor\sqrt{N}\rfloor, i, V, false)$
    if $indexStack.top() < N$ then
        $calculateValues(N, indexStack.top(), valueStack.top(), true)$
    while $steps > 0$ do
        if $indexStack.empty()$ then
            $calculateValues(steps, 0, zeros(N), false)$
        else if $((steps + 1) mod \lfloor\sqrt{N}\rfloor) == 0$ then
            $calculateValues(steps, indexStack.top(), valueStack.top(), true)$
        $V \leftarrow valueStack.top()$
        $chosenAction = calculateBestAction(V[currentState])$
        $takeAction(chosenAction)$
        $steps = steps - 1$
```

Algorithm 2 presents the pseudocode. The *calculateValues* function, in Algorithm 3, computes every value function array using the index of the array to be computed, *targetIndex*, the largest index of a stored array *startingIndex*. The fourth argument (*tree*) is boolean; when true, it causes the function to save every intermediate value function in a stack *valueStack*; when false, the function only stores the final value function array. When a value function array is stored in *valueStack*, its corresponding index is stored in *indexStack*. In Algorithm 2, the first loop computes and stores the value function arrays whose index is a multiple of $\lfloor\sqrt{N}\rfloor$. Variable *steps* indicates the number of remaining steps; when that number is one less than a multiple of $\lfloor\sqrt{N}\rfloor$, the situation is as that in Figure 2, hence we calculate and store intermediate value functions. Otherwise, the required array is already in the stack. In each step, we get the optimal policy from an array to decide on an action and proceed.

Algorithm 3 Auxiliary function calculating value functions

```
procedure CALCULATEVALUES(targetIndex, startingIndex, V, tree)
    $V_{tmp} \leftarrow V; V_{aux} \leftarrow []$
    for $i = startingIndex + 1$ to $targetIndex + 1$ do
        for $s \in S$ do
            for $a \in A(s)$ do
                $Q(s,a) \leftarrow 0$
                for $s' \in S(s)$ do
                    $Q(s,a) \leftarrow Q(s,a) + R(s,a) + p(s'|s,a)V_{tmp}(s')$
            $V_{aux}(s) \leftarrow \max_a Q(s,a)$
        $V_{tmp} \leftarrow V_{aux}$
        $V_{aux} \leftarrow []$
        if tree then
            $valueStack.push(V_{tmp})$
            $indexStack.push(i)$
    $valueStack.push(V_{tmp})$
    $indexStack.push(i)$
```

4.2 Logarithmic Solution

Our second solution further lowers the memory needs to a logarithmic factor of the horizon's length. To illustrate this solution, we visualize value function arrays as stored in a full *binary search tree* [9], as in Figure 3. Tree nodes are ordered by keys, which correspond to the value function array's index k, i.e., the number of remaining epochs in the horizon. The root stands for the epoch in the middle of the horizon, its left child to the epoch in the middle of the first half of the horizon, its right child to the epoch in the middle of the second half, and so on, recursively. Throughout the algorithm's execution, as we compute an array V, we store *only* those *preceding* arrays that lie *along* the path from the tree root to the node of V; in the binary tree, those are ancestors associated with a *lower value of* k, hence lie on the leftward side of the node; in Figure 3, for example, the stored ancestors of the node corresponding to $k = 11$ are those corresponding to $k = 10$ and $k = 8$. After using V, we backtrack to the nearest stored ancestor and use it to derive the next action. As a path from the root to a node has length at most $\log N$, at most $O(\log N)$ arrays are stored at any time.

Figure 3 illustrates a snapshot of the execution of this *logarithmic* algorithm with horizon of length 15. A green node indicates that the corresponding value function array is stored in memory, while a yellow node indicates that the array is currently used to take an action for the remaining number of steps. We always use the stored array of highest index. Initially, we compute value function arrays up to index 15, and store, in *logarithmic intervals*, those of index 8, 12, and 14. Thereafter, the agent sequentially uses the stored arrays of index 15 and 14 to get the optimal policy for those numbers of remaining steps. With 13 steps remaining, we start from the stored array of index 12, compute that of index 13 and take the respective actions. Similarly, we then start from the array of index 8, compute up to that of index 11, while storing in memory those of index 10 and 11 which lie along the root-to-leaf path from 8 to 11 in the tree; we use the arrays of index 11 and 10 to take actions for the respective numbers of remaining steps. Next, we recompute the array of index 9 using that of index 8, and use both of these to take actions with such numbers of remaining steps. We proceed in the same manner, storing in logarithmic intervals arrays of index 4, 6, and 7, using arrays at index 7 and 6, recompute and use that of index 5 and then peruse the one of index 4. Lastly, we recompute and store the arrays of index 2 and 3, to use them, discard them, and recompute the array of index 1 to get the final action.

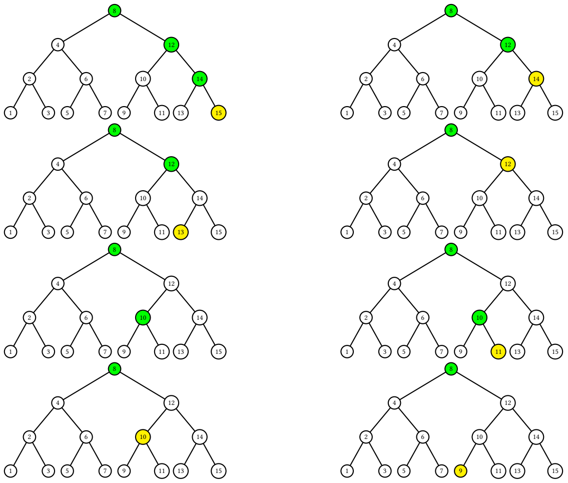

Figure 3: Logarithmic solution progress sequence snapshot; green nodes: in memory; yellow nodes: currently in use.

This algorithm stores at most $\lfloor\log N\rfloor$ arrays of length $|S|$ at any time, yielding a space complexity of $O(|S|\log N)$. Algorithm 4 shows the pseudocode. It uses the *calculateValues* function discussed in Section 4.1, as well as a new auxiliary function (Algorithm 5), which finds the indices of arrays to be stored at each step, using variables l and r as indices for the left and right ends.

Algorithm 4 Logarithmic solution

```
procedure TREESOLUTION(N)
    $steps \leftarrow N$
    $V \leftarrow []$
    while $steps > 0$ do
        $V \leftarrow treeTraversal(steps, N)$
        $chosenAction = calculateBestAction(V[currentState])$
        $takeAction(chosenAction)$
        $steps = steps - 1$
```

Variable k moves as the *key* in binary search. When an array is already in the stack, we continue the search from this array's index (Line 13); if the top of the stack is the target value, the search is over (Lines 8–11). Otherwise, if l is no greater than r, we proceed in one of three ways: If k is the *target*, we compute and return the value function we need, starting from the last stored array. If k is *smaller* than the target, we calculate and store the array corresponding to k, as it lies on the path from the root to the target and on the left of the target, update l to the successor of k and k to $\frac{l+r}{2}$; otherwise, if k is *larger* than the target, the array of k needs no storing; we update r to the predecessor of k and k to $\frac{l+r}{2}$. The process reiterates while $l \leq r$, and returns array V_{tmp} containing the target value function. As Algorithm 4 shows, in each step, we compute the requested value function and use it to get the optimal action.

By the arrangement of epochs (i.e., steps in the horizon) in tree levels, tree level ℓ, counting from the top of the tree as 1, holds $2^{\ell-1}$ nodes. Each node at level ℓ stands for an epoch $\frac{N}{2^{\ell}}$ steps apart from its nearest *stored* (i.e., leftward) ancestor or from the horizon's start, hence requires as many steps of re-computation starting therefrom. In effect, the re-computation steps are $\sum_{\ell=1}^{\log N} 2^{\ell-1}\frac{N}{2^{\ell}} = O(N \log N)$, hence time complexity becomes $O(|S|^2|A|N \log N)$.

Algorithm 5 Auxiliary function to store value function arrays

```
function TREETRAVERSAL($target$, N)
    $l \leftarrow 0; r \leftarrow N; k \leftarrow \frac{l+r}{2}; V_{tmp} \leftarrow []$
    if $notindexStack.empty()$ then
        if indexStack.top() == $target$ then
            $V_{tmp} \leftarrow valueStack.top()$
            $valueStack.pop()$
            $indexStack.pop()$
            return $V_{tmp}$
        else
            $k \leftarrow indexStack.top()$
    while $l \leq r$ do
        if $k == target$ then
            if $indexStack.empty()$ then
                $calculateValues(k, 0, zeros(N), false)$
            else
                $calculateValues(k, indexStack.top(), valueStack.top(), false)$
            $valueStack.pop()$
            $indexStack.pop()$
            $V_{tmp} \leftarrow valueStack.top()$
            break
        else if $k < target$ then
            if $indexStack.empty()$ then
                $calculateValues(k, 0, zeros(N), false)$
            else if $indexStack.top()! = k$ then
                $calculateValues(k, indexStack.top(), valueStack.top(), false)$
            $l = k + 1; k = \frac{l+r}{2}$
        else
            $r = k - 1; k = \frac{l+r}{2}$
    return $V_{tmp}$
```

4.3 Common Framework

While we presented the two SIFTER solutions as distinct, here we sketch a framework that encompasses both. Consider a hierarchy of depth $\log_L N$ that splits the horizon in L intervals of size N/L per level. We traverse this hierarchy in an in-order fashion, keeping in memory at all times the value arrays that correspond to ancestors of the current array and up to $L - 1$ of their interval peers; upon request, we recompute an array not in memory from its nearest predecessor that is in memory. Notably, for $L = \sqrt{N}$ we obtain a hierarchy of depth $\log_{\sqrt{N}} N = 2$, i.e., the radical solution, and for $L = 2$ a hierarchy of depth $\log_2 N$, i.e., the logarithmic solution. Thus, our solutions correspond to the two extreme values of L. One may apply this generic SIFTER solution, choosing the value of L that best resolves the space-time tradeoff for their needs. The generic case has time complexity $O(|S|^2|A|N \log_L N)$ and space complexity $O(|S|NL \log_L N)$.

5 EXPERIMENTAL STUDY

We conducted an experimental study to validate our theoretical results. We measure collected reward, execution time, and memory used. Our code, in C++ 11, is available[1]. Experiments ran on a 378GB Linux server with Intel(R) Xeon(R) E5-2687W v3 @ 3.10GHz.

5.1 Use Case and Data Description

As a use case, we chose the problem studied in [22], which regards an elastic computer cluster housing a distributed database. This cluster constantly receives read requests, while the number of Virtual Machines (VMs) it comprises can be changed depending on system needs. The actions the agent can make are adding a VM, removing a VM, or doing nothing. The number of states in the model varies depending on the maximum and minimum allowed number of VMs and the incoming load. The agent acts as a cluster coordinator, adjusting the number of active VMs as needed to serve the incoming load efficiently. Model states are expressed by a set of parameters, such as the total load and the number of VMs. Using that information, the agent learns the system's parameters and makes the optimal decision every time it is required. In [22], the authors compared four algorithms to solve the resulting MDP, including classic infinite-horizon value iteration, Q-Learning, and versions thereof using Decision Trees. While the model was treated as an Infinite-Horizon MDP, it ran for a finite number of steps in practice, while assuming that the agent did not know the horizon's length in advance. We apply our solutions on top of the classic MDP, while properly treating the problem as a Finite-Horizon MDP, with the agent knowing the number of steps they should make. Moreover, we treat the model as fully known, letting the agent learn the system's parameters during a training phase, and using them in evaluation period. Following [22], we assume a cluster size that varies from 1 to 20 VMs, actions of increasing the cluster size by 1, decrease size by 1, and no operation, available to the agent only when they can be performed. The incoming load before training is a sinusoidal function, $load(t) = 50 + 50\sin(\frac{2\pi t}{250})$; in the evaluation period its frequency doubles. The percentage of the incoming load that is read requests is given by a different sinusoidal function, $r(t) = 0.75 + 0.25\sin(\frac{2\pi t}{340})$. The RAM size is 1024 for the first 220 steps, then 2048 for the next 220. This pattern continues for any number of steps. The I/O operations per second are also given by a sinusoidal function, $io(t) = 0.6 + 0.4\sin(\frac{2\pi t}{195})$. The capacity of the cluster at any time t is $capacity(t) = (10r(t) - io\_penalty - ram\_penalty)vms(t)$, where $vms(t)$ is the number of VMs in the cluster at time t; the parameter $io\_penalty$ is 0 when $io < 0.7$, $10io(t) - 0.7$ when $0.7 \leq io(t) \leq 0.9$, and 2 otherwise; $ram\_penalty = 0.3$ when the RAM size is 1024 and 0 otherwise. The reward for an action is:

$$reward(t) = \min(capacity(t+1), load(t+1)) - 2vms(t+1)$$

Thus, the agent is rewarded when the capacity of the system suffices to serve the load and penalized if it over- or under-delivers.

5.2 Compared Methods

We compare existing and proposed FHMDP methods, including two approximations based on the Turnpike theorem (Section 2.2).

[1] https://github.com/constantinosskitsas/Space-Efficiency-in-Finite-Horizon-MDPs

Infinite-horizon-based approximation. This approach approximates an FHMDP solution using an *infinite-horizon* MDP that executes a single in-place value iteration with a discount factor γ until convergence, and uses the resulting values to make every decision for a finite number of steps. Value iteration for discounted MDPs computes a sequence $\left(V^{(n)}\right)_{n\geq 0}$, with $V^{(0)}$ chosen arbitrarily (e.g., $V^{(0)} = 0$), and for all $n \geq 0$,

$$V^{(n+1)}(s) = \max_{a\in A(s)} \left(R(s,a) + \gamma \sum_{s'} p(s'|s,a)V^{(n)}(s')\right), \quad s \in S.$$

If $\gamma < 1$, $V^{(n+1)}$ converges to a $V^{\star}_{\gamma}$ that satisfies, for all $s \in S$, $V^{\star}_{\gamma}(s) = \max_{a\in A(s)} \left(R(s,a) + \gamma \sum_{s'} p(s'|s,a)V^{\star}_{\gamma}(s')\right)$. This method has time complexity $O(|S|^2|A|)$ and space complexity $O(|S|)$. As γ approaches 1, accuracy should improve, as the operation resembles a finite-horizon MDP iteration with horizon tending to infinity. However, the rate of convergence gets significantly slower as γ approaches 1. This method is expected to not yield good results in the case of Non-Stationary MDPs, as it returns a stationary policy without taking the time dependence into account.

Turnpike-based approximation. This method lets the agent calculate the policy up to the horizon's length N and then use this last policy array (of index N) in every step. With a sufficiently large horizon and stationary MDP, we expect the total reward collected to be slightly less than that collected by an FHMDP. This algorithm has the same time complexity as the exact, state-of-the-art, *standard* finite-horizon solution [3], $O(|S|^2|A|N)$, but space complexity $O(|S|)$. It is expected to yield suboptimal results in the case of Non-Stationary MDPs, as it calculates a stationary policy.

Standard solution [3]. This method (Section 3.1) saves every list of values for every number of remaining steps, yielding space complexity $O(|S|N)$ and time complexity $O(|S|^2|A|N)$; it is the sate-of-the-art value iteration method for finite-horizon MDPs.

InPlace solution. This algorithm (Section 3.2) calculates values and actions to be made for the FHMDP problem utilizing the least possible space (Section 3.2), by recalculating arrays when needed, with each new array overwriting the previous. The space complexity is $O(|S|)$ and time complexity $O(|S|^2|A|N^2)$.

Radical solution. This approach (Section 4.1) stores arrays corresponding to multiples of the square root of the horizon N and treats each interval of length $\lfloor\sqrt{N}\rfloor$ as the state-of-the-art *standard* solution [3]; it lowers space complexity to $O(|S|\sqrt{N})$ while retaining time complexity $O(|S|^2|A|N)$.

Logarithmic solution. This algorithm (Section 4.2) achieves greater space complexity reduction at a small time complexity overhead; it stores arrays along a hierarchy to the index needed. Space is $O(|S|\log_2 N)$ and time $O(|S|^2|A|N\log_2 N)$.

5.3 Reward Comparison, Stationary Case

Here, we compare approximations of the FHMDP solution to the exact one with regard to the total reward collected. We use a random number generator whenever needed for the training and decision-making purposes. In every execution, we provide a seed to the model to ensure result consistency. Naturally, for a particular seed, *all* exact *finite-horizon* solutions (i.e., in-place, standard, square-root, and logarithmic) collect *the same* total reward. Therefore, we present one representative for all finite-horizon solutions.

The reward collected *in practice* may differ from its *expected* value. To attenuate this effect, we evaluate rewards over 20 runs with different seeds. Further, we use 10,000 training steps, 1–20 VMs with initial value 10, hence 200 states, and e-probability 0.7, 500 steps before next value iteration during training, γ in {0.1, 0.3, 0.99} and horizon in geometric growth {100, 200, 400, …819200}.

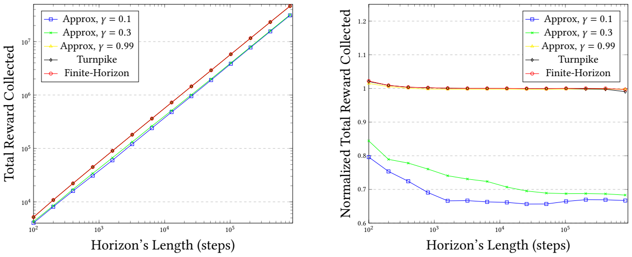

Figure 4: L: average total reward; R: collected over expected.

Figure 4 presents the actual rewards measured. The highest total reward arises using a Finite-Horizon algorithm. Approximation with low γ (i.e., 0.1 and 0.3) yield lower rewards than those with high γ. They achieve best results for low values of the horizon, as the normalized results on the right figure show. Yet, with high γ the approximation obtains results almost identical to those of the exact solution. This result is in accordance with the *turnpike theorem*, which states that there exists a time step n, after which the policy is stationary, which depends on γ. Besides, the turnpike approximation achieve high quality. This result suggests that the policy of this model is almost stationary, as there are many states and a small number of available actions (at most 3).

These results suggest that approximations can yield good results while consuming less resources, especially as a long horizon is likely to contain the *turnpike integer*. Indeed, a good way to solve an FHMDP would be to use the turnpike approximation up to the turnpike integer and the standard solution in remaining steps, if we could calculate the turnpike integer; in cases of large horizon, we may simply use the turnpike approximation in all steps. However, as we will see next, this approach falters in *non-stationary* MDPs.

5.4 Reward Comparison, Non-stationary Case

We saw that a turnpike approximation can be as good as an exact solution when the MDP is *stationary*. Here, we examine the *non-stationary* case. We simulate variable rewards as follows: in each time step n, state s, and (available) action a, the reward the agent receives after transiting to a neighboring state s' is zero if the index of s' is $n \mod |S_{neighbors}|$. In each time step, only one reward is affected. The model parameters are otherwise as before.

The left side in Figure 5 presents the average reward collected for each horizon value over 20 runs, for the *exact* solution, *infinite-horizon-based* approximations with $\gamma \in \{0.1, 0.3, 0.99\}$ and the turnpike approximation. As the difference between exact and approximate solutions appears small, we normalize them by the corresponding average expected reward by the *exact* solution, shown on the right in Figure 5. The exact solution outperforms others, with reward almost identical to expectation. Contrary to results with constant rewards, the approximation with $\gamma = 0.99$ cannot hanlde reward variability and thus collects a total below 90% of the optimal. The turnpike approximation oscillates at around 75% of the optimal, even worse than the approximation with $\gamma = 0.99$.

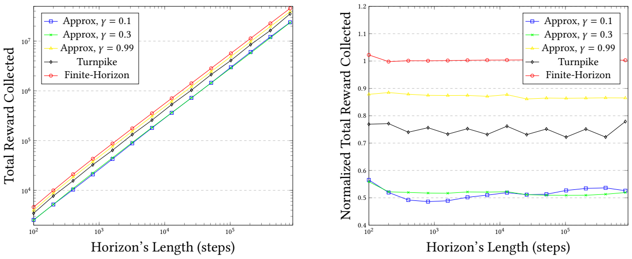

Figure 5: Reward collected, non-stationary case.

Next, we increase the maximum number of VMs to 40 and the number of VMs that can be added or removed per action to 10, yielding approximately 21 actions per state vs. the initial 3. We simulate variable rewards in a round-robin fashion, conferring a *bonus* reward if the selected action ID α equals the time step n mod the total number of actions at that state S_α. Figure 6 presents our results on the ratio of cumulative reward collected by the $\gamma = 0.99$ and turnpike approximations over that of the exact solution, with bonus rewards $\in [10, 25, 50, 100]$. Notably, approximation algorithms cannot adapt to the non-stationary model, whereas the exact solution benefits from the situation, especially as the bonus reward grows.

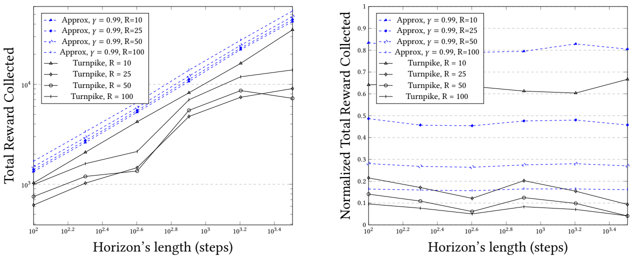

Figure 6: Reward by stationary policy over optimal, 400 states.

We conclude that, on NSMDPs, the exact finite-horizon solution yields results no approximation can provide; this finding exemplifies the necessity for practicable, space-efficient *exact* FHMDP solutions.

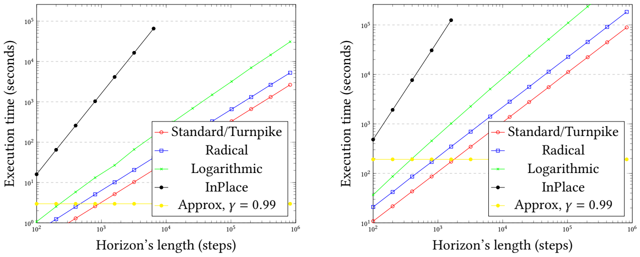

Figure 7: Execution time. Left: 200 states. Right: 2000 states.

5.5 Execution Time

Here, we assess the runtime for policy evaluation. The parameters of the model are as before, while we try 200 and 2000 states. Figure 7 presents, in logarithmic scales, results for four *exact* solutions, in-place, standard, radical, and logarithmic, and the two approximations, approximate and turnpike. The in-place algorithm yields unscalable runtime, while the approximate ones run in time independent of the horizon; turnpike requires the same time as the standard solution, hence we present them as one curve. As we have established, these approximations yield suboptimal reward. Our results verify the analysis in Section 3: the radical solution shows the same runtime behavior as standard, while logarithmic exhibits a negligible overhead due to its logarithmic time complexity factor.

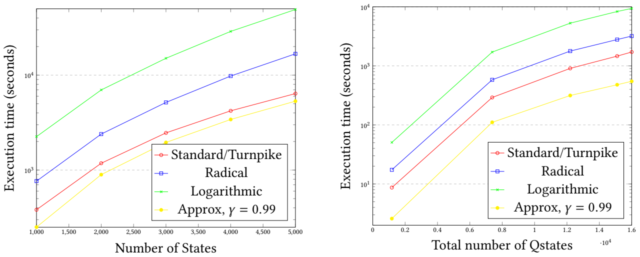

Figure 8: Time vs. states, available actions; horizon 1600.

Next, we fix horizon length to 1600 and increase states under 5 actions, and actions under 400 states; to increase actions, we set the maximum number of VMs to 40 and vary the maximum number of VMs that can be added or removed per action $\in [1, 10, 20, 30, 40]$, yielding 1180 to 16000 Qstates (state-action pairs). The runtime results in Figure 8 shows similar growth trends for all methods.

5.6 Memory Consumption

We have established that SIFTER solutions yield superior rewards, especially in non-stationary problems, with no major runtime disadvantage. Now we turn to our main focus, space efficiency. We measure and compare the memory needs of all exact and approximate solutions, using the same model parameters as before.

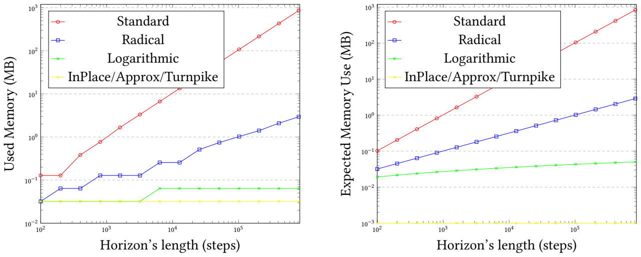

Figure 9: Memory use. L: actual. R: expected. 200 states.

Figure 9 reports on a model with 200 states. The graph on the left shows memory consumption, and the one on the right shows expected memory needs. Practical measurment agrees with theory, apart from a few flat points. This is because memory is assigned by the system in the form of pages; occasionally, the extra memory required is available within the same page. With the logarithmic solution, memory use appears constant, with a slight increase from $N = 3200$ to $N = 6400$; this result indicates that, until $N = 3200$ a constant number of pages is sufficient, while at $N = 6400$, another page is needed. The standard solution requires the highest memory, growing linearly with the horizon, even while its execution time is matched by the logarithmic solution, as we observed. The radical solution stands between those two, raising lower memory needs than the standard solution, while requiring asymptotically the same execution time. The in-place, approximate, and turnpike solutions have constant space complexity vs. horizon length. However, as we have seen, approximate and turnpike yield suboptimal reward, while in-place is highly unscalable in runtime.

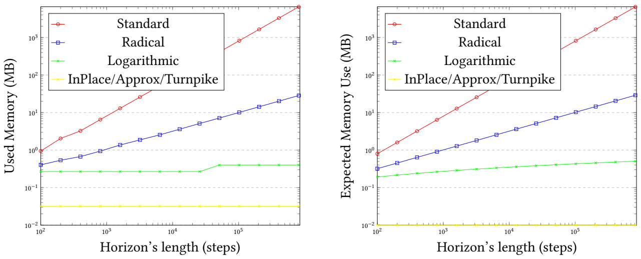

Figure 10: Memory use. L: actual; R: expected. 2000 states.

Figure 10 presents results on 2000 states. Now the horizon's growth induces greater growth in memory needs, hence the memory curves of the standard and radical solutions grow smoothly. Logarithmic still requires tiny amounts of memory, almost independent of the horizon. A constant amount of pages appears to suffice when the horizon ranges from 100 to 25,600 steps. Thereafter, new pages are required, but their number stabilizes again.

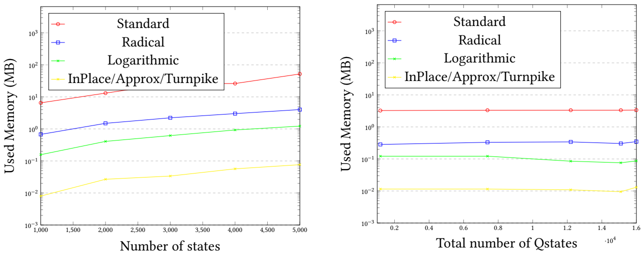

Figure 11: Memory vs. states, available actions; horizon 1600.

Figure 11 reports memory consumption with increasing number of states and actions, under the configuration of Section 5.5; unsurprisingly, the advantage of SIFTER methods remains.

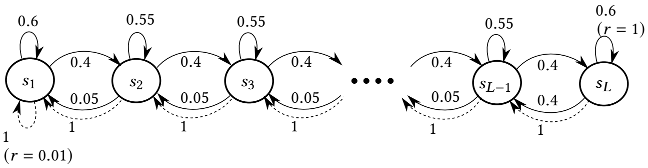

Figure 12: The RiverSwim MDP [27].

5.7 RiverSwim-based MDPs

FHMDPs are most relevant when confronting a dilemma between a low, easily accessible reward and a high, far-reach one. Such scenarios arise in healthcare (e.g., waiting queues in hospitals), transportation (e.g., choosing among transportation means under time constraints), IoT systems (e.g., managing sensor data under battery constraints). Decisions may be repeated in episodes [1, 24], thus even a small loss of reward may yield significant cumulative loss, as in regret minimization [5, 6, 10, 16] where performance is measured against an oracle following an *optimal* policy.

We report on an experiment that simulates the aforementioned scenarios based on the *RiverSwim* MDP [27], a reinforcement learning benchmark [5, 13, 24, 28] shown in Figure 12: in each out of $|S| \geq 2$ states, the agent may swim `right` (against a current) or `left`. Transitions are shown in solid arrows for right actions and dashed for left; `left` always succeeds, while `right` in $s \neq s_1, s_L$ fails with probability (w.p.) 0.55 and relapses backwards w.p. 0.05, otherwise succeeds; `right` in $s = s_1$ succeeds w.p. 0.6; the bank s_L is slippery, hence `right` may lead back into the river w.p. 0.4. The agent starts in s_1 and executes N steps, accruing reward 0.01 in s_1 and 1 in s_L. We experimented with $|S| = 1000$ and horizon $N \in [2000, 4000]$. Figure 13 depicts the memory-time tradeoff and the ratio of the expected accumulated reward under the turnpike approximation over the exact solution. Remarkably, the advantage of the exact solution over turnpike exceeds 38% for horizon $N = 2870$, whereas the memory-time tradeoff reconfirms that the logarithmic solution raises practically constant memory needs with a manageable runtime overhead. We also tried a variant where, upon a `right` action, the agent moves backwards w.p. 0.04 (instead of 0.05) and relapses to state $^i/_2$ w.p. 0.01. Figure 14 shows the tradeoff and reward ratio in this case; the deviation is even more striking, further verifying the need for exact solutions.

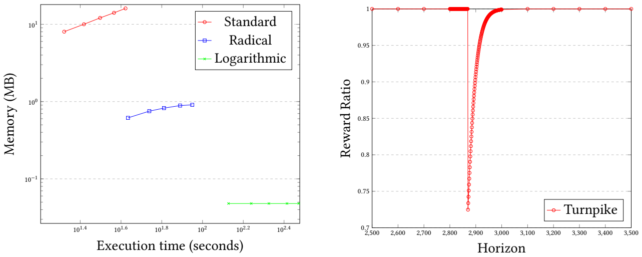

Figure 13: RiverSwim MDP, 1000 states.

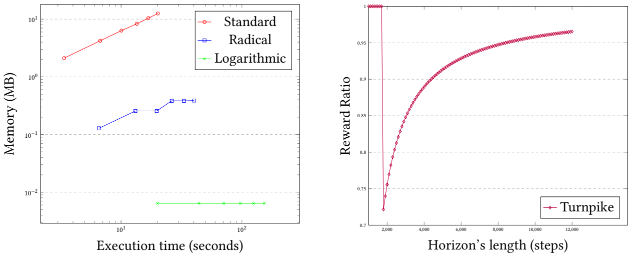

Figure 14: RiverSwim variant, 200 states.

6 CONCLUSION

We studied the question of space efficiency in optimization with finite-horizon MDPs. We proposed SIFTER, a suite of two algorithms that require drastically lower memory with no or negligible runtime overhead. We examined the behavior of the SIFTER algorithms in terms of collected reward under diverse settings, compared with infinite-horizon-based and turnpike-based approximations. Our results show that, while approximations may yield viable solutions on stationary finite-horizon models, they fail to do so on non-stationary ones, rendering exact solutions for FHMDPs a necessity. SIFTER algorithms provide such exact solutions while conferring scalability in terms of memory requirements that the state of the art had been wanting since 1957.

REFERENCES

[1] Mohammad Gheshlaghi Azar, Ian Osband, and Rémi Munos. 2017. Minimax Regret Bounds for Reinforcement Learning. In *ICML (Proc. of Machine Learning Research, Vol. 70)*. 263–272.

[2] Nazila Bazrafshan and M. M. Lotfi. 2020. A finite-horizon Markov decision process model for cancer chemotherapy treatment planning: an application to sequential treatment decision making in clinical trials. *Annals of Operations Research* 295, 1 (2020), 483–502.

[3] Richard E. Bellman. 1957. A Markovian Decision Process. *Journal of Mathematics and Mechanics* 6, 5 (1957), 679–684.

[4] Dimitri P. Bertsekas. 2017. *Dynamic programming and optimal control* (4th ed.). Vol. 1. Athena Scientific.

[5] Hippolyte Bourel, Odalric Maillard, and Mohammad Sadegh Talebi. 2020. Tightening Exploration in Upper Confidence Reinforcement Learning. In *ICML (Proc. of Machine Learning Research, Vol. 119)*. 1056–1066.

[6] Apostolos N. Burnetas and Michael N. Katehakis. 1997. Optimal adaptive policies for Markov decision processes. *Mathematics of Operations Research* 22, 1 (1997), 222–255.

[7] Dan Calderone and S. Shankar Sastry. 2017. Markov Decision Process Routing Games. In *8th Intl Conf. on Cyber-Physical Systems (ICCPS)*. 273–279.

[8] Tang Lung Cheung, Kari Okamoto, Frank Maker, Xin Liu, and Venkatesh Akella. 2009. Markov Decision Process (MDP) Framework for Optimizing Software on Mobile Phones. In *7th ACM Intl Conf. on Embedded Software (EMSOFT)*. 11–20.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[10] Wesley Cowan and Michael N. Katehakis. 2020. Exploration–exploitation policies with almost sure, arbitrarily slow growing asymptotic regret. *Probability in the Engineering and Informational Sciences* 34, 3 (2020), 406–428.

[11] Peng Dai, Daniel S Weld Mausam, and Daniel S Weld. 2008. Partitioned External-Memory Value Iteration.. In *AAAI*. 898–904.

[12] Stefan Edelkamp, Shahid Jabbar, and Blai Bonet. 2007. External Memory Value Iteration. In *17th Intl Conf. on Automated Planning and Scheduling (ICAPS)*. 128–135.

[13] Sarah Filippi, Olivier Cappé, and Aurélien Garivier. 2010. Optimism in reinforcement learning and Kullback-Leibler divergence. In *48th Annual Allerton Conf. on Communication, Control, and Computing (Allerton)*. 115–122.

[14] Yasin Gocgun, Brian W. Bresnahan, Archis Ghate, and Martin L. Gunn. 2011. A Markov decision process approach to multi-category patient scheduling in a diagnostic facility. *Artificial Intelligence in Medicine* 53, 2 (2011), 73–81.

[15] Jason H. Goto, Mark E. Lewis, and Martin L. Puterman. 2004. Coffee, Tea, or …?: A Markov Decision Process Model for Airline Meal Provisioning. *Transportation Science* 38, 1 (2004), 107–118.

[16] Thomas Jaksch, Ronald Ortner, and Peter Auer. 2010. Near-optimal Regret Bounds for Reinforcement Learning. *J. Mach. Learn. Res.* 11 (2010), 1563–1600.

[17] Lodewijk Kallenberg. 2020. *Lecture Notes Markov Decision Problems*. https://www.math.leidenuniv.nl/~kallenberg/Lecture-notes-MDP.pdf

[18] Panagiotis Karras and Nikos Mamoulis. 2008. Hierarchical synopses with optimal error guarantees. *ACM Trans. Database Syst.* 33, 3 (2008), 18:1–18:53.

[19] Erwan Lecarpentier and Emmanuel Rachelson. 2019. Non-Stationary Markov Decision Processes, a Worst-Case Approach using Model-Based Reinforcement Learning. In *NeurIPS*. 7214–7223.

[20] Mark E. Lewis and Anand A. Paul. 2019. Uniform Turnpike Theorems for Finite Markov Decision Processes. *Math. Oper. Res.* 44 (2019), 1145–1160.

[21] Yuxi Li. 2019. Reinforcement Learning Applications. arXiv:1908.06973 [cs.LG]

[22] Konstantinos Lolos, Ioannis Konstantinou, Verena Kantere, and Nectarios Koziris. 2017. Elastic management of cloud applications using adaptive reinforcement learning. In *IEEE Intl Conf. on Big Data (BigData)*. 203–212.

[23] Sadegh Nobari, Panagiotis Karras, HweeHwa Pang, and Stéphane Bressan. 2014. *L*-opacity: Linkage-Aware Graph Anonymization. In *EDBT*. 583–594.

[24] Ian Osband, Daniel Russo, and Benjamin Van Roy. 2013. (More) Efficient Reinforcement Learning via Posterior Sampling. In *NeurIPS*. 3003–3011.

[25] Martin L. Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[26] Jeremy F. Shapiro. 1968. Turnpike Planning Horizons for a Markovian Decision Model. *Management Science* 14, 5 (1968), 292–300.

[27] Alexander L. Strehl and Michael L. Littman. 2008. An analysis of model-based interval estimation for Markov decision processes. *J. Comput. System Sci.* 74, 8 (2008), 1309–1331.

[28] Mohammad Sadegh Talebi and Odalric-Ambrym Maillard. 2018. Variance-Aware Regret Bounds for Undiscounted Reinforcement Learning in MDPs. In *Algorithmic Learning Theory (Proc. of Machine Learning Research, Vol. 83)*. 770–805.

[29] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. 2018. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *Proc. VLDB Endow.* 11, 12 (2018), 2074–2077.

[30] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.* 46, 3 (2021), 9:1–9:45.

[31] D. J. White. 1993. A Survey of Applications of Markov Decision Processes. *The Journal of the Operational Research Society* 44, 11 (1993), 1073–1096.